Diseño de una base de datos para analizar la actividad de usuarios en Twitter

Cristina Pérez Solà

PID_00209828

Universitat Oberta de Catalunya

Índice

Introducción

A medio camino entre una red social y un servicio de *microblogging*, Twitter es una plataforma que permite el intercambio de pequeños mensajes de texto, conocidos como tuits, entre los usuarios que la conforman. Fundado en el 2006, el servicio creció rápidamente hasta alcanzar la cifra de 500 millones de usuarios en el 2012. Paralelamente a la popularización de Twitter, las redes sociales en línea (y Twitter entre ellas) desempeñaron un papel crucial en la Primavera Árabe, pasando a formar parte de la base de las herramientas comunicativas usadas para coordinar grandes masas de ciudadanos: protestas, movimientos sociales y revoluciones fueron organizadas usando redes sociales como medio de comunicación.

Con el incremento del uso de las redes sociales en línea, grandes cantidades de información sobre individuos y sobre las relaciones que estos compartían empezaron a volcarse en la web y con ello surgía, por un lado, la necesidad de diseñar sistemas de almacenamiento capaces de guardar eficientemente estos datos y, por otro lado, la oportunidad de explotar estos datos con múltiples objetivos.

En este sentido tomó especial importancia el análisis de redes sociales, una rama que comparten varias disciplinas académicas y que se centra en utilizar teoría de redes para analizar datos de redes sociales. Entre las muchas aplicaciones que tiene el análisis de redes sociales encontramos la minería de datos, el análisis del comportamiento de los usuarios, los sistemas de recomendación o el estudio de la propagación de enfermedades.

Sin embargo, las redes sociales en línea no eran el único tipo de servicio interesado en almacenar y procesar este tipo de datos, basado en las conexiones y representable en forma de grafo, a gran escala. Empresas de gran protagonismo en la economía a nivel mundial (como Google o Amazon) también basaban sus negocios en datos que son esencialmente grafos, y la idea de que la representación de los datos en forma de grafo permitía tanto tratar con grandes cantidades de datos, como procesar estos datos para obtener información antes impensable, iba tomando peso. En este contexto, el paradigma de las bases de datos orientadas a grafos resurgió de su letargo.

El caso práctico que se tratará en este documento está basado en un escenario real. En concreto, se centrará en el diseño de una base de datos para analizar la actividad de un subconjunto de usuarios en Twitter. A partir de la descripción del funcionamiento de Twitter, que permitirá obtener los requisitos de los datos que serán almacenados, se diseñará y construirá una base de datos. Como

veremos, el tipo de datos que se van a procesar son apropiados para una representación en forma de grafo, por lo que se procederá al diseño de una base de datos orientada a grafos adecuada a las necesidades de datos descritas.

Objetivos

En los materiales que forman parte de este caso práctico el estudiante encontrará las herramientas y los contenidos necesarios para lograr los siguientes objetivos:

1. Realizar el proceso de diseño de una base de datos orientada a grafos, desde la descripción de los datos hasta el diseño físico, pasando por la creación del modelo conceptual de datos y el diseño lógico.
2. Adquirir un conocimiento genérico sobre bases de datos orientadas a grafos, sus propiedades y las principales ventajas que suponen sobre una base de datos relacional.
3. Identificar las características de un conjunto de datos que lo hacen idóneo para ser almacenado en una base de datos orientada a grafos.
4. Conocer neo4j, un sistema gestor de bases de datos orientada a grafos con capacidad de procesamiento nativo.
5. Conocer Cypher, un lenguaje declarativo que permite realizar consultas sobre neo4j.
6. Tener una visión global del tipo de información que se puede extraer del análisis de los datos de una red social en línea.

1. Descripción del dominio

En este apartado describiremos el dominio sobre el que trabajaremos durante este caso práctico. En primer lugar, presentaremos el contexto en el que se enmarcará el diseño de la base de datos. Después, describiremos los datos que se almacenarán en dicha base de datos. Para acabar, discutiremos qué información será interesante extraer, así como los requisitos no funcionales de la base de datos.

1.1. Contexto

TwitterGaggle es un cliente de Twitter para dispositivos móviles. Como tal, TwitterGaggle es una pequeña aplicación que se puede instalar en cualquier teléfono inteligente y que permite al propietario del teléfono interactuar con la plataforma Twitter, realizando las mismas funciones que podría hacer a través de la página web, pero con una interfaz especialmente diseñada para el dispositivo móvil.

Como muchas otras aplicaciones para dispositivos móviles, TwitterGaggle se ofrece en dos versiones plenamente funcionales, una de ellas gratuita y la otra *premium*, es decir, de pago. La versión gratuita es plenamente funcional, pero incluye un pequeño *banner* donde se visualiza publicidad. La versión *premium* requiere una suscripción mensual, pero se encuentra libre de cualquier tipo de publicidad.

Un año después de lanzar al mercado la primera versión de TwitterGaggle, GaggleCo, la empresa propietaria de la aplicación, hace un estudio sobre el impacto que han tenido las dos versiones de su aplicación en el mercado. El estudio concluye que la gran mayoría de sus usuarios utilizan la versión gratuita y que esto no está generando suficientes beneficios a la empresa. Por este motivo, GaggleCo decide que el objetivo para el próximo año es aumentar los ingresos por publicidad que le genera esta versión de la aplicación.

Para ello, GaggleCo determina que será necesario:

1) Aumentar el número de usuarios de la aplicación.

2) Incrementar los ingresos por publicidad que se obtienen por cada usuario.

Para lograr este objetivo, GaggleCo decide realizar un estudio sobre el comportamiento que los usuarios exhiben en Twitter. GaggleCo confía en que podrá

aprovechar la información extraída de este estudio para lograr sus objetivos para este año, por ejemplo, analizando el flujo de información en la red para promocionar el uso de la aplicación e incrementando los ingresos por usuario con la realización de segmentación de comportamiento y publicidad dirigida.

El objetivo de este caso práctico es diseñar una base de datos que nos permita almacenar datos sobre las actividades de un conjunto de usuarios de Twitter en esta red, enfatizando la selección de la tecnología y el SGBD que nos permita realizar los análisis deseados de la manera más eficiente posible y simplificando el proceso de extracción de información.

SGBD

El acrónimo SGBD se utiliza para denotar los sistemas gestores de bases de datos.

1.2. Los datos de la red Twitter

Twitter es una red social creada a principios del 2006 que proporciona una plataforma para intercambiar pequeños mensajes de texto. Estos pequeños mensajes de texto son conocidos como tuits (*tweets* en inglés), y su característica principal es la limitación que la propia red impone sobre su longitud, fijada en 140 caracteres como máximo.

El contenido de los tuits es muy diverso. Originalmente los tuits se consideraban como mensajes de actualización de estado, que permitían describir qué se estaba haciendo en un momento dado. Sin embargo, el uso actual que se le da a la red sobrepasa con creces la idea original para la que fueron concebidos: los tuits se utilizan para comunicar cualquier tipo de información, desde trivialidades de la vida cotidiana hasta mensajes para organizar movilizaciones reivindicativas. La información que se comparte en Twitter puede ser muy valiosa. Por este motivo, GaggleCo ha decido recogerla y analizarla.

Traducción de *tweet*

La traducción literal de *tweet* hace referencia a la acción de emitir un sonido por cierto tipo de aves.

Los siguientes subpartados describen los datos que GaggleCo ha decidido capturar sobre la red. En concreto, veremos cómo GaggleCo está interesada en recoger datos sobre los usuarios de Twitter, sobre las relaciones explícitas que estos usuarios establecen y sobre sus comunicaciones a través de la red.

Evolución de Twitter

Originalmente Twitter mostraba la pregunta "`¿Qué estás haciendo?`" delante de la caja de texto que se utiliza para crear un nuevo tuit. La frase fue cambiada por un "`¿Qué está pasando?`" en el año 2009, reflejo de los cambios que los propios usuarios estaban ejerciendo sobre la funcionalidad de la red.

1.2.1. Los usuarios de Twitter

Para participar activamente en Twitter es necesario crear un usuario. Todos los usuarios de Twitter tienen un identificador único, que es un valor numérico que Twitter utiliza para hacer referencia al usuario dentro de la red. Aunque este identificador resulta muy práctico para seleccionar un usuario concreto de forma inequívoca, es difícil de recordar para las personas, por lo que Twitter permite definir tanto un nombre de usuario, que se utilizará para iniciar sesión

en la red, como un nombre de perfil, que se mostrará en el perfil y que podrá ser cambiado con el tiempo, si el usuario así lo desea.

Twitter es usado tanto por empresas, medios de comunicación tradicionales u organizaciones sin ánimo de lucro, como por celebridades o ciudadanos anónimos. Especialmente pensando en empresas y en celebridades, Twitter ofrece un servicio de verificación de identidad, que permite asegurar que una cuenta de Twitter concreta pertenece a quien dice ser ese usuario. Así pues, algunos de los perfiles de Twitter se muestran como verificados.

La red dispone también de una opción de geolocalización que permite incluir datos sobre la posición del usuario cuando este interactúa con Twitter. Estos datos suelen estar disponibles cuando el usuario accede a la red mediante un dispositivo móvil que incorpora GPS. Los usuarios pueden declarar sus preferencias en relación con la geolocalización en su perfil, indicando si desean mantenerla activada o desactivada. Así pues, cuando la geolocalización está desactivada las interacciones del usuario con la red no contendrán datos de posicionamiento aunque estos estén disponibles. Del mismo modo, cuando la geolocalización está activada, las interacciones contendrán datos de posicionamiento siempre y cuando estos estén disponibles.

Twitter está disponible en múltiples idiomas. Los usuarios pueden configurar con qué idioma quieren interactuar con Twitter, seleccionando su preferencia de entre una lista de idiomas predefinida por Twitter. El idioma elegido entra a formar parte del conjunto de datos que forman el perfil del usuario en la red.

Además de estos datos, los usuarios de Twitter pueden incluir más datos en su perfil, por ejemplo, su localización actual y un enlace a su página web personal. De todos modos, no existe ninguna restricción ni validación sobre estos datos, de manera que un usuario puede fijar cualquier valor para la localización que aparece en su perfil, siendo habitual que los usuarios incluyan sitios ficticios o pequeñas bromas en este espacio.

Asimismo, la propia red incorpora datos adicionales en los perfiles de los usuarios; la fecha en la que fueron creados, el huso horario en el que se encuentra el usuario (y la diferencia entre ese huso horario y el tiempo universal coordinado) o el número de mensajes que ha enviado desde que se unió a la red son algunos de los atributos adicionales que se pueden encontrar en los perfiles de los usuarios de Twitter.

1.2.2. Las relaciones explícitas entre los usuarios

Una de las características principales de la redes sociales es que permiten a sus usuarios crear relaciones explícitas dentro de la propia red. Twitter implemen-

ta estas relaciones explícitas a través de las relaciones de seguimiento entre usuarios. Así pues, los usuarios de Twitter pueden seguir a otros usuarios de la red.

