Big Data: análisis y visualización de grafos utilizando Neo4j

Harún Tilmatine Pérez

Resumen– En los últimos años la sociedad ha ido evolucionando a pasos agigantados junto de la mano de las nuevas tecnologías. Hace unos pocos años la cantidad de personas que disponían de conexión a Internet no era muy alta, y la vida social en la red se limitaba a unas salas de chat. A lo largo de estos años las redes sociales han ido evolucionando cambiando la forma de interactuar con los demás y son cada vez más populares, siendo extremadamente difícil encontrar una persona que no sea usuario de alguna. La información es poder, y la cantidad de datos que generan estas redes sociales es descomunal, el problema es la capacidad analizar tal cantidad de datos. En este proyecto se verá un procedimiento de análisis de la red social Twitter.

Palabras clave– Neo4J, Twitter, Crawler, Visualización, D3.js, grafos.

Abstract– During the last few years society has been evolved a lot with new technologies. A few years ago the number of people who had Internet connection was very low and the social life in the network was limited to a chat room. Through the years, social networks have improved so much by changing the way to interact with other users and now they are incredibly popular, being extremely difficult to find a person who is not a user of some social network. Information is power, and the amount of data generated by these social networks is huge, the problem is the ability to analyze so much data. In this project will see how to analyze the social network Twitter.

Keywords– Neo4J, Twitter, Crawler, Visualization, D3.js, graph.

✦

1 INTRODUCCIÓN

EL proyecto que presentaremos a lo largo de este docu-mento tratará sobre el estudio e implementación de nuevas tecnologías aplicadas al mundo del Big Data. Con el principal objetivo de mostrar grandes cantidades de datos en forma de grafo.
Este proyecto nace en aras de la necesidad del conocer los pensamientos de la población aprovechando los recursos que nos ofrecen las nuevas tecnologías. Saber en qué lugares se habla más del producto de una empresa, para reforzar el marketing en aquellos en los que se habla menos. Conocer el rango de edad mayoritario de un partido político. Averiguar los usuarios que podrían ser de utilidad para utilizarlos como herramientas de marketing, según sus conexiones con potenciales clientes. Son muchos los usos que se le podrían dar a este proyecto haciendo pequeñas modificaciones para adaptarlo a nuestras necesidades.
En este documento se explicará el procedimiento que hemos seguido para analizar la estructura de la red social Twitter. Desde su inicio en el que crearemos un crawler para la recopilación de datos, la estructura de la base de datos que crearemos en Neo4J, la depuración necesaria de los datos recolectados para ser almacenados en nuestra base de datos, la exportación de los datos almacenados en ésta y finalmente el proceso seguido para su posible visualización en forma de grafo.
Este proyecto podrá ser utilizado por todos aquellos que quieran introducirse en el mundo del big data orientado a las redes sociales.

• E-mail de contacte: harun.tilmatine@gmail.com
• Mención realizada: Tecnologías de la Información.
• Trabajo tutoritzado por: Jordi Casas-Roma (Departamento de Ingeniería de la Información y de las comunicaciones).
• Curs 2015/16

2 ALCANCE DEL PROYECTO

2.1. Objetivo general

El objetivo del proyecto es ser capaz de visualizar en forma de grafo una pequeña parte de la red de Twitter, para ser capaces de apreciar las conexiones entre los distintos

tipos de estructuras de datos que alberga esta red social

2.2. Objetivos específicos

Los objetivos específicos que nos hemos marcado son:

- (O1) Investigación de los diferentes métodos de creación de un crawler y su desarrollo. El crawler deberá recopilar datos utilizando como criterio de búsqueda un hashtag. El desarrollo se hará preferiblemente en lenguaje Java.
- (O2) Instalación del entorno y creación de la estructura que contendrá nuestra base de datos en Neo4J.
- (O3) Almacenar cierta cantidad suficiente de datos de Twitter en nuestra base de datos para sea possible examinar el alto grado de conectividad de una red social.
- (O4) Investigación de los diferentes métodos de visualización de grafos.
- (O5) Creación de una página web para visualizar el grafo.
- (O6) Mejoras visuales y funcionales de la página web.

2.3. Prioridad de los objetivos

Como podemos observar en la tabla 1, hemos clasificado nuestros objetivos según su prioridad.

TABLA 1: PRIORIDADES DE LOS OBJETIVOS

Clasificación de los objetivos			
Objetivos	Crítico	Principal	Secundario
O1.	X		
O2.	X		
O3.		X	
O4.		X	
O5.			X

3 CONCEPTOS BÁSICOS

3.1. Big Data

Desde hace unos años las palabras Big Data son cada vez más nombradas.
En los últimos años la cantidad de dispositivos inteligentes que forman parte de la población ha sufrido un aumento espectacular. En 2011 la cantidad de smartphones vendidos fueron de 466M, en 2012 de 680M, en 2013 de 968M y en 2014 de 1.245M[1].
Podemos hacernos una idea del incremento de dispositivos inteligentes, donde no sólo ha habido un crecimiento de smartphones, sino de muchos otros tipos de dispositivos como tablets, diferentes tipos de sensores, etc.
Todos estos dispositivos transmiten un gran volumen de informaci ón, en diferentes formatos (audio, video, sistemas GPS..) y además algunos requieren de una gran velocidad de respuesta. Aquí es cuando entra en escena Big Data. El desafío de Big Data se basa en el modelo de las 3Vs (volumen, variedad y velocidad) [2].
Para este proyecto el uso de Big Data es imprescindible. Al hacer un crawling de Twitter obtendremos un gran volumen de datos que con el que Big Data nos proporciona un rendimiento superior a las soluciones tradicionales. Aunque para este proyecto una solución convencional también podría funcionarnos ya que por una limitación de recursos sólo se hará un crawling de una parte de Twitter y cuando se habla de un gran volumen de datos se suele hablar de petabytes o exabytes.
Videos, comentarios, imágenes, fechas, lugares.. Se recogerán diferentes tipos de datos por lo que sacaremos rendimiento de otra de sus principales características. Teniendo en cuenta que Twitter requiere de una escalabilidad enorme habrá que elegir una sistema de almacenamiento adecuado.

3.2. Bases de datos no relacionales

Hay diferentes sistemas de almacenamiento que no cumplen el esquema de entidad-relación sino que son mucho más flexibles y concurrentes permitiéndonos manipular grandes cantidades de datos mucho más rápidamente [3].

Podemos distinguir cuatro grupos de base de datos NoSQL:

- Almacenamiento Documental.
- Almacenamiento Clave-Valor.
- Almacenamiento en Grafo.
- Almacenamiento Orientado en Columnas.

Para este proyecto necesitamos un almacenamiento en grafo.
El almacenamiento en grafo se basa en la teoría de los grafos. Un grafo no es mas que un conjunto de objetos a los que llamamos nodos que están conectados entre ellos y a esas conexiones las llamamos aristas.
Con este tipo de almacenamiento podríamos representar los usuarios de Twitter y sus relaciones mucho más fácilmente.
La solución que utilizaremos (Neo4j) es un software de almacenamiento en grafo.

4 ESTADO DEL ARTE

Este proyecto se compone de tres subproyectos: creación del crawler, la base de datos en Neo4J y la parte de visualización de grafos.

Observando la figura 1 podemos hacernos del conjunto del sistema y de las tecnologías aplicadas en los tres subproyectos. Para la recolección de *tweets* hemos hecho uso de la librería Twitter4J. En cuanto a su almacenamiento tal y como hemos comentado anteriormente, hemos optado por Neo4J. Para la visualización del grafo, hemos utilizado D3.JS, una librería de JavaScript.
Es difícil encontrar una aplicación que haga lo mismo de la misma manera. Pero si hemos encontrado diferentes formas de implementar cada subproyecto llegando a los mismos resultados.

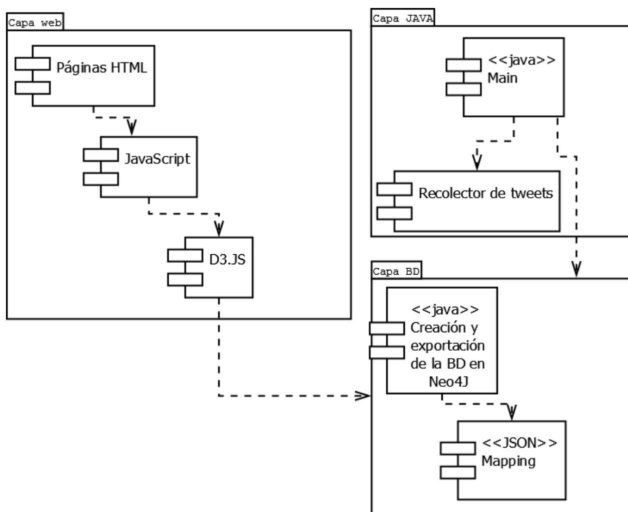


Fig. 1: *Diagrama de componentes.*

A continuación hablaremos de los diferentes métodos que hemos ido encontrando durante nuestra investigación y podrían haber sido válidos para nuestros objetivos.

4.1. Creación de un crawler mediante Tweepy

Tweepy es una librería de Pyhton que nos permite acceder a Twitter [4]. Nos da acceso a todos los métodos de la REST API de Twitter y nos permite trabajar con los datos extraídos envueltos en objetos Pyhton fácilmente accesibles.
Podríamos haber utilizado esta librería perfectamente pero como disponemos de mayor experiencia con lenguaje Java optamos por escoger la librería Twitter4J.

4.2. Creación de la base de datos

Hemos comentado anteriormente que para nuestro proyecto lo ideal sería utilizar una base de datos en grafo. Son varias las opciones que disponemos, como InfiniteGraph o OrientDB, pero no hemos dudado en decantarnos por Neo4J. La razón es que es una de las más conocidas. Por lo que nos resulta mucho más fácil documentarnos al haber mucha más información en la red.

4.3. Visualización de grafos con Gephi

Gephi es un software de código abierto que nos permite analizar redes en forma de grafo [5]. Es una herramienta con una curva de aprendizaje baja y con bastantes opciones de personalización.
Cumple con nuestro objetivo principal de visualizar nuestros datos en forma de grafo y además la implementación es sencilla. ¿Entonces por qué no hemos utilizado Gephi? Nos hemos decantado por D3.js porque el plugin de Gephi destinado al soporte de Neo4 [6], dejó de actualizarse para las últimas versiones de Gephi. También queríamos cumplir con el objetivo de crear una página web en la que visualizar el grafo y aprender a utilizar librerías punteras. Por estas razones, y aunque la dificultad de implementación de una librería como D3.js es mucho más alta, creemos que estos conocimientos nos han aportado más que aprender a utilizar un software ya desarrollado.

5 Metodología

5.1. Planificación

Como hemos comentado anteriormente, este proyecto se podría dividir fácilmente en varios subproyectos, por lo que hemos seguido un desarrollo iterativo e incremental. En cada etapa se han desarrollado diferentes funciones del sistema y en la siguiente fase durante su implementación si era necesario se revisaba la fase anterior.
En la figura 2 podemos observar el proceso que hemos seguido para desarrollar nuestro proyecto.

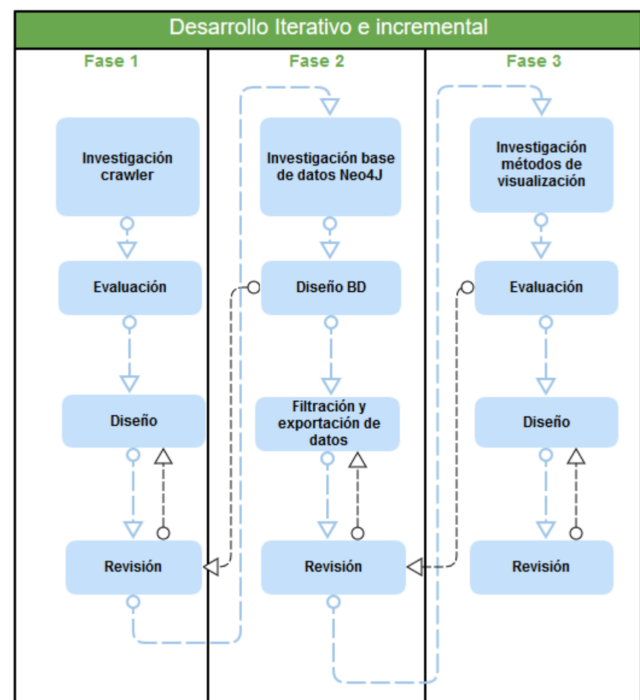


Fig. 2: *Modelo de desarrollo.*

5.1.1. Fase 1

Durante la primera fase se investigó cómo podríamos crear un crawler que fuera capaz de recolectar datos de Twitter. Encontramos que la librería Tweepy de Python y la librería Twitter4J eran las más adecuadas para nuestros requisitos.

Durante la etapa de evaluación optamos por utilizar Twitter4J ya que tenemos más experiencia desarrollando en lenguaje Java.
Durante la etapa de diseño creamos y recolectamos los datos que creíamos que necesitaríamos después de haber estudiado la estructura de Twitter y los diferentes métodos que nos proporciona Twitter4J.
Durante la etapa de revisión optimizamos el tiempo de recolección de *tweets* convirtiendo la aplicación en un procedimiento de ejecución multihilo.

5.1.2. Fase 2

En esta segunda fase se estudió en un principio sobre las bases de datos no relacionales para después profundizar en Neo4J. Estudiamos Cypher Query Language el lenguaje que utiliza Neo4J.
En el diseño de la base de datos definimos los diferentes tipos nodos que conformarían nuestro grafo. Durante esta etapa tuvimos que revisar la fase 1 ya que estábamos recolectando datos innecesarios y además algunos en un tipo de objeto que Neo4J no admite.

5.1.3. Fase 3

En la tercera fase lo primero que hicimos fue investigar los métodos viables de visualización de grafos. Como queríamos ser partícipes del desarrollo de una herramienta de visualización de grafos implementada en una página web, decidimos documentarnos de las diferentes librerías para lenguajes de programación web en vez de utilizar Gephi.
Habíendonos documentado brevemente, sabíamos que lo primero que necesitábamos era exportar nuestro grafo a algún tipo fichero de datos. Por lo que durante esta etapa filtramos los datos que pensábamos que podrían ser útiles para la visualización y los exportamos a un fichero JSON.
Durante la fase de evaluación optamos por D3.js y vimos que necesitábamos hacer unos cambios en nuestros ficheros JSON para adecuarlos a las necesidades de D3.js, por lo que tuvimos que revisar la fase 2.
En el diseño creamos un html que nos permitía visualizar nuestros datos en forma de grafo animado con el que es posible interactuar arrastrando los diferentes nodos. Durante la revisión decidimos añadirle una funcionalidad a nuestra página web, permitiendo que el usuario pueda hacer click en los nodos, y ver en una tabla toda la información correspondiente a éste.