El hecho de seguir a un usuario de Twitter implica, además de hacer explícita la relación, que se reciban todas las actualizaciones de estado de este usuario en la página principal (que en Twitter se conoce como *timeline*). De este modo, por ejemplo, si Alice está siguiendo a Bob, Alice recibe en su *timeline* todas las actualizaciones de estado que Bob envía. Un usuario puede seguir a tantos usuarios como desee, ya que Twitter no impone restricciones en este sentido. Así, el *timeline* del usuario Alice estará formado por los tuits que envían los usuarios a los que Alice sigue.

Las relaciones entre usuarios en Twitter no son bidireccionales. Por lo tanto, el hecho que Alice esté siguiendo a Bob no implica necesariamente que Bob esté siguiendo a Alice. Las dos relaciones, es decir, que Alice siga a Bob y que Bob siga a Alice, se tratan de manera independiente en la red. Esto es un punto diferencial entre Twitter y otras redes sociales como Facebook, que exigen bidireccionalidad en las relaciones de seguimiento.

1.2.3. Los mensajes o tuits

Los tuits son las unidades básicas de comunicación entre los usuarios de Twitter. Son pequeños mensajes de texto, limitados a 140 caracteres, que permiten a los usuarios expresar su estado actual, comunicar noticias o mantener pequeñas conversaciones. Del mismo modo que los usuarios, todos los tuits tienen asociado un identificador único dentro de la red.

Además del identificador y del texto que comunican, todos los tuits contienen información sobre el momento en el que fueron creados (el día y la hora), así como datos del cliente usado para crearlos, es decir, la aplicación utilizada por el usuario para crear el tuit. No existe un formato concreto para representar el identificador de la aplicación que ha creado un tuit. La mayoría de las aplicaciones incluyen un código html enlazando a su página web y el nombre de la aplicación para identificarse, pero existen otras que no incluyen ningún tipo de información. Los tuits creados a través de la interfaz web de Twitter contienen simplemente la cadena `web` como identificador de aplicación de origen.

Si el usuario que crea el tuit tiene la geolocalización activada y, además, en el momento de crearlo la información sobre su posición actual está disponible, esta se incluye en el tuit. A diferencia de la información de localización que se incluye en el perfil de un usuario, los datos de geolocalización de los tuits tienen un formato predefinido. La información de geolocalización de un tuit

consiste en las coordenadas geográficas de este (longitud y latitud) y, opcionalmente, el punto de interés al que pertenecen estas coordenadas.

Twitter permite dar de alta puntos de interés en el sistema. Estos puntos de interés, que también tienen un identificador único en la red, tienen un nombre (completo) que permite describirlos, así como las coordenadas geográficas que delimitan su localización. Dado que las localizaciones pueden ser espacios de tamaño arbitrario, su posición se aproxima definiendo un rectángulo en el espacio que engloban. Este rectángulo se representa especificando las coordenadas geográficas de sus cuatro vértices. Las localizaciones también suelen tener un nombre corto asociado que permite designarlas usando menos caracteres; un atributo, que indica el tipo de localización usado para describir, por ejemplo, si se trata de una ciudad, un barrio, o una sede, y, finalmente, un código de país, que señala el país al cual pertenecen las coordenadas. Así pues, algunos tuits geolocalizados contendrán únicamente las coordenadas geográficas desde donde se han creado, mientras que otros tendrán, adicionalmente, un punto de interés asociado.

Como hemos visto, cuando un usuario crea un tuit, este aparece en los *timelines* de todos aquellos usuarios que lo siguen. De este modo, podemos considerar los tuits como mensajes de *broadcast*, que no van dirigidos a nadie en concreto. Este formato es ideal para enviar notificaciones a los seguidores, pero no es muy adecuado para mantener conversaciones. En ocasiones, un usuario leerá un tuit y deseará responder o comentar el contenido de ese tuit en concreto. Para estos casos, Twitter permite crear un tipo de tuit especial llamado respuesta (o *reply*). Así pues, un mensaje de *reply* es un tuit que responde a otro tuit anteriormente creado en la red.

Replies

El funcionamiento de los *replies* explicado en el caso práctico es una simplificación del funcionamiento real de estos en Twitter. Twitter permite hacer un uso más general de las respuestas, permitiendo enviar *replies* no solo a tuits existentes sino también a usuarios. De este modo, los *replies* se convierten también en mensajes dirigidos a usuarios concretos.

Otra situación habitual en Twitter se produce cuando un usuario lee un tuit y decide que quiere compartirlo con sus seguidores. En vez de copiar el contenido del tuit y enviarlo como si fuera propio, Twitter permite reenviar el tuit original. Así, un retuit (*retweet* en inglés) es un tipo de tuit que permite reenviar un tuit anterior.

Un tuit nunca puede ser un retuit y un *reply* a la vez. Si un usuario hace un retuit de un mensaje que era un *reply* a otro mensaje anterior, el primero se considera que es un retuit. Siguiendo la misma línea de razonamiento, si un usuario hace un *reply* a un tuit que era un retuit de otro mensaje anterior, el primero se considera que es un *reply*. Así pues, un retuit de un *reply* es un retuit (y no un *reply*) y un *reply* de un retuit es un *reply* (y no un retuit).

Nótese que un usuario puede hacer tanto retuits como *replies* de cualquier tuit, independientemente de que este haya sido creado por un usuario al que sigue o no. Es decir, el hecho de seguir o no a un usuario no influye en la posibilidad de poder responder o reenviar sus mensajes.

1.3. Delimitación de los datos recogidos

Recopilar los datos relativos a los usuarios de Twitter, sus relaciones de seguimiento y sus mensajes para todos los usuarios de Twitter es un proceso que requiere una capacidad computacional elevada (con su coste económico asociado). Este coste supera con creces el presupuesto que GaggleCo tiene para el estudio. Por este motivo, GaggleCo identifica un subgrupo de usuarios de Twitter sobre los que centrará el proyecto. Estos usuarios, de especial relevancia para el análisis, serán el núcleo de usuarios sobre el que se recogerán los datos de la red.

En concreto, GaggleCo ha seleccionado 500 usuarios que considera relevantes para el estudio. De estos usuarios, se recogen todos los datos del perfil disponibles.

Además, se identifican también todos los usuarios que son seguidos por estos usuarios relevantes, es decir, todas la relaciones explícitas de seguimiento que los usuarios relevantes expresan. De todo el conjunto de usuarios no relevantes que son seguidos por usuarios relevantes, se recoge únicamente el identificador de usuario (y no todos los datos que forman el perfil), ya que estos usuarios no son interesantes por sí mismos, sino por las relaciones que muestran con el subconjunto de usuarios relevantes.

De manera análoga, se recogen todos los tuits vinculados a los usuarios relevantes. El conjunto de tuits vinculados a un usuario concreto está formado por:

- Los tuits creados por el usuario en cuestión, ya sean actualizaciones de estado, respuestas a otros tuits o reenvíos de tuits de terceros.

- Los tuits que otros usuarios de la red envían al usuario, es decir, los *replies* dirigidos al usuario de estudio.

- Los tuits que reenvían un tuit del usuario de estudio, es decir, los retuits de tuits escritos por el usuario.

Nótese que los dos últimos tipos de tuits pueden haber sido enviados por usuarios que no pertenecen al grupo de usuarios relevantes. En este caso, de estos usuarios únicamente se conocerá el identificador de usuario, pero no se recogerán todos los atributos del perfil (nombre de usuario, localización, url, etc.), ya que no resultan interesantes para el estudio, más allá de su conexión con los usuarios relevantes.

Los datos recogidos también estarán delimitados en el tiempo. Es decir, GaggleCo recogerá los tuits vinculados a los usuarios relevantes durante un periodo de tiempo fijado y, por lo tanto, no se dispondrá de datos sobre la actividad

anterior de estos usuarios, ni tampoco de la que tendrán en el futuro, después de haber finalizado el tiempo de recogida de datos. GaggleCo ha decidido que, para este primer estudio, recogerá datos de la actividad en Twitter de los usuarios relevantes durante un periodo de 40 días.

1.4. Explotación de la base de datos

GaggleCo ha decidido recoger datos sobre el comportamiento de algunos usuarios en Twitter con un objetivo muy concreto: aumentar los ingresos que TwitterGaggle le reporta. En este subapartado, analizaremos algunas ideas de explotación de los datos recogidos que pueden ayudar a lograr el objetivo propuesto, es decir, comentaremos qué tipo de información se puede obtener a partir de estos datos que pueda ser interesante para este propósito. De todos modos, hay que tener en cuenta que las preguntas que mencionamos son solo algunos ejemplos del tipo de información que puede ser interesante extraer, pero en ningún caso pretenden ser una lista exhaustiva.

1.4.1. La actividad en la red

En primer lugar, los datos recogidos pueden servir para responder a un conjunto de cuestiones muy genéricas, que permiten obtener una visión general del nivel de actividad de los usuarios relevantes para el estudio y del uso que se le da a la red. Así, será interesante responder a preguntas como:

- ¿Cuántos tuits envían de media los usuarios?
- ¿Qué proporción de los tuits enviados son retuits? ¿Y *replies*? ¿Qué proporción de tuits contiene información de geolocalización? ¿Cuántos de estos últimos están asociados a una localización de Twitter?
- ¿Qué días de la semana y a qué horas se produce el mayor nivel de actividad en la red?
- ¿Existen lugares especialmente utilizados para enviar tuits? ¿Cuáles son estos lugares?

1.4.2. Las aplicaciones que interactúan con Twitter

Antes de intentar aumentar el número de usuarios de TwitterGaggle, será importante analizar cuál es la situación actual de TwitterGaggle en el mercado de clientes de Twitter. En este sentido, se intentará responder a preguntas como:

- ¿Cuántos usuarios utilizan aplicaciones para interactuar con Twitter en vez de acceder a través de la página web?

- ¿Cuál es el cliente de Twitter más usado?
- ¿Cuál es el ranquin de herramientas usadas? ¿En qué lugar se encuentra TwitterGaggle en dicho ranquin?
- ¿Cuál es el idioma más usado por los usuarios que utilizan clientes externos?

1.4.3. Detección de usuarios importantes

Una vez se tenga una visión general de los datos recogidos y se haya analizado la situación actual de TwitterGaggle, se estará en posición de empezar a realizar consultas enfocadas al aumento de usuarios de TwitterGaggle. En este sentido, es interesante detectar (de entre el conjunto de los usuarios relevantes para el estudio) usuarios clave que permitan focalizar las campañas publicitarias minimizando el coste, a la vez que se maximiza la efectividad de estas.

Existe una extensa literatura de teoría de redes sociales centrada en analizar qué se considera un usuario importante. Las medidas de centralidad son métricas que permiten determinar la importancia de un nodo de un grafo siguiendo un criterio concreto. Las tres métricas básicas de centralidad son centralidad de grado, de intermediación y de proximidad.

Véase también

El apartado 4 contiene una breve descripción de los conceptos básicos asociados a los grafos.