5.1.4. Tabla de planificación

En la tabla 2 hemos indicado las fechas de inicio, fin y el tiempo de desarrollo que nos ha llevado cada una de las fases.

TABLA 2: TABLA DE PLANIFICACIÓN

Tiempos de desarrollo por fases			
Objetivos	Inicio	Final	Duración
Fase 1.	09/02/2016	17/04/2016	68 días.
Fase 2.	17/04/2016	22/05/2016	35 días.
Fase 3.	22/05/2016	10/06/2016	19 días

5.2. Estructura de nuestra base de datos

Nuestro estudio deberá responder a diferentes preguntas por lo que necesitaremos almacenar la mayor cantidad de datos posible.

A primera vista hay dos nodos evidentes: *Tweet* y *Usuario*. Sabemos que hay tres tipos diferentes de mensajes pero tanto el *reply* como el *retweet* no dejan de ser *tweets* con un *tweet* destinatario (*reply*) o del que derivan (*retweet*). Por lo que decidimos crear un único nodo *Tweet*, que contiene como propiedades un campo llamado *isRetweet* y otro campo *idReplyUser*, que contiene el identificador del usuario al que hace referencia el *reply*. Si por lo tanto esta última propiedad no contiene ningún identificador, se concluye que no es un *reply*. En las aristas indicaremos la relación entre los *tweets* y los usuarios.

Para poder responder preguntas como:

- ¿Cuál es el idioma más utilizado?
- ¿Desde dónde se escribe la mayor cantidad de mensajes?

Nos ayudaría tener un nodo *Language* y *Place*. En el nodo *Place* guardamos toda la información disponible como la ciudad, el país o el tipo de lugar. También sería posible almacenar la localización del usuario, pero esto no nos resultaría útil ya que Twitter no verifica estos campos y muchos usuarios utilizan este campo para poner bromas o ciudades en las que realmente no viven.

Buscando por un *hashtag* podríamos encontrar diez mil *tweets*, de estos diez mil, podría haber mil que fueran *reply* a otros *tweets*, estos otros *tweets* podrían ser un reply a otros *tweets* y éstos a otros, etc. Twitter es una red altamente conectada por lo que hemos puesto una limitación a la búsqueda de *tweets* creando un nodo adicional al *User* llamada *RelevantUser*. Este nodo contiene toda la información que necesitamos para el estudio mientras que el nodo *User* solamente contiene el identificador del usuario. Así pues, cuando se haga la búsqueda de *tweets* si se hace una limitación a mil *tweets*, los autores de los mil *tweets* serán representados por los nodos *RelevantUser*. De estos mil *tweets* los que fueran un *reply* o *retweet* se buscaría a su autor que sería representado solamente con el nodo *User*.

¿Por qué no buscamos también los *tweets* a los que pueden hacer referencia? Con el nodo *Tweet* dudábamos entre guardar el identificador del *tweet* al que uno de nuestros *tweets* encontrados podía hacer referencia ya sea mediante un *reply* o un *retweet*, o guardar el identificador del usuario que haya escrito el *tweet* referenciado. Al final hemos optamos por guardar el identificador del usuario ya que creemos que para el uso que le daremos a la aplicación no es necesario almacenar también las relaciones entre *tweets*. Si en un futuro se requiere de esta característica se añadiría solamente una relación del nodo *Tweet* al nodo *Tweet*, independientemente de si el *tweet* referenciado es por un *retweet* o *reply*, ya que un *tweet* sólo puede ser un tipo (*retweet*, *reply* o sin ningún tipo referencias).
En el anexo A.1 podemos ver un diagrama de clases de nuestra base de datos. Se puede observar que hay relaciones duplicadas. Esto es así ya que en Neo4j las relaciones se hacen con una dirección única por lo que las relaciones bilaterales no son posibles.

5.3. Creación del crawler

Hay varias formas de buscar *tweets*, nosotros utilizaremos la librería para Java creada por Yusuke Yamamoto [7].

Creamos un proyecto Maven [8] y añadimos en el pom la dependencia con la versión 4.0.4 de Twitter4J.
El siguiente paso es utilizar el constructor *ConfigurationBuilder* para configurar Twitter4J con nuestras opciones deseadas. Un requisito que nos impone Twitter es registrar nuestra aplicación para poder autenticar nuestras peticiones. Para eso nos solicita unas claves secretas que se nos proporcionan al registrar la aplicación en la página de *apps* de Twitter [9] (se necesita tener una cuenta de Twitter para poder acceder a la página). Una vez registrada nuestra aplicación configuraremos nuestro constructor *ConfigurationBuilder* y sus propiedades *OAuthConsumerKe*, *OAuthConsumerSecret*, *OAuthAccessToken* y *OAuthAccessTokenSecret* con los valores que Twitter nos ha proporcionado (a estas claves se las conoce como *tokens*).
Una vez que tengamos los tokens de acceso ya podremos lanzar *querys* a Twitter. Según nuestras necesidades utilizaremos los diferentes operadores que acepta Twitter [10]. Por ejemplo podemos lanzar una *query* que busque con el *hashtag* de la película ”The Revenant”simplemente lanzando una *query* con un *String* con el valor: revenant”. Si queremos buscar *tweets* que contengan la palabra ”tiempoz sean una pregunta haríamos una *query* tal que: ”tiempo ?”.
Nuestra *query* nos devuelve una lista de objetos de tipo *Status* [11]. A partir de este objeto podremos obtener todos los datos que necesitamos para lograr nuestros objetivos.

En en anexo A.2 nos encontramos con el objeto *Status* y el resto de objetos que nos proporciona y que hemos utilizado para extraer datos. Podemos observar que el objeto *Status* sería un equivalente a un *tweet* ya que contiene toda su información como su texto, identificador o fecha de creación.
Llamando al método *getPlace()* obtenemos todos los datos relacionados con el lugar donde se ha escrito el tweet. Esta objeto será almacenado en nuestra base de datos como un nodo tipo *Place* aunque por los requisitos del proyecto no nos será necesario guardar todos los campos.
Hemos podido observar en el digrama de clases de la base de datos que no existe un nodo que represente la clase *Geolocation*. Esto es así porque optamos por almacenar la latitud y longitud en el propio nodo *Tweet*, ya que no íbamos a usar la geolocalización para los lugares creímos que no aportaría nada a nuestro grafo y que nos acabaría perjudicando por el hecho de haber más tipos de nodos, lo que resultaría en una mayor dificultad de visualización del grafo.
El objeto usuario es el que más nos ha costado de mapear. Recordemos que tenemos dos tipos de nodo usuario: *User*, *RelevantUser*. Cada *tweet* (*Status*) tendrá un usuario autor que será un nodo *RelevantUser*. ¿En qué casos se dará que crearemos un nodo tipo *User*? En el caso de que sea un *reply* se creará un nodo tipo *User* con su identificador. También en un *tweet* nos podemos encontrar que en el texto mencionan a otros usuarios utilizando una arroba y el nombre de usuario, para este caso también crearemos otro nodo *User*.
A la hora de mapear el nodo *RelevantUser* vemos que existen los métodos *getFollowersCount()* y *getFollowingCount()* que devuelven un objeto *Integer* con el número de seguidores y seguidos del usuario en cuestión. Pero esto no es suficiente para nosotros, ya que queremos saber las relaciones que tiene con el resto de usuarios y para poder generar nodos *User* necesitaríamos una lista de identificadores. Esto lo hemos logrado haciendo dos llamadas independientes que nos devuelven un objeto *User* del cual solamente hemos extraido su identificador y nombre de usuario para crear su correspondiente nodo *User*.
La llamada que realizamos para obtener la lista de seguidores en la REST API de Twitter corresponde a la llamada *Get followers/list* [12]. Por cada llamada obtenemos 20 seguidores y el límite de veces que podemos realizar esta llamada es de 15 cada 15 minutos. Teniendo en cuenta que un usuario puede tener miles de seguidores, tardaríamos días en conseguir todos los seguidores de todos los *RelevantUser*. Y además, con los recursos de los que disponemos no sería posible procesar tal cantidad de nodos. Para obtener la lista de *following* hemos utilizado la llamada *GET friends/list* [13] que tiene las mismas limitaciones que la anterior llamada.
Para no perder la información sobre las conexiones cono otros usuarios de nuestros RelevantUser en un principio lo que hicimos fue llegar a una solución intermedia. Es decir, en vez de recuperar toda la lista de usuarios lo que hicimos era recuperar los 20 primeros usuarios que es el máximo que nos devuelven las anteriores llamadas. Por lo que por cada *RelevantUser* se obtendrían 20 usuarios de su lista de seguidores y otros 20 de sus seguidos. Cada 15 minutos seríamos capaces de obtener las listas de 15 *RelevantUser*.
Durante la fase 3 tuvimos que cambiar estos valores ya que vimos que nuestro ordenador era incapaz de procesar tal cantidad de nodos. Por lo que para obtener una visualización fluida decidimos obtener solamente 5 usuarios de cada lista.

5.4. Cypher Language

Cypher es el lenguaje que utiliza Neo4J para permitirnos realizar consultas o actualizaciones de nuestros grafos. Es un lenguaje declarativo, es decir basado más en la lógica, pero aun así es fácil de entender y a diferencia de SQL está pensado para que sólo tengamos que definir lo que queramos consultar, insertar, actualizar o eliminar de nuestro grafo sin tener que describir exactamente cómo hacerlo.

En la figura 3 podemos ver el concepto de relaciones entre nodos en Cypher.

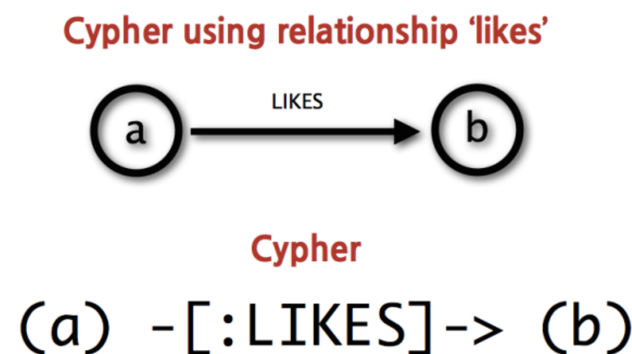


Fig. 3: *Concepto básico de relaciones entre nodos en Cypher[14].*

5.4.1. Instrucciones fundamentales

Hay tres instrucciones básicas para poder realizar operaciones en Neo4J:

- **MATCH:** Es la instrucción más común para encontrar datos.
- **WHERE:** Es una instrucción de condición que nos permite utilizarla para filtrar por patrones.
- **RETURN**: Con esta instrucción indicamos exactamente qué queremos que nos devuelva la consulta. En una consulta siempre tendrá que concluir con un RETURN o con una cláusula de actualización.

5.4.2. Nodos

Para referirnos a los nodos, los encapsulamos con paréntesis (nodo), lo que lo hace aparentar como si fueran un círculo. Si queremos luego referirnos a un nodo con un nombre más corto dentro del paréntesis ponemos el nombre de la variable, dos puntos y a continuación el nombre del nodo. Por ejemplo para crearnos una variable para el nodo *Persona: (p:Persona).*

La estructura más general es:

MATCH(node:Label) RETURN node.property

Hay que tener en cuenta que Cypher es sensible a las mayúsculas.

5.4.3. Relaciones

Para sacar toda la potencia de nuestra base de datos de grafos hay que utilizar las relaciones.
Las relaciones son básicamente una flecha –¿entre dos nodos en la que se puede añadir información adicional.
Esto se puede hacer de varias formas:

- Un tipo de relación como:
  -[:FOLLOWS]-
- Añadiendo un nombre de variable antes de los dos puntos:
  -[rel:FOLLOWS]-
- Añadiendo propiedades a la relación:
  -[since:2016]-

La estructura más general es:

```
MATCH (n1:Node1) – [rel:FOLLOWS] → (n2:Node2)
WHERE rel.property < {value}
RETURN rel
```

5.5. Exportación de los datos del crawler a un fichero CSV

Neo4J nos permite importar ficheros CSV de una manera sencilla por lo que para importar los datos obtenidos por nuestro *crawler* lo que hemos hecho es exportarlos a ficheros CSV. Después se importarán a nuestra base de datos.

Un fichero CSV no deja de ser un texto plano en forma de tabla en la que las columnas son separadas normalmente por comas y las filas por salto de líneas. Un ejemplo sencillo incluyendo cabecera:

```
"idUser","name","location"
"01AB1","Charlie","Cerdanyola"
"10CE2","Julia","Barcelona"
```

Para poder crear estos ficheros de una forma sencilla en Java disponemos de la librería OpenCSV [15]. Tan solo tenemos que añadir una dependencia en nuestro fichero *pom.xml* para añadirla a través de Maven:

```
<dependency>
  <groupId>com.opencsv</groupId>
  <artifactId>opencsv</artifactId>
  <version>3.7</version>
</dependency>
```

Para seguir el correcto formato de un fichero CSV seguimos el formato descrito por RFC 4180[16]. El problema surgía con los nodos *RelevantUser* y *Tweet* ya que muchos campos pueden contener carácteres incompatibles con el formato. Lo que hemos hecho ha sido utilizar el método *ReplaceAll* que contiene la clase *String* y buscar las expresiones regulares para sustituirlas por espacios vacíos.
Otro problema surgió con el nodo *Tweet* y el campo *idMentionedUser*. El método *getUserMentionEntities()* heredado de la interfaz *EntitySupport* nos devuelve una lista de objeto tipo *UserMentionEntity* que contiene el identificador del usuario que se ha mencionado en el *tweet* y su *nick*. Evidentemente no podemos almacenar una lista en el fichero CSV por lo que hemos decidido separar los identificadores de los usuarios mencionados por un guión.
Comprobamos con la herramienta online CSVLint [17] que nuestros ficheros son correctos. Hemos creado cinco ficheros CSV que después se importarán a nuestra base de datos correspondiendo cada uno a un tipo de nodo.