La **centralidad de grado** es la métrica más simple y corresponde directamente con el grado del nodo. Aplicado sobre una red social, la centralidad de grado nos informa del número de conexiones directas de un nodo. La **centralidad de intermediación** tiene en cuenta cuántas veces un nodo concreto se encuentra en el camino más corto entre cada par de nodos del grafo. En una red social, nos indica el poder que tiene el usuario para ocultar o distorsionar la información que fluye a través de la red. La **centralidad de proximidad** tiene en cuenta cómo de cerca se encuentra un nodo del resto de los nodos del grafo. En redes sociales, un nodo será central siguiendo este último enfoque en la medida en que puede evitar el potencial de otros nodos para controlar la comunicación. Así, por ejemplo, un nodo no será central en relación con la proximidad cuando necesite transmitir mensajes a través de otros, es decir, cuando dependa de otros usuarios de la red para comunicarse.

Así pues, en relación con la detección de usuarios importantes, GaggleCo estará interesada en responder a preguntas como las siguientes:

- ¿Cuáles son los usuarios con más seguidores? ¿Y con más seguidores de segundo y tercer nivel?

- ¿Cuáles son los usuarios más bien conectados, con relación a la proximidad? ¿Y con relación a la intermediación?

- ¿Cuáles son los usuarios más activos en la red?

1.4.4. Segmentación de comportamiento

Además de aumentar el número de usuarios, otra vía para incrementar los beneficios de la versión gratuita supone aumentar los ingresos por publicidad que cada usuario genera.

Se conoce como segmentación de comportamiento (o *behavioral targeting*) al proceso consistente en clasificar a los visitantes de una web a partir de los patrones de comportamiento que exhiben. Esta información se aprovecha para incrementar la eficacia de las campañas publicitarias.

En el caso que estamos tratando, puede ser interesante identificar los diferentes tipos de usuarios relevantes para nuestro caso de estudio en relación con la actividad que tienen en Twitter y, por lo tanto, con el uso que hacen de la aplicación. Esta identificación puede servir, por un lado, para entender mejor cómo se utiliza TwitterGaggle y, por otro lado, para ajustar la publicidad que se muestra en la aplicación a cada tipo de usuario.

Dentro de este contexto, puede ser interesante responder a preguntas como las siguientes:

- ¿Cuántos tuits, retuits y *replies* hace cada usuario? ¿Se pueden observar *clusters* de usuarios con un comportamiento similar en relación con el número de tuits, retuits y *replies* que envían?
- ¿En qué momento (día y franja horaria) publica mayoritariamente los tuits cada usuario? ¿Se pueden observar *clusters* de usuarios con comportamiento similar en relación con los horarios de publicación?
- ¿Con cuántas personas diferentes mantiene conversaciones mediante *replies* cada usuario? ¿Podemos caracterizar a los usuarios en función del número de personas con las que se comunican directamente?
- ¿En qué temas están interesados los usuarios?
- ¿Qué usuarios reenvían los mismos tuits?, es decir, ¿podemos clasificar a los usuarios en función de los retuits que realizan?
- ¿Quiénes son los mejores amigos de cada usuario?
- ¿Cuáles son los pares de usuarios con mayor número de seguidores en común?

1.5. Requisitos no funcionales

Para este primer estudio, GaggleCo pretende recoger toda la actividad en Twitter de 500 usuarios durante un periodo de 40 días. Además, se recogerán tam-

bién todas la relaciones de seguimiento de estos 500 usuarios seleccionados. Datos globales sobre la red estiman que en el 2012 la media de seguidores por usuario estaba cerca de los 200, que había aproximadamente 100 millones de cuentas activas de Twitter cada mes y que estas generaban unos 500 millones de tuits diarios. Además, la mayoría de los tuits no generaban ninguna reacción (ya sea en cuanto a reenvío o en cuanto a respuesta). Así pues, intentaremos hacer una estimación (*grosso modo*) del volumen de datos que se prevé usando estos datos globales de la red.

Teniendo en cuenta el número de tuits generados, se esperan recoger 100.000 tuits que representen actualizaciones de estado ($500/100 \times 500 \times 40 = 100.000$). Además, recogeremos también retuits y *replies* que todos estos tuits generen. Supondremos que cada tuit recibe al menos un retuit y un *reply*, con lo que la estimación para el número de tuits recogidos es de 300.000.

En relación con los usuarios identificados, conoceremos unos 100.000 usuarios que serán seguidores directos de los 500 usuarios relevantes ($500 \times 200 = 100.000$). En este cálculo estamos obviando que algunos de estos seguidores serán los mismos para diferentes usuarios, pero no necesitamos hacer cálculos exactos. Además, conoceremos también a todos los usuarios que reenvíen o contesten a tuits generados por los usuarios relevantes. Hemos contado que estos suponían 200.000 tuits, con lo que asumiremos que todos ellos provienen de usuarios diferentes; la estimación sobre el número de usuarios alcanzará entonces el valor de 300.000.

En relación con los requisitos de eficiencia, GaggleCo no necesita que las consultas sean extremadamente rápidas en ejecutarse, ya que no habrá ningún sistema funcionando en tiempo real sobre los datos almacenados. En cambio, sí que será necesario obtener el resultado de las consultas a tiempo para realizar el análisis y presentar los resultados, con lo que se espera que el tiempo máximo de ejecución de las consultas sea del orden de minutos.

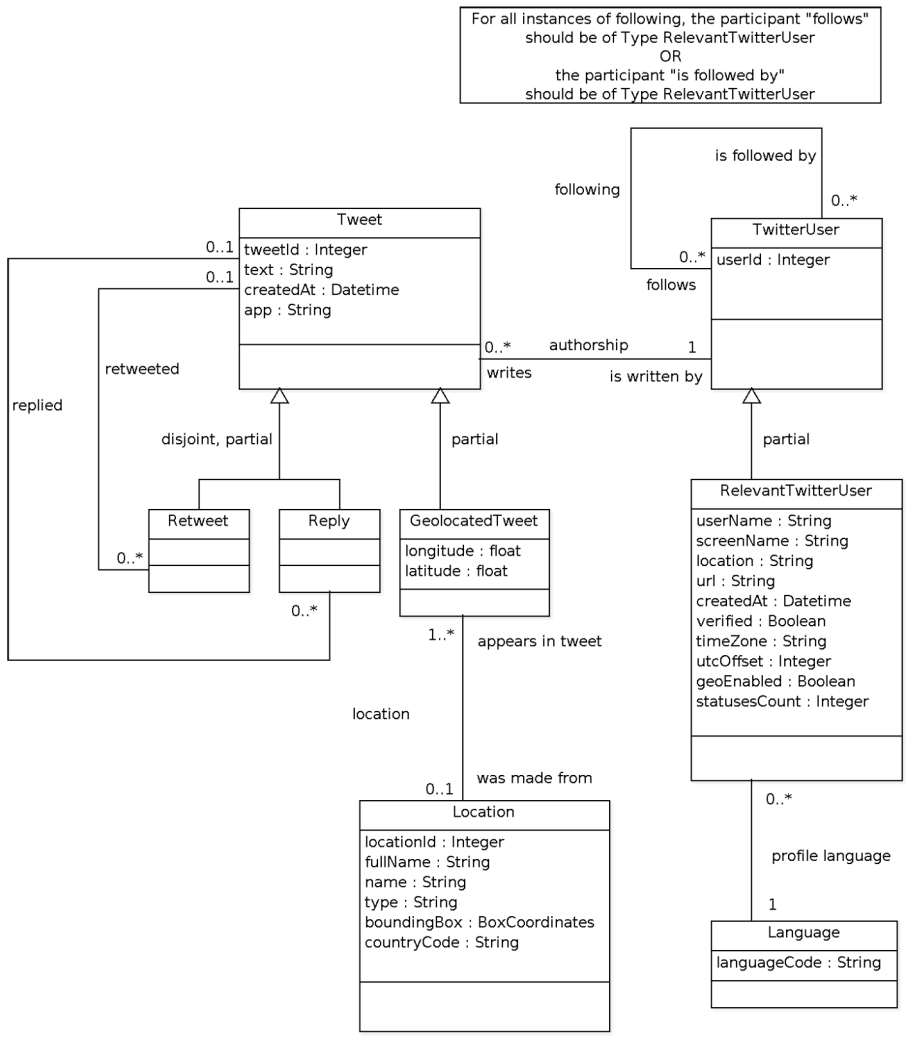
Aunque para esta primera prueba se han seleccionado únicamente 500 usuarios, si el proyecto tiene éxito, GaggleCo prevé tener un equipo centrado en analizar datos de redes sociales de manera permanente. Se desea que la solución desarrollada sea la base sobre la que este equipo pueda trabajar, por lo que la base de datos diseñada tiene que ser escalable. Así, se prevé que en el futuro se aumente tanto el número de usuarios seleccionados como relevantes, como la tipología de los datos recogidos. Además, se cree que en el futuro la recogida de tuits no se realizará durante periodos de tiempo limitados, sino que se hará de manera indefinida.

2. Diseño conceptual

Una vez tenemos claras las características de los datos que será necesario almacenar y de los que se extraerá posteriormente información, nos encontramos en posición de realizar el diseño conceptual de la base de datos. En este apartado, diseñaremos la estructura que tendrá la base de datos que almacenará la información sobre la actividad de los usuarios de Twitter. De momento, nos limitaremos al diseño conceptual, diseño que es totalmente independiente de la tecnología que usaremos para crear la base de datos.

La figura 1 muestra el diagrama de clases UML de un modelo conceptual que satisface los requisitos de datos expuestos en el apartado anterior. Cabe destacar que el modelo que proponemos no es único, es decir, que pueden existir otros modelos que permitan representar el mismo dominio de interés.

Figura 1. Diagrama de clases UML del modelo conceptual de datos



Las dos clases principales que se extraen de la descripción efectuada son los usuarios de Twitter (`TwitterUser`) y los mensajes que estos envían (`Tweet`).

En primer lugar, describiremos la clase que representa el usuario de Twitter (`TwitterUser`). Todos los usuarios tienen un atributo *userid* que los identifica dentro de la red. Sabemos también que de algunos usuarios de la red (en concreto, de aquellos que han sido seleccionados para el estudio) dispondremos de datos adicionales del perfil. Así pues, queremos modelar a los usuarios teniendo en cuenta que todos ellos tienen un identificador de usuario y que, además, algunos de ellos tienen datos adicionales en el perfil. Para hacerlo, definimos una clase `RelevantTwitterUser`, que es una especialización de `TwitterUser`, y que contiene los datos adicionales que se conocen sobre los usuarios relevantes del estudio. La especialización es parcial, ya que existirán usuarios de Twitter no relevantes de los cuales no tendremos la información complementaria del perfil.

Los usuarios relevantes (`RelevantTwitterUser`) disponen de un conjunto de atributos adicionales que los describen. Así, atributos como el nombre de usuario (*userName*), el nombre de perfil (*screenName*), la localización (*location*) y la url de la página personal (*url*) son cadenas de caracteres que el usuario puede introducir libremente. Otros atributos tienen un dominio diferente: la fecha de creación del perfil (*createdAt*) debe ser una fecha válida; los atributos que indican si el perfil está verificado o no (*verified*) y si el perfil tiene la geolocalización activada o no (*geoEnabled*) son booleanos; y la diferencia horaria con el tiempo universal coordinado en horas (*utcOffset*), así como el número de mensajes que el usuario ha publicado desde que se registró (*statusesCount*), se indican con un entero.