5.6. Creación de la base de datos

5.6.1. Importar ficheros CSV

El primer paso para crear nuestra base de datos es importar los ficheros CSV generados. Hay que tener en cuenta que Neo4J leerá todos los datos como un *String* a no ser se lo indiquemos explicitamente. Esto lo podemos lograr utilizando las funciones *toInt*, *toFloat* y funciones similares.
Para generar el nodo que es uno de los más pequeños hemos tenido que indicarle a Neo4J lo siguiente:

```
LOAD CSV WITH HEADERS
    FROM 'file:///C:/files/csv/relevantUser.csv' AS line
CREATE (:User {Id: line.id, IdUser: line.idUser,
            ScreenName: line.screenName})
```

En la primera línea le estamos indicando que a la hora de importar el fichero CSV tenga en cuenta que la primera fila será una cabecera.

Después en la segunda le indicamos la ruta del fichero y con la instrucción *AS* el nombre de la variable que utilizaremos para referirnos a los datos que están siendo leídos.

En la tercera línea con la instrucción *CREATE* le indicamos a Neo4J que cree un nodo por cada fila leída que se llame User y que contenga las propiedades Id, IdUser y ScreenName. A cada una de las propiedades le indicamos que su valor será el leído en el fichero con la variable line, y especificamos el nombre de la columna de la que vamos a extraer su valor escriendo un punto y el nombre de la cabecera a la que corresponde esa columna.
Para los nodos *Language* y *Place* hemos tenido que modificar la consulta:

```
LOAD CSV WITH HEADERS
FROM 'file:///C:/files/csv/language.csv' AS line
MERGE (: Language {Id: line.id})
```

La única diferencia es que en vez de utilizar el comando *CREATE* que hemos utilizado para el resto de nodos, utilizamos el comando *MERGE* que equivaldría a una mezcla de los comandos *MATCH* y *CREATE*. Buscará un nodo que tenga todos los valores proporcionados por el CSV, si lo encuentra no hará nada, sino actuará igual que el comando *CREATE* y creará otro nodo.
Hemos tenido que recurrir a esta instrucción porque los datos de *Language* y *Place* son únicos, es decir, no necesitamos dos nodos *Language* que equivalgan por ejemplo al idioma inglés.

5.6.2. Creación de las relaciones

Una vez importados todos los datos el siguiente paso es la creación de las relaciones entre los nodos.
Crear una relación es muy sencillo:

```
MATCH (tweet: Tweet)MATCH(lang: Language)
WHERE tweet.IsoLanguageCode = lang.Id
CREATE (tweet) − [: USES_LANGUAGE]−> (lang)
```

Encontramos todos los nodos tipo *Tweet* y *Language* y les asignamos un nombre a la variable para después referirnos a ellos como tweet y lang. Después le indicamos en la condición que para que esos dos nodos puedan tener una relación tienen que cumplir la condición de que el campo *IsoLanguageCode* del nodo *Tweet* tiene el mismo valor que el Id del nodo *Language*. Finalmente creamos una relación entre estos dos nodos diciendo que el nombre de la relación es: *USES LANGUAGE*.
Pero hay un problema con crear las relaciones de esta manera. Tuvimos un problema parecido al que nos ocurrió con los nodos *Language* y *Place* que deben ser únicos. En este caso ocurre lo mismo ya que si había dos *tweets* que referenciaban a un usuario, en vez de haber tres nodos (dos nodos *Tweet* y uno *User*) y sus correspondientes relaciones, lo que ocurría es que se generaba otro nodo más igual al usuario al que hacen referencia (dos nodos *Tweet* y dos nodos *User* con los mismos valores). Para solucionarlo utilizamos el comando *CREATE UNIQUE* para asegurarnos que no pueda haber más de un usuario o *tweet* igual.

5.7. Métodos de visualización

Creada nuestra base de datos lo siguiente que hicimos fue buscar la forma de poder visualizarla en forma de grafo. Realmente Neo4J ya nos proporciona una herramienta para ver nuestros grafos pero es bastante limitada.

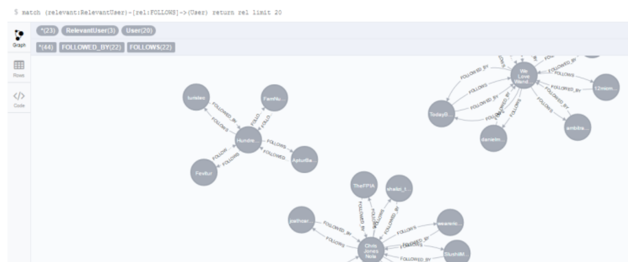


Fig. 4: *Herramienta de visualización de grafos proporcionada por Neo4J.*

Existen herramientas como Gephi que permiten personalizar nuestros grafos con muchas opciones pero como ya hemos comentado anteriormente, nos descantamos por utilizar una librería para lenguaje web.
Hay varias liberías que nos permiten visualizar grafos pero la mayoría requieren que los datos estén en formato JSON.

5.8. Mapeo de datos

Para obtener nuestros datos en ficheros JSON la opción que vimos más fácil es la de mapear nuestra base de datos a objetos POJO para después exportarlos con la librería Jackson [18].
Para traer todos los datos de nuestra base de datos a nuestro proyecto JAVA utilizamos el método *findNodes(Label label)* que nos proporciona la librería de Neo4J. Éste método devuelve todos los nodos del mismo tipo que le indicamos por parámetro. El tipo de nodo se lo indicamos pasándole un objeto *Label* que contiene el nombre del tipo de nodo. Para hacer esto simplemente nos hemos creado una clase por cada tipo de nodo que implementa la interfaz *Label*.
Para conseguir un mapeo directo nos hemos creado un objeto POJO con los atributos del nodo correspondiente. Por cada tipo de nodo hemos creado un POJO y también otro para las relaciones que contiene el identificador del nodo inicio, el identificador del nodo final y el tipo de relación que tienen.
En un principio habíamos creado un fichero JSON por cada tipo de nodo (ver figura 8) pero después vimos que era mucho más complicado unificar todos esos datos con D3.js.

Decidimos unificar todos los nodos en una única lista. Quitando la cabecera de cada tipo de nodo por una única cabecera llamada "nodos" y en la que cada tipo de nodo se diferencia del resto por un atributo con diferentes valores dependiendo del tipo de nodo que sea.

Con el objeto ObjectMapper de la librería Jackson pasando creamos el fichero JSON pero tan solo le podemos pasar una lista y tenemos la de nodos y la de relaciones. Si creamos un JSON por cada lista y después unimos los

```
{
  "users" :[
    {
    "screenName" : "peedink66",
    "idUser" : "373564386"
     },
     {
    "screenName" : "pepito87",
    "idUser" : "373484386"
     }
    ]
}
```

```
{
  "language" :[
    {
    "id" : "en",
     },
     {
    "id" : "it",
     }
    ]
}
```

Fig. 5: *Breve muestra de cómo eran nuestros ficheros JSON en un principio. A la izquierda el fichero correspondiente a los nodos User, a la derecha a los Language.*

```
{
  "nodes" :[
    {
     "group" : "4",
     "id" : "en"
     },
     {
      "group" : "4",
      "id" : "it"
     }
    ]
}
```

Fig. 6: *Breve muestra de la versión final del fichero JSON.*

Strings, no estaríamos respetando el formato correcto ya que el último corchete del primer *String*, ya indicaría que que es el final del fichero.
Lo que hicimos fue crear un método que recoge los datos de ambos ficheros para crearlo de forma “manual “ y después utilizando el objeto ObjectMapper para que aplique las correspondientes sangrías y sea visualmente más fácil de leer.