Twitter ofrece su interfaz en diferentes idiomas y cada usuario puede elegir en qué idioma visualizarla. Todos los perfiles están configurados para utilizar un idioma de los que Twitter ofrece y, aunque la situación sea poco probable, pueden existir idiomas que Twitter ofrezca y que nadie utilice. Además, aunque todos los usuarios tengan un idioma configurado, solo tendremos los datos del idioma del perfil de los usuarios relevantes. Por este motivo, se ha modelado el idioma como una clase (`Language`) que mantiene una asociación de uno (1) a muchos ($0..*$) con la clase de usuarios relevantes de Twitter. Así, los usuarios relevantes tendrán asociado un único idioma y un idioma puede ser utilizado en cero o más perfiles. El único atributo de la clase idioma es su código (*languageCode*).

Las relaciones explícitas de seguimiento entre los usuarios se han modelado como una asociación recursiva binaria (*following*) con conectividad de muchos ($0..*$) a muchos ($0..*$). De este modo, podemos representar que un usuario puede seguir a múltiples usuarios y, a su vez, un usuario puede ser seguido también por múltiples usuarios, a la vez que indicamos que es posible que un usuario no esté siguiendo a nadie ni sea seguido por nadie. Tal como he-

mos descrito, nuestra base de datos contendrá únicamente relaciones de seguimiento en las que uno de los participantes sea un usuario relevante. Por este motivo, se añade una restricción de integridad en el diagrama de clases UML, indicando que al menos uno de los dos participantes de la relación tiene que ser un usuario de Twitter relevante.

Aparte de los usuarios, la segunda clase principal del esquema descrito son los mensajes o tuits que los usuarios envían a través de la red (clase `Tweet`). Todos los tuits tienen un conjunto de atributos que se extrae directamente de la descripción del dominio: un identificador dentro de la red (*tweetId*), el texto que constituye el tuit en sí (*text*), la fecha de creación (*createdAt*) y una cadena de caracteres que la aplicación emisora del tuit incluye (*app*).

Existen dos tipos de tuits especiales: los retuits y los *replies*. Los retuits y *replies* son también tuits, es decir, contienen todos los datos que conforman un tuit, pero además representan o bien un reenvío o bien una respuesta de un tuit anterior. Por este motivo, se han modelado los retuits y *replies* como una especialización de la clase tuit (respectivamente, a través de las clases `Retweet` y `Reply`). La especialización es parcial, ya que pueden existir tuits que no sean *replies* ni retuits; y disjunta, porque un tuit no puede ser un retuit y un *reply* a la vez. Los retuits tienen una asociación con los tuits que nos permite indicar cuál es el tuit original que un retuit está reenviado. Cada retuit puede suponer un reenvío de un único tuit anterior pero, en cambio, pueden existir múltiples retuits (o ninguno) que hagan referencia a un mismo tuit original. Por este motivo, la conectividad de la asociación entre retuit y tuit (*retweeted*) es de $0..1$ a $0..*$. Una situación similar se produce con los *replies*, por lo que se han modelado del mismo modo: los *replies* participan en una asociación de $0..1$ a $0..*$ con los tuits (*replied*), que nos permite indicar, en este caso, a qué tuit responde un determinado *reply*. Nótese que en ambas asociaciones los tuits tienen una conectividad de $0..1$. Esto permite expresar que un retuit o un *reply* tendrá como mucho un tuit anterior al que hace referencia, pero también se puede dar el caso de que no tenga ninguno. Esto puede ocurrir ya que solo se recogen datos durante un periodo de tiempo concreto, por lo que el tuit anterior al que se hace referencia puede no haber estado capturado.

Algunos tuits contienen datos de geolocalización. Para modelarlos se ha creado la clase `GeoLocatedTweet`, que es una especialización de `Tweet` que contiene las coordenadas geográficas desde donde se ha creado el tuit. Al igual que con los retuits y *replies*, la especialización es parcial, ya que no todos los tuits serán geoposicionados. Todos los tuits geoposicionados contienen las coordenadas geográficas (*longitude* y *latitude*) pero, además, opcionalmente pueden indicar también la localización desde donde se han creado, entendiendo aquí por localización un punto de interés dado de alta previamente en Twitter. Estos puntos de interés se modelan con la clase `Location` y tienen como atributos: un identificador único (*locationId*), un nombre largo (*fullName*), un nombre corto (*name*), un tipo (*type*) y las coordenadas geográficas que definen

el espacio (*boundingBox*). Hemos definido el tipo de datos *BoxCoordinates* para representar los cuatro vértices que definen el rectángulo donde se engloba la localización, teniendo en cuenta que cada vértice se compone de dos valores *float* que indican su latitud (*latitude*) y su longitud (*longitude*). La asociación entre un `GeolocatedTweet` y una localización (*location*) tiene conectividad de $1..*$ a $0..1$, indicando que una localización puede estar vinculada a uno o varios tuits geolocalizados y que un tuit geolocalizado puede estar creado desde una localización (o desde ninguna).

Por último, se ha creado la asociación binaria *autorship*, que nos permite indicar el usuario autor de cada tuit. La asociación tiene conectividad de $0..*$ a 1, ya que todos los tuits tienen a un único usuario como autor y un mismo usuario puede escribir varios tuits (o incluso ninguno). Nótese que la clase que aparece en la asociación es `TwitterUser` (y no `RelevantTwitterUser`), ya que nuestra base de datos contendrá tanto tuits creados por usuarios relevantes como tuits creados por usuarios no relevantes. Estos últimos serán capturados porque van dirigidos a un usuario relevante o porque representan un retuit de un tuit escrito por un usuario relevante.

3. Alternativas tecnológicas

En este apartado se comparan las alternativas de continuar el diseño de la base de datos siguiendo el modelo relacional en comparación con hacerlo con un modelo orientado a grafos. Después, se justifica la elección de neo4j (una base de datos orientada a grafos) para almacenar los datos del caso de estudio.

3.1. Base de datos relacional frente a orientada a grafos

Teniendo en cuenta el dominio de interés expuesto en el apartado anterior, analizaremos las ventajas de una base de datos relacional y de una base de datos orientada a grafos para almacenar los datos objeto de este trabajo. Esto nos permitirá decidir qué paradigma será el más adecuado para almacenar los datos recopilados sobre la actividad de los usuarios en Twitter.

3.1.1. Ventajas de una base de datos orientada a grafos

A continuación se listan las principales ventajas de almacenar los datos descritos en una base de datos orientada a grafos:

- Los datos que almacenar se expresan de manera natural en forma de grafo. Así, a grandes rasgos, la base de datos que diseñar contendrá, por un lado, usuarios de Twitter y las relaciones que estos tienen en la red y, por otro lado, tuits, que estarán vinculados a los usuarios indicando autoría y, en ocasiones, a otros tuits a través de los retuits y *replies*. Por tanto, usando una base de datos orientada a grafos la representación de los datos será muy cercana al dominio de interés.

- El tipo de preguntas a las que se pretende responder para extraer información de los datos requiere realizar cálculos y aplicar algoritmos sobre grafos. Así, por ejemplo, las preguntas planteadas en el apartado anterior requieren el cálculo del camino más corto entre pares de nodos. Este tipo de consultas son ideales para las bases de datos orientadas a grafos, tanto por lo que se refiere a la expresión de las consultas como en lo relativo a eficiencia de estas.

- Los requisitos no funcionales recogidos especificaban la necesidad (condicionada al éxito del proyecto) de que la base de datos pueda incorporar nuevos tipos de datos en el futuro. El hecho de no requerir una definición

explícita del esquema de la base de datos y que las propiedades de los nodos no sean estrictas facilita la extensión *ad hoc* de la base de datos, de manera que nuevos tipos de nodos y relaciones, así como nuevas propiedades, pueden ser añadidas al grafo actual directamente, sin necesidad de redefinir el esquema de la base de datos.

- De un modo similar, representar los datos como un grafo hace que añadir nodos y aristas a un grafo ya existente permita expandir la base de datos sin que esto implique cambios en los procesos ya operativos sobre la base de datos. Así, se podrán adquirir nuevos datos sobre la actividad de usuarios de Twitter e incorporarlos al grafo sin tener que cambiar, por ejemplo, las consultas ya creadas sobre el grafo.

- También se prevé que la base de datos pueda crecer para almacenar grandes cantidades de nodos y relaciones. En las bases de datos con capacidad de procesamiento nativo, el tiempo de ejecución de las consultas no depende del tamaño del grafo, sino de la porción del grafo recorrido. Por tanto, utilizando una base de datos con esta propiedad podemos conseguir sistemas escalables en los que el tiempo de ejecución de las consultas no crezca con el tamaño del grafo.

3.1.2. Ventajas de una base de datos relacional

Veamos ahora las ventajas que supondría apostar por una alternativa relacional:

- Aunque no de manera tan eficiente como las bases de datos orientadas a grafos, las bases de datos relacionales también son capaces de expresar relaciones a través de claves foráneas y navegar por estas relaciones a través de *joins* (u operaciones de combinación) entre diferentes tablas. Así pues, es posible representar los datos especificados en el apartado anterior siguiendo el paradigma relacional.

- No todas las preguntas a las que GaggleCo quiere responder dentro del estudio necesitarán el uso de algoritmos sobre grafos para ser respondidas. Así, para el resto de las preguntas, el tiempo de ejecución de las consultas será comparable o incluso mejor para una base de datos relacional. Del mismo modo, la complejidad de expresar este tipo de consultas en SQL no tiene por qué ser superior a la complejidad de expresarlas sobre el grafo.

- El uso de una base de datos relacional permite realizar consultas sobre la base de datos usando SQL, un lenguaje estandarizado. Esto ofrecería robustez a las aplicaciones que GaggleCo desarrolle sobre la base de datos, ya que la sintaxis de SQL no cambia radicalmente de versión en versión, y minimiza el riesgo de problemas que puedan surgir de su uso, ya que SQL tiene un gran bagaje y existe una extensa documentación sobre él.

3.2. Elección de una base de datos orientada a grafos

Después de sopesar las ventajas que aportaría utilizar una base de datos orientada a grafos y una base de datos relacional, se ha decidido utilizar la primera para representar los datos de Twitter. Los motivos principales por los cuales se ha tomado esta decisión son, por un lado, el hecho de que las consultas planteadas exploten básicamente las relaciones entre los datos (lo que es idóneo para una base de datos orientada a grafos) y, por otro lado, la necesidad de que el esquema sea fácilmente evolucionable (lo que casi descarta, por sí solo, la utilización de un SGBD relacional).

Además, de entre los diferentes sistemas gestores de bases de datos orientados a grafos, se ha elegido neo4j. Los motivos de esta elección son los siguientes:

- Numerosas empresas de tamaño medio-grande utilizan neo4j para almacenar los datos. Además, el proyecto tiene ya cierta madurez, lo que tiende a garantizar su continuidad.

- Es una de las bases de datos orientada a grafos que proporciona las cuatro propiedades ACID para la gestión de transacciones: atomicidad, consistencia, aislamiento (*Isolation*) y durabilidad (o definitividad).

- Permite definir (de manera opcional) un esquema para el grafo, lo que posibilita incorporar restricciones sobre el grafo creado y crear índices sobre sus propiedades. Esto será útil para garantizar, por ejemplo, que no existen tuits ni usuarios duplicados en la base de datos, además de permitir agilizar las consultas sobre propiedades de los nodos.

- Dispone de un conjunto de herramientas (la consola y el navegador) que facilitan la interacción con la base de datos.

- Ofrece tanto un servidor *standalone* como la posibilidad de ser embebida dentro de una aplicación. Así, mientras que el desarrollo actual se realizará utilizando el servidor *standalone*, si en un futuro los requisitos de rendimiento crecen se puede optar por embeberla dentro de una aplicación.

- Permite realizar consultas a la base de datos usando Cypher, un lenguaje declarativo con una sintaxis visual y con puntos comunes con SQL, lo que hace que su aprendizaje sea fácil.

4. Diseño lógico

Una vez realizado el diseño conceptual de la base de datos y con la tecnología seleccionada, podemos pasar a realizar el diseño lógico de la base de datos orientada a grafos. En este apartado, veremos cómo podemos modelar nuestra base de datos en forma de grafo poniendo especial énfasis en los motivos por los cuales hemos tomado cada una de las decisiones de diseño.

4.1. Grafos

Formalmente, definimos un grafo como una estructura de datos que contiene un conjunto de **vértices** o **nodos** y un conjunto de **aristas** (por ejemplo, pares de nodos que representan las relaciones entre estos). El **orden** de un grafo es el número de nodos que contiene y el **tamaño** de un grafo es su número de aristas.

Se conoce como grafo no dirigido (o simplemente grafo) al grafo cuyas aristas no tienen dirección (es decir, al grafo cuyas aristas representan simplemente pares no ordenados de nodos). En cambio, se conoce como grafo dirigido o **digrafo** al grafo cuyas aristas tienen dirección (es decir, estas son pares ordenados de nodos). En este caso, llamamos a las aristas **arcos** o simplemente **aristas dirigidas**. Las aristas dirigidas tienen un **nodo inicial** (o nodo de origen) y un **nodo final** (o nodo de destino). Cuando existe una arista entre dos nodos, diremos que los nodos son **incidentes** a la arista y, de manera equivalente, que la arista es incidente a los dos nodos.

Los grafos son estructuras abstractas que se utilizan desde hace siglos para representar datos y resolver cierto tipo de problemas sobre estos datos.

En un grafo dirigido, se dice que el nodo u es **sucesor** del nodo v si existe una arista que va del nodo v al nodo u. De manera análoga, se dice que el nodo v es **antecesor** del nodo u si existe una arista que va del nodo v al nodo u. Esto nos permite definir el **grado exterior o de salida** de un nodo como el número de sucesores de este nodo. De forma similar, se puede definir el **grado interior o de entrada** de un nodo como el número de antecesores. Si estamos tratando con un grafo no dirigido, se utilizan los términos **nodos adyacentes o vecinos** de un nodo para definir al conjunto de los nodos con los cuales está relacionado por una arista. Entonces, el **grado** de un nodo es su número de vecinos.

4.2. El modelo de grafo con propiedades

Existen diferentes modelos que nos permiten representar el diseño lógico de una base de datos orientada a grafos. El modelo de grafo de propiedades es una de las alternativas más conocidas. Otras alternativas son el modelo de hipergrafos o las tripletas.

Un grafo de propiedades está formado por nodos, relaciones y propiedades. En un grafo de propiedades las relaciones siempre deben tener un nombre y son dirigidas. Así, todas las relaciones tienen siempre un nodo de origen y un nodo de destino. Tanto los nodos como las relaciones que los unen pueden tener propiedades, que no son más que pares de clave-valor. Las claves son siempre cadenas de caracteres, pero los valores pueden ser tipos de datos arbitrarios.

Así pues, un **grafo con propiedades** está formado por:

- **Nodos**: pueden tener aristas de salida y de entrada, pueden tener propiedades, pueden tener etiquetas.
- **Relaciones**: están etiquetadas, son dirigidas (tienen un nodo de origen y uno de destino), pueden tener propiedades.
- **Propiedades**: son pares de clave-valor, se asignan a los nodos y/o a las relaciones.

Grafo de propiedades y neo4j

El modelo de grafo de propiedades está estrechamente ligado a la implementación de neo4j. En versiones anteriores a la 2.0, neo4j no permitía tener etiquetas en los nodos y, por lo tanto, el modelo de grafo de propiedades no contemplaba el uso de estas. En nuevas versiones de neo4j, se ha añadido el soporte de etiquetas para los nodos, con lo que tiene sentido incorporarlas también al grafo de propiedades.

Nótese que un grafo de propiedades puede contener nodos aislados que no están conectados de ningún modo al resto del grafo (nodos sin aristas de salida ni de entrada). En cambio, las relaciones siempre tendrán un nodo de origen y un nodo de destino, es decir, no se pueden crear relaciones que no tengan explícitamente un origen y un destino. Además, las relaciones deben tener siempre una etiqueta que indique el tipo de relación. Los grafos de propiedades son multirrelacionales, ya que permiten representar diferentes tipos de aristas y, por lo tanto, diferentes tipos de relaciones.

Terminología

En este módulo, utilizaremos de manera indistinta las expresiones "un nodo tiene la etiqueta x" y "un nodo es de tipo x".

4.3. El diseño lógico de la base de datos

Así pues, el diseño lógico de la base de datos orientada a grafos consistirá en decidir cómo representar el modelo conceptual en forma de grafo, de manera que los datos estén almacenados en forma de nodos, relaciones entre estos nodos y propiedades de ambos. Por lo tanto, será necesario identificar qué elementos del modelo conceptual serán representados como nodos, cuáles serán representados como relaciones y cuáles serán propiedades de los nodos o las relaciones.

4.3.1. Definición de nodos y relaciones

En general, para representar el modelo conceptual en forma de grafo, representaremos los objetos de las clases identificadas en nuestro diseño conceptual como nodos del grafo. Una vez identificados los nodos, explicitaremos las relaciones necesarias entre estos nodos para representar todo nuestro dominio, sin olvidar añadir las propiedades necesarias tanto a los nodos como a las relaciones.

Así pues, utilizaremos ocho etiquetas distintas para caracterizar los nodos de nuestro grafo:

- `TwitterUser`
- `RelevantTwitterUser`
- `Tweet`
- `Reply`
- `Retweet`
- `GeoLocatedTweet`
- `Location`
- `Language`

Etiquetaremos cada nodo con las etiquetas correspondientes a cada clase de la jerarquía a la que pertenece su objeto asociado. Por tanto, cada nodo de nuestro grafo tendrá entre una y tres etiquetas. Por ejemplo, un nodo que represente un objeto de la clase usuario no relevante de Twitter tendrá únicamente la etiqueta `TwitterUser`; un nodo que represente un *reply* tendrá tanto la etiqueta `Reply` como la etiqueta `Tweet`, y un nodo que represente un retuit geolocalizado tendrá tres etiquetas: `Retweet`, `GeoLocatedTweet` y `Tweet`.

Una vez identificados los nodos del grafo, será necesario describir las relaciones que pueden existir entre estos nodos, es decir, las aristas del grafo. Teniendo en cuenta las relaciones identificadas en el modelo conceptual, definiremos seis tipos de relaciones distintas. La tabla 1 muestra los tipos de las relaciones que se crearán y los tipos de los nodos de origen y destino para cada relación.

Tabla 1. Tipos de relaciones

Relación	Tipo del nodo de origen	Tipo del nodo de destino
FOLLOWS*	TwitterUser	TwitterUser
IS_WRITEN_FROM	GeoLocatedTweet	Location
HAS_AS_PROFILE_LANGUAGE	RelevantTwitterUser	Language
HAS_WRITEN	TwitterUser	Tweet
IS_A_REPLY_OF	Reply	Tweet
IS_A_RETWEET_OF	Retweet	Tweet

***Uno de los dos participantes en la relación deberá ser de tipo `RelevantTwitterUser`.**

4.3.2. Las propiedades de los nodos

El modelo del grafo de propiedades permite asignar atributos a los nodos a través de sus propiedades. Estas propiedades son simplemente parejas de clave-valor.

Así pues, los nodos de tipo `TwitterUser` tendrán una propiedad correspondiente al identificador de usuario. Los nodos de tipo `RelevantTwitterUser` (que estarán también etiquetados como `TwitterUser`) tendrán, además del identificador de usuario, unas propiedades que nos ofrecerán datos adicionales sobre estos (el nombre de usuario, el nombre de perfil, la localización, la url, etc.).

Los nodos de tipo `Tweet` tendrán como propiedades el identificador del tuit, el texto, la fecha de creación y la aplicación desde la que fueron creados. Los nodos de tipo `Retweet` y `Reply` estarán también etiquetados con la etiqueta `Tweet` y no contendrán ninguna propiedad adicional. En cambio, los nodos de tipo `GeoLocatedTweet` (que también tendrán la etiqueta `Tweet`) tendrán dos propiedades adicionales con las coordenadas geográficas (longitud y latitud).

Los nodos de tipo `Language` contendrán una única propiedad con el código del idioma que representan.

Los nodos de tipo `Location` tendrán propiedades con su identificador, el nombre largo, el nombre corto, el tipo de localización, el código del país al que pertenecen y las coordenadas geográficas que definen el espacio.

4.3.3. Restricciones de unicidad

Neo4j y esquemas de grafos

Neo4j permite definir, de manera opcional, un esquema para el grafo. El esquema está centrado en el uso de etiquetas y nos permite definir algunas restricciones sobre el grafo.

Neo4j permite definir restricciones de unicidad sobre una propiedad para los nodos de un tipo concreto. Esto nos permite asegurar que no existirá más de un nodo (con una etiqueta específica) con el mismo valor para una propiedad dada.

Para el caso de estudio, es interesante aprovechar que neo4j nos permite crear restricciones de unicidad para asegurar que no existen identificadores duplicados para cada una de las clases descritas en la fase de diseño conceptual, es decir, para asegurar que no existen tuits, usuarios, localizaciones ni idiomas duplicados en nuestra base de datos.

Así pues, crearemos una restricción de unicidad sobre la propiedad *tweetId* de todos los nodos que representen un tuit (independientemente de que este represente una actualización de estado, un retuit, un *reply* o un tuit geolocalizado). Esto será posible porque todos ellos tendrán la etiqueta `Tweet`. Del

mismo modo, crearemos una restricción de unicidad sobre la propiedad *userId* de todos los nodos que representen usuarios de Twitter (independientemente de si son o no relevantes). De nuevo, esto será posible porque todos ellos tendrán la etiqueta `TwitterUser`. Además, crearemos también restricciones de unicidad sobre los identificadores de los idiomas y las localizaciones. La tabla 2 resume todas las restricciones de unicidad que crearemos en nuestra base de datos.