También tuvimos que crear un fichero JS asignándole a una variable todos los valores de la lista de nodos. Esto es porque D3.js al final no deja de ser una librería de JavaScript, por lo que subir un fichero local aunque no es imposible, es muy complicado. Por lo que la mejor opción es crear un fichero JS que podemos cargar fácilmente con una variable que contenga todos los nodos.

5.9. D3.JS

D3.js es una librería de JavaScript para manipular documentos basados en datos. Nos permite, a partir de nuestros datos, crear gráficos vivos en los que es posible interactuar con ellos.

Para poder utilizarlo solamente tenemos que cargar el script referenciando su librería.
Para desarrollar nuestro html nos hemos basado en un desarrollo que encontramos en el apartado de ejemplos en la página oficial de D3.js [19].
Para cargar nuestro fichero JSON utilizamos el método *d3.json()* donde indicamos su ruta.
Utilizando el campo “group” que tenemos en nuestro fichero JSON para diferenciar los tipos de nodos, hacemos que muestre cada nodo de un color distinto dependiendo del valor de ese campo. También, cuando hacemos click en algún nodo, hacemos que se abra un fichero html pasándo por el método GET del protocolo HTTP el identificador del nodo y el grupo al que pertenece.
En el html que abrimos al clickar un nodo, tenemos una tabla con todos los campos de los nodos. En un principio son invisibles y sin valor. Después de haber cargado la página llamamos a una función de otro fichero JS.
En esta función hacemos que según el grupo que recibamos por el método GET, haga visibles algunos campos de la tabla. Con el identificador del nodo que también recibimos, buscamos ese nodo en el fichero JS que durante la exportación de la base de datos, y asignamos estos valores a los campos correspondientes de la tabla.
Tuvimos problemas a la hora de pasar los valores a la tabla porque intentábamos asignar valores a una tabla que todavía no existía. Para solucionar este problema indicamos expresamente que se ejecute nuestro script cuando la página haya terminado de cargarse:

```
<script> window.onload = showInfo; </script>
```

En el anexo A.3 y A.4 se puede ver el visualización del grafo mediante D3.JS y una tabla con la información de un nodo.

6 Proyecto Java

Nuestro proyecto JAVA se compone de un total de 16 clases. De las 16, 5 son clases que están en el paquete Label que simplemente implementan la interfaz *Label* para cada tipo de nodo.

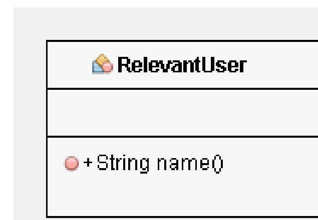


Fig. 7: *Diagrama de clase de RelevantUser.*

Después tenemos 6 clases entidad que utilizamos para mapear los datos de la base de datos. Hay una para cada nodo y otra para las relaciones.
Todas las clases tienen un identificador único. Y aunque puede ser redundante, porque ya contamos con los identificadores que extraemos de los datos de Twitter, hemos asignado a cada nodo un identificador único porque a la hora de realizar la visualización resulta mucho más cómodo trabajar con un único identificador que con varios (idPlace, idTweet, etc.).
Tenemos otra clase llamada TwitCrawNeo que es la que contiene el main de la aplicación. En esta clase inicializamos todos los variables que necesitamos para conectarnos a twitter a través de la librería Twitter4J. Llama a la clase TweetReader para recopilar los tweets, y a la clase

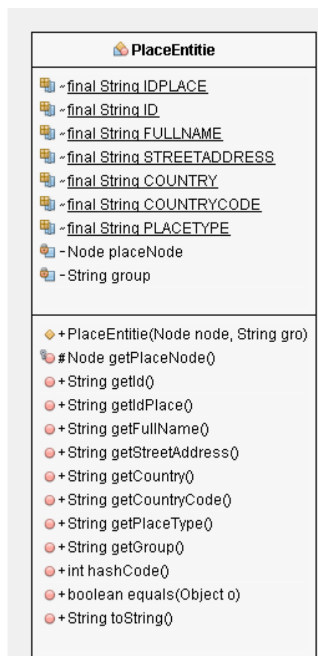


Fig. 8: *Diagrama de clase de PlaceEntitie.*

Neo4jBD para almacenar en la base de datos todos los datos recopilados y después exportarlos en ficheros.

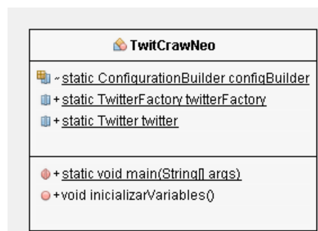


Fig. 9: *Diagrama de clase de TwitCrawNeo.*

La clase TweetReader es el crawler que hemos creado para recopilar todos los datos de Twitter. Se compone de un método principal llamado *searchTweets()* que se encarga de buscar y escribir en ficheros CSV todos los datos encontrados según el hashtag que pongamos en la consulta. Este método llama a otros dos hilos llamados *searchFollowingThread()* y *searchFollowersThread()* que se encargan de encontrar parte de los nodos *User*. Cuando exceden el número de llamadas permitido por la REST API de Twitter, se detienen durante 15 minutos hasta que se resetea el límite.

La clase Neo4jBD se compone de distintos métodos para crear nuestra base de datos. El método *initiateDb()* configura los parámetros de nuestra base de datos. Con el método *clearDb* lanzamos una consulta para eliminar todos los nodos y relaciones, para después con el método *createDb()* importar los ficheros CSV y crear las relaciones entre nodos. En el método *exportData()* generamos tanto el fichero JSON como el JS.

Finalmente tenemos la clase *Utils* que dispone de cuatro métodos. Dos métodos son para ayudar a generar los ficheros JSON y JS. Otro para generar un identificador único para cada nodo y finalmente otro que consultamos durante la recolección de datos para comprobar si hemos alcanzado el límite de llamadas que nos permite Twitter.