Tabla 2. Restricciones de unicidad

Tipo de nodo	Propiedad
`Tweet`	*tweetId*
`TwitterUser`	*userId*
`Language`	*languageCode*
`Location`	*locationId*

Neo4j y restricciones sobre más de una propiedad

En el momento de escribir estas líneas, neo4j (2.0.1) no soporta la creación de restricciones de unicidad sobre más de una propiedad a la vez. Es decir, se puede asegurar que el valor la propiedad A de un nodo con una etiqueta concreta es único, y también que el valor de la propiedad B para nodos con la misma etiqueta es también único. En cambio, no se puede añadir una restricción que asegure que el conjunto de los valores de las dos propiedades, (A,B), sea único.

4.3.4. Justificación de las decisiones tomadas

Los usuarios relevantes de Twitter (`RelevantTwitterUser`) tienen relaciones de seguimiento (`FOLLOWS`) que representan a qué usuarios está siguiendo cada uno de ellos. Estas relaciones siempre implicarán un nodo de origen o de destino que represente a un usuario relevante, ya que desconocemos las relaciones de seguimiento que existen entre pares de usuarios no relevantes.

Para el subconjunto de usuarios relevantes se dispone de datos sobre el idioma del perfil. Podríamos representar el idioma del perfil simplemente como una propiedad más de este tipo de nodos. No obstante, existen varias razones (algunas de más peso que otras) para decidir representar los idiomas de los perfiles como nodos en vez de como propiedades de los perfiles. Por un lado, los idiomas están predefinidos por Twitter y representan un conjunto finito (y bastante reducido) de posibilidades. Además, los idiomas pueden existir de manera independiente de los perfiles de los usuarios, es decir, pueden existir idiomas que no se estén usando en ningún perfil. Por otro lado, representar los idiomas como nodos nos puede facilitar la realización de cierto tipo de consultas, ya sea en cuanto a especificación de la propia consulta o en cuanto a rendimiento*. Por lo tanto, representamos los idiomas de los perfiles como nodos de tipo `Language` en nuestro grafo. El hecho de representar los idiomas como nodos nos induce a representar el idioma de un perfil como una relación entre el nodo que representa al usuario relevante y el nodo que representa el idioma. Esta relación está etiquetada como `HAS_AS_PROFILE_LANGUAGE`.

***Veremos algunos ejemplos de este caso en el apartado 6.**

Dado que los tuits representan mensajes que un usuario envía (a otros usuarios) en la red, podríamos haber intentado representar los tuits como relaciones entre usuarios (en concreto, relaciones entre el emisor del tuit y el destinatario). De todos modos, al profundizar un poco más en el dominio descrito, vemos que esta alternativa no es viable. Por un lado, como hemos expuesto, las relaciones en un modelo de grafo con propiedades (de manera similar a las aristas de un grafo dirigido abstracto) requieren un nodo de origen y un nodo

de destino. Aunque si bien es cierto que todos los tuits tienen un único emisor, la definición del nodo receptor se complica en nuestro dominio. Algunos tuits van dirigidos a usuarios concretos (los *replies*), pero los tuits que suponen actualizaciones de estado o los retuits no tienen un destinatario específico. Modelar estos tuits como aristas a todos los nodos de la red sería una posible interpretación, pero esto generaría una cantidad inmensa de relaciones que aportarían poca información. Además, representar incluso los *replies* (que son el tipo de tuit que encajaría más en esta representación) de este modo supone una pérdida de información, ya que no podríamos vincular una respuesta al tuit original al que está respondiendo.

Así pues, los tuits serán representados como nodos del grafo (de tipo `Tweet`).

Además, las relaciones de los tuits nos ofrecerán información muy rica sobre estos. Existirán relaciones entre nodos que representan tuits y nodos que representan usuarios de la red indicando el usuario creador de cada tuit. Estas relaciones están etiquetadas como `HAS_WRITEN`.

En el diseño conceptual, identificamos dos tipos de tuits especiales que representaban o bien una respuesta a un tuit anterior, o bien un reenvío de un tuit anterior. En el grafo, representamos estos dos tipos de tuit como dos tipos de nodos diferentes (respectivamente, `Reply` y `Retweet`). Recordemos que todos los nodos de tipo `Reply` y `Retweet` tendrán también la etiqueta `Tweet`.

Los nodos de tipo `Retweet` podrán tener también, además de las relaciones de autoría, una relación con el tuit que representa el mensaje original que se reenvía con el retuit. Estas relaciones estarán etiquetadas como `IS_A_RETWEET_OF`. De un modo análogo, los nodos de tipo `Reply` podrán tener también una relación con el tuit al que responden. Estas relaciones están etiquetadas como `IS_A_REPLY_OF`.

Las aristas de tipo `IS_A_RETWEET_OF` tendrán siempre como nodo inicial un nodo de tipo `Retweet`. Del mismo modo, las aristas de tipo `IS_A_REPLY_OF` tendrán como nodo inicial un `Reply`. En cambio, para ambos casos, el tipo del nodo final será `Tweet`. Los nodos finales podrán tener únicamente la etiqueta `Tweet` (cuando se esté respondiendo a una actualización de estado), pero también podrán tener las etiquetas `Retweet` o `Reply` (además de `Tweet`), cuando la respuesta o reenvío sea de un nodo de estos tipos.

La distinción entre los tres tipos de nodos (`Tweet`, `Reply` y `Retweet`) es necesaria para no perder información. Además, esta distinción nos permite identificar mejor el significado de cada nodo y facilitar definir las instancias de cada tipo.

Utilizaremos la etiqueta `GeoLocatedTweet` para identificar los tuits geolocalizados. Los tuits geolocalizados podrán participar en otro tipo de relacio-

nes cuando estén asociados a localizaciones concretas. Los nodos que representen mensajes geolocalizados que estén asociados a una localización tendrán una relación con los nodos que representen esa localización de tipo `IS_WRITEN_FROM`. Los tuits geolocalizados tendrán siempre tanto la etiqueta `GeoLocatedTweet` como la etiqueta `Tweet`. Adicionalmente, cuando estos representen retuits o *replies*, también podrán tener la etiqueta `RETWEET` o `REPLY`.

Herencia y clasificación

Nótese que el diseño conceptual de nuestra base de datos incluye varias relaciones de generalización/especialización. En concreto, las podemos encontrar entre las clases `RelevantTwitterUser` y `User`; entre `Retweet`, `Reply` y `Tweet`; y entre `GeolocatedTweet` y `Tweet`. Si tuviéramos que representar estas clases en una base de datos relacional, tendríamos que tomar una decisión a la hora de traducir el diseño conceptual al esquema lógico. Una de las alternativas sería crear más de una tabla para representar las diferentes clases (añadiendo las correspondientes claves foráneas). Otra alternativa sería diseñar una sola tabla que contenga los atributos comunes y los propios de cada clase. En este caso, no sería necesario crear claves foráneas. Para representar este tipo de clases en un grafo no es necesario seguir ninguna de las estrategias que aplicaríamos para bases de datos relacionales.

Dado que no es necesario especificar de manera explícita el esquema de los datos ni de manera estricta las propiedades que tiene cada nodo, pueden existir nodos de un mismo tipo con un conjunto de propiedades diferentes, y también se pueden definir tipos de nodos diferentes para cada clase si así se desea. Además, como hemos visto, también se puede definir un nodo con más de una etiqueta.

Así pues, las tres alternativas principales que se contemplan a la hora de representar las relaciones de generalización/especialización que hemos identificado en el diseño conceptual son las siguientes:

Representación de las relaciones

La decisión de representar las relaciones de generalización/especialización de un modo u otro no tiene que ser forzosamente homogénea para toda la base de datos. Por ejemplo, podríamos elegir utilizar una estrategia para los tuits y otra diferente para los usuarios de Twitter.

- Disponer de una única etiqueta, correspondiente a la superclase. Es decir, para nuestro caso de estudio, disponer de las etiquetas: `TwitterUser`, `Tweet`, `Location` y `Language`. Si optamos por esta alternativa, habrá nodos de tipo `Tweet` que tendrán únicamente las propiedades de la clase `Tweet`, pero también podrán existir nodos del mismo tipo con propiedades como la longitud o la latitud.

  En este caso, la jerarquía se aplana por arriba, colapsando todas las subclases en la superclase. Es el caso más sencillo, ya que genera el menor número de tipos de nodos, pero también es el que supone una pérdida de información mayor, ya que desconoceremos de qué tipo es cada nodo (únicamente

sabremos el tipo de la superclase a la que pertenece). Además, esta alternativa puede generar grafos incorrectos, puesto que no se podrán garantizar las restricciones de integridad.

- Disponer de etiquetas para todas las clases (ya sean o no especializaciones de una superclase), etiquetando cada nodo con la etiqueta más específica posible. Para el caso de estudio, esto implicaría tener etiquetas para `Reply`, `Retweet`, `GeoLocatedTweet`, `Tweet`, `RelevantTwitterUser` y `TwitterUser` (además de `Location` y `Language`). Así pues, un nodo que represente un usuario relevante tendría únicamente la etiqueta `RelevantTwitterUser`.

  Esta alternativa representa el enfoque totalmente opuesto a la alternativa anterior. En este caso, la jerarquía se aplana por debajo, instanciando todos los objetos con la clase más específica a la que pertenecen. Aunque se pierde menos semántica que en el caso anterior, se sigue perdiendo información. Además, también como en el caso anterior, nos podríamos encontrar con instanciaciones del esquema que no son válidas.

- Disponer de etiquetas para todas las clases (ya sean o no especializaciones de una superclase), etiquetando cada nodo con todas las etiquetas de su jerarquía. Para el caso de estudio, esto implicaría tener las mismas etiquetas que en la situación anterior (`Reply`, `Retweet`, `GeoLocatedTweet`, `Tweet`, `RelevantTwitterUser`, `TwitterUser`, `Location` y `Language`). La diferencia radica en que ahora un nodo que represente un usuario relevante tendría dos etiquetas: `RelevantTwitterUser` y `TwitterUser`. Siguiendo esta línea, podría existir un nodo con hasta tres etiquetas, que correspondería a un retuit (o *reply*) que está geoposicionado.

  Esta última alternativa es la mejor desde un punto de vista formal y permite mantener la semántica de los datos. Como inconvenientes, para casos extremos con herencia con muchas clases se puede generar una cantidad significativa de información innecesaria y se puede complicar la gestión de la base de datos.

Dado que a nivel formal la última alternativa es mejor que las dos anteriores y que para el caso de estudio aplicar esta última alternativa no supone generar una cantidad muy grande de información adicional, hemos elegido esta opción para representar nuestros datos. La descripción del grafo siguiendo esta última estrategia corresponde a la descripción realizada en el subapartado 4.3.1.

Nótese que al no representar el esquema de la base de datos de manera explícita, la herencia no se representa a nivel de esquema sino a nivel de los datos. Por lo tanto, la herencia se tratará en la forma en que se clasifican los datos según sus tipos.