Fig. 10: *Diagrama de clase de TweetReader.*

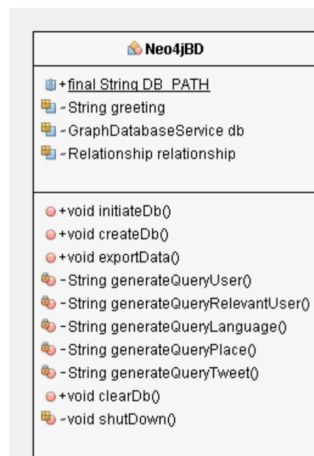


Fig. 11: *Diagrama de clase de Neo4jBD.*

7 RESULTADOS

Como resultado del proyecto tenemos un sistema capaz de recolectar *tweets* según el *hashtag* que pongamos como parámetro de búsqueda, y almacenar gran cantidad de información relacionada con éste. Unas pautas para la creación de una base de datos no relacional con Neo4J, y la forma de visualizar esta información mediante una tecnología puntera como lo es D3.js.
Hemos cumplido todos los objetivos que nos habíamos propuesto. La idea inicial era utilizar como método de visualización Gephi si estábamos con falta de tiempo, y sino D3.js. Hemos conseguido visualizar el grafo con D3.js, y además hemos añadido la funcionalidad de poder ver la información de cada nodo, por lo que hemos cumplido con los objetivos de visualización.
Aun así no acabamos de estar satisfechos con esta última fase, y creemos que se podría haber conseguido un mejor resultado si hubiesemos hecho una planificación mejor. Ahora resulta evidente que nos excedimos con el tiempo asignado en la primera fase, y tendríamos que haberlo repartido mejor con la tercera fase que es la que ha resultado ser la de mayor dificultad.
También es necesario reconocer que la falta de inversión de tiempo en las etapas de investigación y evaluación de los diferentes métodos de visualización, nos causó una gran

pérdida tiempo en intentos de implementar tecnologías que no eran las adecuadas para nuestros requisitos y que después tuvimos que descartar. Si hubiésemos dedicado más tiempo a estas dos etapas nos habríamos ahorrado un tiempo de desarrollo innecesario.

8 Conclusiones

Después de varios meses de desarrollo de este proyecto creemos que este proyecto nos ha enriquecido como profesionales. Estamos completamente seguros de que cada vez más empresas empezarán a utilizar bases de datos no relacionales como Neo4J.
Los métodos de visualización de datos como D3.js son cada vez más populares para comunicar mejor el significado de los datos. Y en un mundo en el que todos los días estamos rodeados de tal cantidad de datos de la que somos incapaces de procesar, las herramientas de visualización de datos serán indispensables para abstraernos de los datos innecesarios.ºº Como mejoras en nuestro proyecto nos centraríamos principalmente en la última fase. Son muchas las posibilidades que nos ofrece D3.js. Se podría por ejemplo, utilizar un mapa del mundo y aprovechar que tenemos almacenados los lugares de los tweets para localizar cada nodo en el mapa. También se podría añadir a cada nodo información sobre el tiempo ya que disponemos de su zona de horario.

Agradecimientos

Me gustaría agradecer a mi tutor Jordi Casas-Roma por los libros y artículos recomendados que me fueron de gran utilidad para las dos primeras fases.

Referencias

[1] Fernando Rivero (2015), Informe ditrendia: Mobile en España y en el Mundo 2015

[2] Margaret Rouse (2013), 3Vs (volume, variety and velocity) , http://whatis.techtarget.com/definition/3Vs

[3] Adriana Martin, Susana Chavez, Nelson R. Rodríguez, Adriana Valenzuela, Maria A. Murazzo, Bases de Datos NoSql en Cloud Computing, En: XV Workshop de Investigadores en Ciencias de la Computación, Abril 2013, p. 166-170

[4] Tweepy Documentation, http://docs.tweepy.org/en/v3.2.0/

[5] The Open Graph Viz Platform, https://gephi.org/

[6] Neo4J Graph DatabseSupport, https://marketplace.gephi.org/plugin/neo4j-graph-database-support/

[7] Yusuke Yamamoto (2007), Twitter4J, an unofficial Java library for the Twitter API, http://twitter4j.org/en/index.html

[8] Creating a Maven Swing Application Using Hibernate - NetBeans IDE Tutorial, https://netbeans.org/kb/docs/java/maven-hib-java-se.html

[9] Twitter Apps, https://apps.twitter.com

[10] Twitter Public Api The Search API, https://dev.twitter.com/rest/public/search

[11] Yusuke Yammamoto, Interface Status, http://twitter4j.org/javadoc/twitter4j/Status.html

[12] Get Followers/list, https://dev.twitter.com/rest/reference/get/follower

[13] Get Friends/list, https://dev.twitter.com/rest/reference/get/friends/list

[14] Intro To Cypher, http://neo4j.com/developer/cypher-query-language/

[15] OpenCsv, http://opencsv.sourceforge.net/

[16] The Internet Society (2005), Common Format and MIME Type for Comma-Separated Values (CSV) Files, https://tools.ietf.org/html/rfc4180

[17] CSV Lint, Check your CSV files with CSVLint, http://csvlint.io/

[18] Core Part Of Jackson, https://github.com/FasterXML/jackson

[19] https://github.com/d3/d3/wiki

Apéndice

A.1. Diagrama de clases nuestra BD

A.2. Diagrama de clase de TweetReader.

A.3. Grafo con D3.JS.

A.4. Tabla de un nodo Relevant User.