Nótese también que no se hace ningún tipo de definición para garantizar la disyunción en la jerarquía de herencia de `Retweet` y `Reply`. Se asume que el usuario (o la aplicación) que interactúe con la base de datos respetará esta disyunción y no creará nodos que tengan ambas etiquetas (`Retweet` y `Reply`).

4.4. Consideraciones adicionales

En este subapartado, repasaremos algunas consideraciones adicionales que pueden ser interesantes a la hora de realizar el diseño lógico de una base de datos orientada a grafos.

4.4.1. Dirección de las aristas

Otro punto interesante que cabe remarcar es la dirección de las aristas del grafo. Como hemos visto, el modelo de grafo de propiedades solo permite relaciones direccionales, que distinguen entre el nodo inicial y el nodo final. En este sentido, hay tres puntos que remarcar.

En primer lugar, dados dos nodos entre los cuales existe una relación dirigida, siempre podemos elegir entre expresar la relación en un sentido o en el sentido contrario. Por ejemplo, para nuestro caso, podemos decidir expresar que el usuario u sigue al usuario v (u `FOLLOWS` v) o bien, de manera equivalente, expresar que el usuario v es seguido por el usuario u (v `IS_FOLLOWED_BY` u). Algunos tipos de relaciones se expresarán de manera más natural en uno de los dos sentidos pero, para otras relaciones, puede no haber mucha diferencia entre expresar la relación en un sentido o en el sentido contrario.

En segundo lugar, dependiendo del tipo de las aristas, también se puede dar el caso de que entre dos nodos (u y v) existan dos aristas del mismo tipo, una en cada sentido (es decir, una arista de u a v y una de v a u). En cambio, para otros casos este escenario no tendría sentido. Por ejemplo, para las relaciones de tipo `FOLLOWS`, es posible que un usuario u esté siguiendo a otro usuario v y, a su vez, que el usuario v esté siguiendo también al usuario u. En cambio, para una relación de tipo `HAS_WRITTEN` nunca nos encontraremos con esta situación, ya que un usuario puede escribir un tuit pero un tuit nunca escribirá un usuario.

En tercer lugar, el hecho de que los grafos de propiedades solo permitan aristas dirigidas provoca que todas las relaciones tengan que ser, forzosamente, representadas por aristas de este tipo. Esto hace que ciertos tipos de relaciones, inherentemente no orientadas, tengan que ser representadas como aristas con dirección. Sería el caso, por ejemplo, de una relación que implicara amistad entre dos nodos. La amistad es una propiedad (en principio) recíproca y, por lo tanto, si quisiéramos representarla en un grafo dirigido sería necesario incluir, para todos los pares de nodos amigos, las aristas en los dos sentidos

(u es amigo de v y v es amigo de u). Aunque en ocasiones se elige esta representación, esta solución incluye mucha redundancia. Por este motivo, a veces se opta por representar solo las aristas en uno de los dos sentidos, interpretando entonces las aristas dirigidas como si no lo fuesen. Dado que las aristas se pueden navegar en ambas direcciones, esta última representación no supone ningún inconveniente a la hora de realizar consultas sobre el grafo.

4.4.2. El grado de los nodos

Al representar los datos en forma de grafo, el grado de los nodos indica información que puede ser de interés. Así, por ejemplo, podemos saber cuántos perfiles utilizan un idioma concreto mirando el grado de entrada del nodo del idioma en cuestión; cuántos tuits se envían desde una localización concreta mirando también el grado de entrada del nodo que representa la localización; cuántos tuits ha escrito un usuario dado observando el grado de salida del usuario, teniendo en cuenta únicamente las relaciones de tipo `HAS_WRITTEN`; a cuántos usuarios sigue un usuario relevante estudiando su grado de salida atendiendo solamente a las relaciones de tipo `FOLLOWS`; o cuántas respuestas ha tenido un determinado tuit analizando el grado de entrada del nodo tuit teniendo en cuenta únicamente las relaciones `IS_A_REPLY_OF`.

4.4.3. Claves foráneas

Si quisiéramos representar el modelo conceptual descrito en el apartado 3 en una base de datos relacional, lo traduciríamos a tablas y añadiríamos claves foráneas entre algunas tablas para representar ciertos tipos de relaciones. Así, por ejemplo, seguramente añadiríamos una clave foránea que asegurase que cada tuit ha sido escrito por un usuario de Twitter existente en nuestra base de datos.

En cambio, no hay claves foráneas en neo4j (ni en las bases de datos orientadas a grafos). Los nodos de las diferentes clases se encuentran interconectados entre ellos a través de las aristas del grafo. Siguiendo con el ejemplo de los autores de los tuits, tendremos aristas (de tipo `HAS_WRITEN`) entre el nodo que representa el tuit y el nodo que representa al usuario. Además, estas aristas pueden contener información adicional (propiedades en formato clave-valor), aunque no tenemos ningún ejemplo de este caso en la base de datos que estamos construyendo.

4.5. Un pequeño ejemplo

La figura 2 presenta un pequeño grafo de ejemplo siguiendo el diseño lógico de nuestra base de datos. Como se puede apreciar, existen ocho tipos de nodos diferentes, representando las clases usuario de Twitter (`TwitterUser`), usuario relevante de Twitter (`RelevantTwitterUser`), tuit (`Tweet`), retuit

(`Retweet`), *reply* (`Reply`), tuit geolocalizado (`GeoLocatedTweet`), idioma del perfil (`Language`) y localización (`Location`).

Figura 2. Un pequeño ejemplo

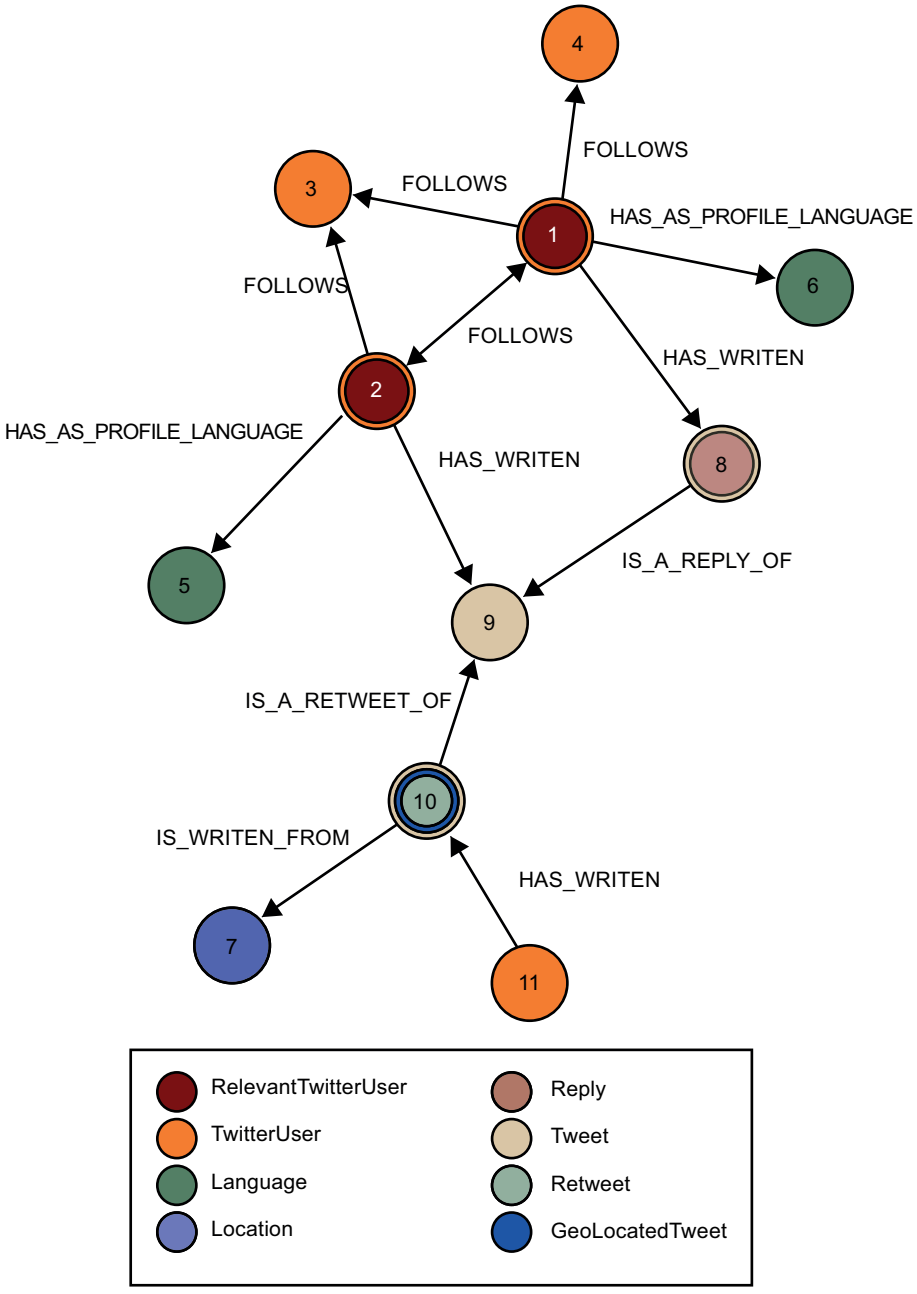


Figura 2

En el grafo de la figura 2, los nodos muestran una etiqueta con un número único dentro del grafo. Esto nos permite identificarlos de manera única, lo que facilita las explicaciones. En general, se pueden modificar las propiedades de visualización de los grafos para que la representación gráfica obtenida sea lo más comprensible posible. Por ejemplo, se puede visualizar algún atributo de interés del nodo como etiqueta, se puede ajustar el color de los nodos, se pueden reorganizar los nodos en el espacio (modificar el *layout*), etc.

Los usuarios relevantes de Twitter (`RelevantTwitterUser`) tienen relaciones de seguimiento (`FOLLOWS`) que representan a qué usuarios está siguiendo cada uno de ellos. El nodo 1 está siguiendo a los nodos 2, 3 y 4; y el nodo 2 está siguiendo a los nodos 1 y 3. Los nodos 3 y 4 son usuarios de Twitter no relevantes (`TwitterUser`) que se han incluido en la base de datos por su relación con los nodos relevantes (`RelevantTwitterUser`). Así pues, desconocemos las relaciones de seguimiento que podrían existir entre los nodos 3 y 4, ya que los usuarios que representan no son relevantes. Como se puede apreciar, este tipo de datos (usuarios y relaciones de seguimiento entre ellos) se expresan de

forma natural utilizando grafos, de manera que la representación obtenida es muy cercana a la propia idea que queremos representar.

Para el subconjunto de usuarios relevantes se dispone de datos sobre el idioma del perfil (`Language`). En el grafo de ejemplo, existen relaciones `HAS_AS_PROFILE_LANGUAGE` entre los usuarios relevantes (nodos 1 y 2 del grafo) y el idioma de su perfil (nodos 6 y 5).

El grafo muestra un tuit que supone una actualización de estado (nodo 9), un *reply* (nodo 8) y un retuit (nodo 10). Este último es, además, un tuit geolocalizado, por lo que tiene también la etiqueta `GeoLocatedTweet`.