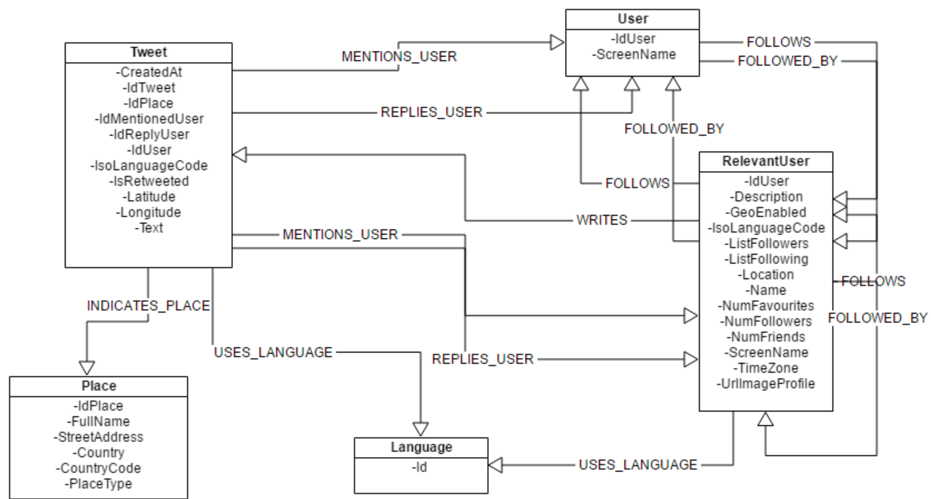


Fig. 12: *A.1. Diagrama de clases nuestra BD*

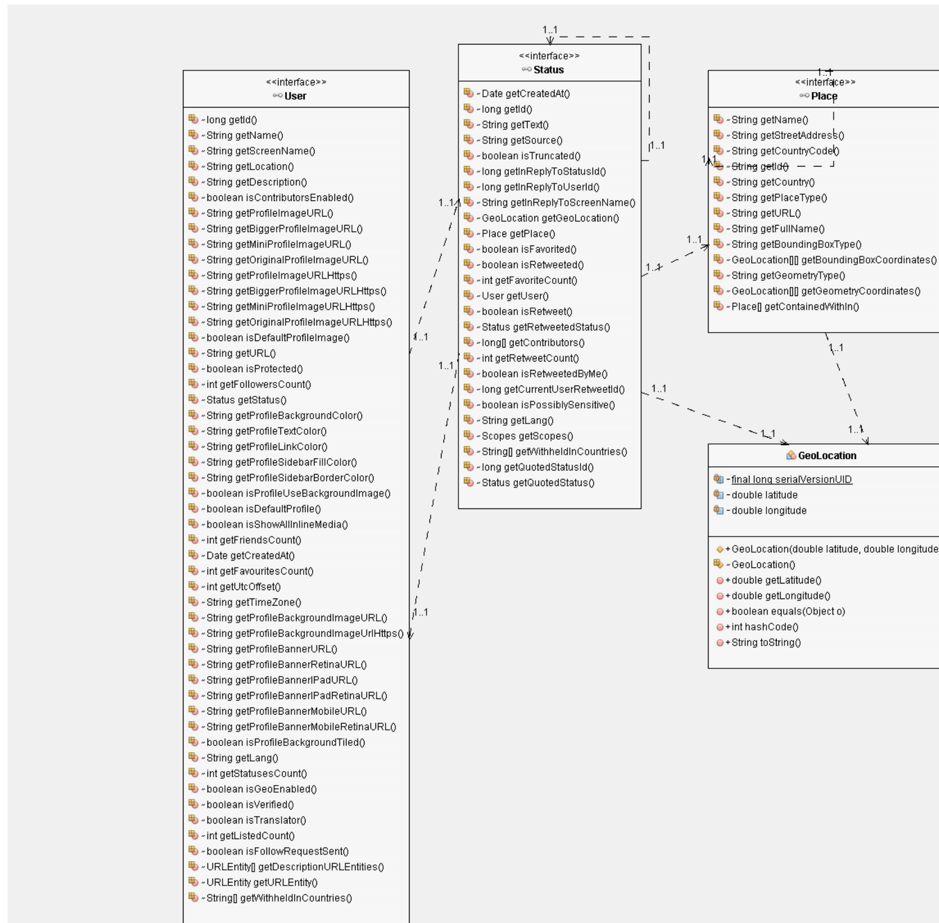


Fig. 13: *A.2. Diagrama de clase de TweetReader.*

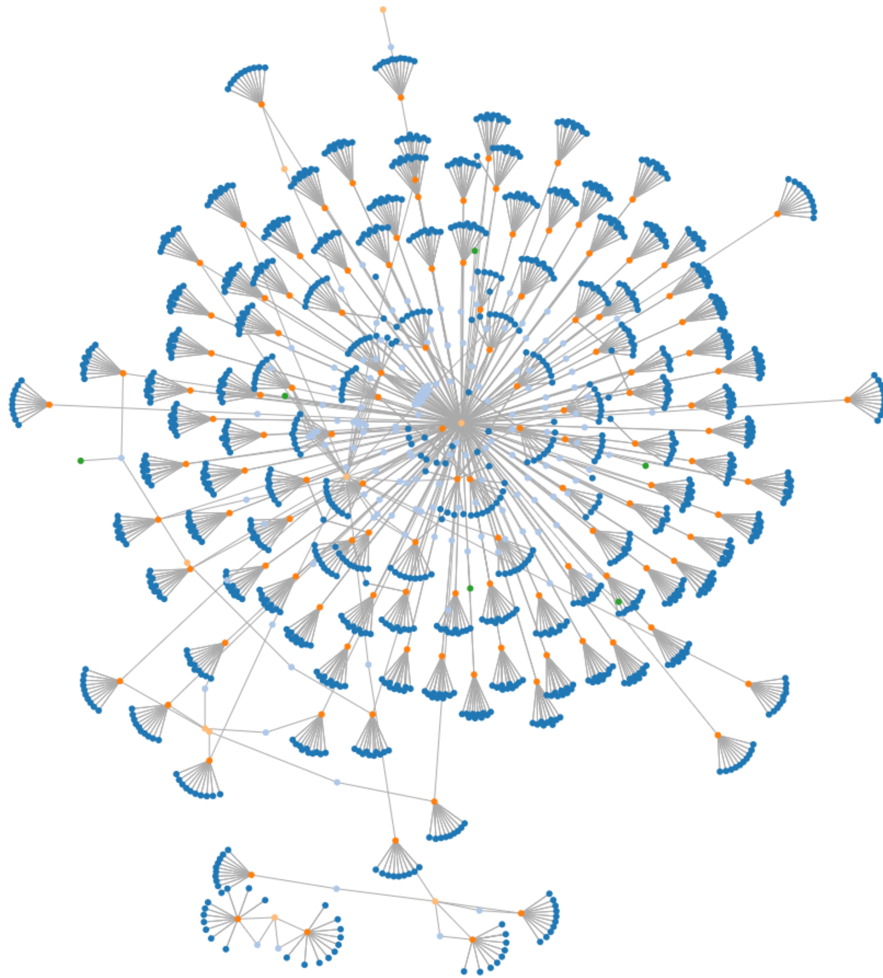
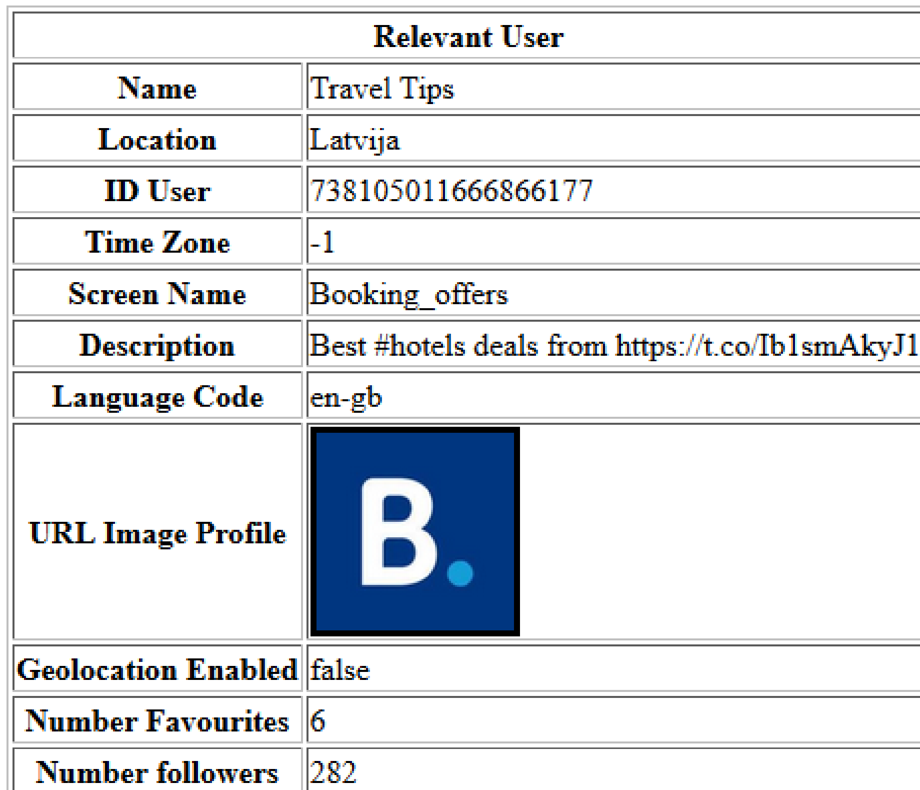


Fig. 14: *A.3. Grafo con D3.JS.*



Relevant User	
Name	Travel Tips
Location	Latvija
ID User	738105011666866177
Time Zone	-1
Screen Name	Booking_offers
Description	Best #hotels deals from https://t.co/Ib1smAkyJ1
Language Code	en-gb
URL Image Profile	
Geolocation Enabled	false
Number Favourites	6
Number followers	282

Fig. 15: *A.4. Tabla de un nodo Relevant User.*