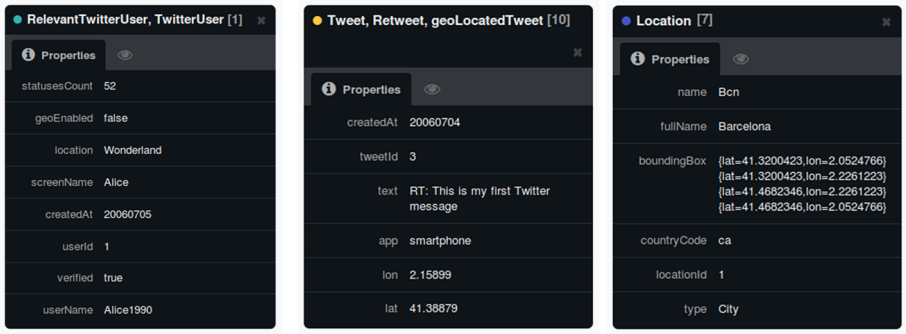
Cada tuit tiene una relación de tipo `HAS_WRITEN` con el usuario que lo ha escrito, de manera que en el grafo se expresa que los usuarios representados por los nodos 1, 2 y 11 han escrito los tuits representados por los nodos 8, 9 y 10, respectivamente.

Siguiendo con el ejemplo de la figura, podemos observar que el tuit representado por el nodo 10 es un retuit del tuit representado por el nodo 9, lo que en el grafo se indica mediante la arista `IS_A_RETWEET_OF`. De un modo análogo, podemos ver cómo el tuit representado por el nodo 8 es un *reply* del tuit representado por el nodo 9.

Finalmente, los nodos que representen mensajes geolocalizados que estén asociados a una localización concreta tendrán una relación con los nodos que representen esa localización. Este tipo de relaciones se etiqueta como `IS_WRITEN_FROM`. En el grafo de la figura, se puede observar cómo el nodo 10 ha sido enviado desde la localización representada por el nodo 7.

Como hemos descrito, los nodos pueden tener propiedades asociadas. A modo de ejemplo, la figura 3 muestra las propiedades de tres nodos: un nodo de tipo `RelevantTwitterUser`, uno de tipo `Tweet` y uno de tipo `Location`. Nótese que en el caso de estudio no existe ningún ejemplo de arista con propiedades. Esto ocurre porque no se ha necesitado expresar propiedades en las relaciones, pero para otros dominios puede ser interesante asignar propiedades a las aristas.

Figura 3. Ejemplo de propiedades de nodos de tipo `RelevantTwitterUser`, `Tweet` y `Location`



4.6. Mejoras

Una vez tenemos un primer grafo que se adapta al modelo conceptual, el proceso de diseño lógico normalmente incluye también una fase en la que se enriquece la representación. En esta fase, se pueden añadir relaciones entre los nodos existentes o incluso añadir nodos, con el objetivo de obtener una representación lo más adecuada posible para alcanzar los objetivos de explotación de datos que se han propuesto para el dominio de aplicación.

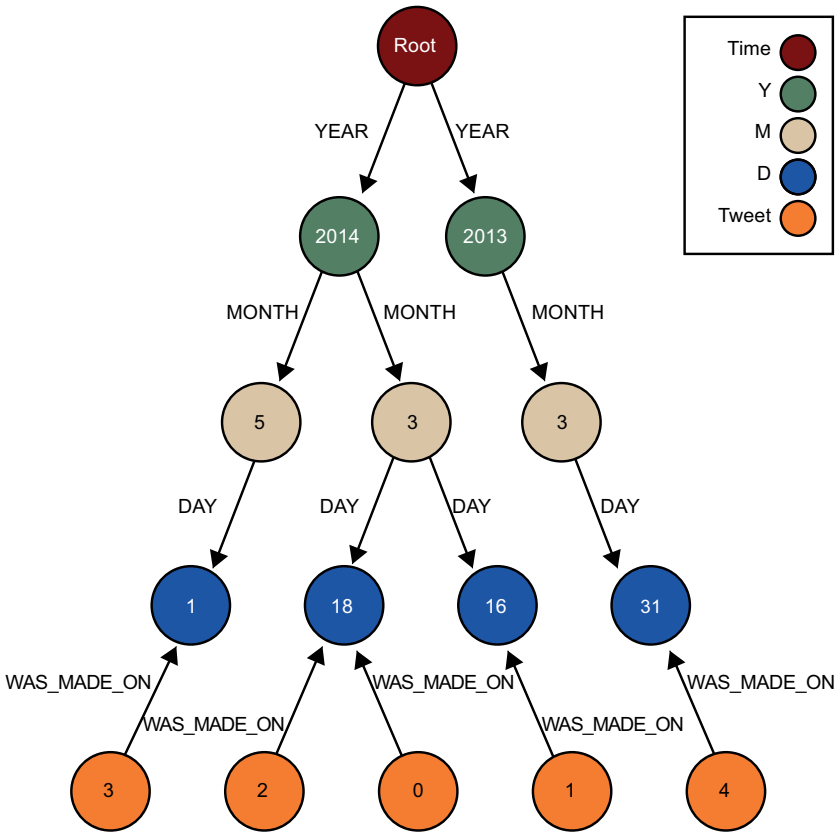
4.6.1. Representación del tiempo

Los datos que se quieren capturar de Twitter contienen datos temporales. Así pues, los usuarios relevantes tienen asociada la fecha en la que el perfil fue creado y todos los tuits incluyen también la fecha en la que fueron enviados. Hasta ahora, hemos modelado estas fechas como propiedades de los respectivos nodos. Dependiendo del tipo de consultas que queramos hacer sobre estos datos, la representación como propiedades será suficiente para nuestros propósitos. Por ejemplo, si dado un tuit concreto queremos saber en qué año fue creado, podremos acceder fácilmente a la propiedad *createdAt* del tuit y obtener esta información. En cambio, para otro tipo de consultas esta representación resultará bastante pobre y creará problemas de eficiencia. Esto ocurre, por ejemplo, si es necesario acceder a todos los tuits que fueron creados durante un día específico o si queremos recorrer secuencialmente todos los tuits que ha escrito un usuario concreto. Veamos cómo podemos representar los datos temporales para facilitar este tipo de búsquedas.

Si estamos interesados en realizar consultas que recojan todos los tuits de un día concreto (o de un periodo de tiempo concreto), podemos modificar el grafo para representar las fechas de creación de los tuits como un árbol. Así, el árbol estaría formado por un nodo raíz que tiene como descendientes a n nodos que representan los años (conectados mediante aristas de tipo `YEAR`), que a su vez tienen como descendientes los nodos que representan los meses de cada año (conectados mediante aristas de tipo `MONTH`) y estos últimos tienen como descendientes a los nodos que representan los días de cada mes (conectados mediante aristas de tipo `DAY`). No es necesario crear nodos para todas las fechas posibles, sino simplemente para aquellas en las que se ha creado algún tuit. La figura 4 muestra un ejemplo de cómo se usaría este tipo de árbol para señalar las fechas de creación de los tuits.

Con esta representación de los datos y asumiendo que el nodo raíz es fácil de descubrir, podemos entonces seleccionar de manera sencilla todos los tuits que se han creado en un día concreto, navegando desde la raíz al día en cuestión, pasando por el año y el mes adecuados, y obteniendo entonces acceso a todos los tuits creados ese día. De un modo similar, podemos seleccionar todos los tuits que fueron enviados durante un mes concreto, navegando el árbol desde la raíz al mes en cuestión y seleccionando luego todos los nodos situados a distancia dos (siguiendo las relaciones `DAY` y `WAS_MADE_ON`).

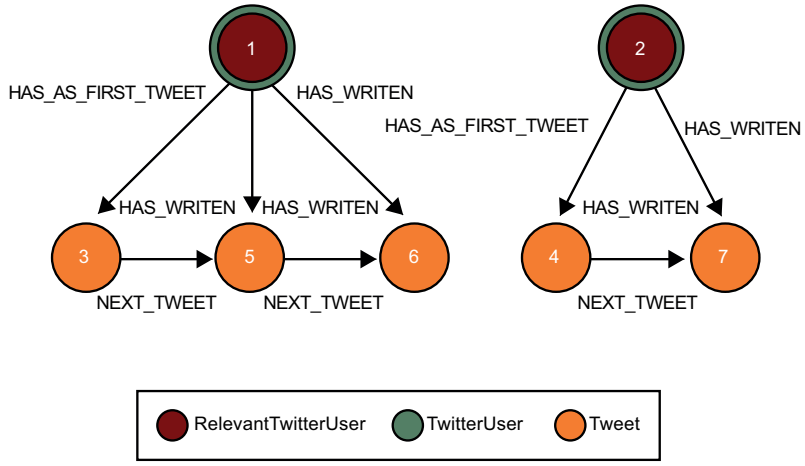
Figura 4. Representación de las fechas como árbol



Esta representación permite responder fácilmente a la pregunta sobre cuáles son los tuits que fueron creados en un día concreto pero, en cambio, no ayuda mucho a responder a la segunda pregunta: ¿Cómo podemos recorrer secuencialmente todos los tuits de un usuario?

Si prevemos que necesitaremos responder a esta última pregunta, nos puede interesar cambiar la representación para optimizar su respuesta. En este caso, una posible solución es añadir aristas entre los tuits de cada usuario, de manera que cada tuit esté conectado con el siguiente tuit del usuario. De este modo, se crea una lista enlazada de los tuits de cada usuario, que nos permite recorrerlos de manera cronológica, del más antiguo al más reciente. La figura 5 muestra un ejemplo de cómo se podría usar esta representación.

Figura 5. Lista enlazada de los tuits de cada usuario



5. Diseño físico

Después de haber realizado el diseño lógico de la base de datos que almacenará los datos recogidos sobre los usuarios de Twitter, nos encontramos en posición de realizar el diseño físico. El diseño físico de una base de datos consiste en crear el grafo de propiedades que hemos descrito en el apartado 4 en una base de datos orientada a grafos, en este caso, neo4j. En este apartado, se describe el proceso de creación de la base de datos sobre neo4j.

5.1. Neo4j

El sistema gestor de bases de datos elegido para implementar la base de datos orientada a grafos es neo4j. Neo4j puede operar en dos modos distintos: como servidor *standalone* o como base de datos embebida en una aplicación. Cuando neo4j opera como servidor, la base de datos puede ser accedida a través de la REST API. Con el segundo modo de operación, neo4j se puede embeber dentro de aplicaciones escritas en cualquier lenguaje que soporte una máquina virtual de Java (por ejemplo, Scala o el propio Java). El anexo A (apartado 7) incluye una pequeña guía de instalación y uso de neo4j.

Además, neo4j dispone de Cypher, un lenguaje de consultas nativo que proporciona un método independiente del *framework* para acceder a la base de datos. Cypher es un lenguaje declarativo que permite realizar consultas sobre grafos. Entre sus ventajas, destacan una sintaxis muy visual (basada en ASCII art), su simplicidad de uso y su eficiencia para realizar consultas sobre grafos. En este documento, utilizaremos Cypher para crear el grafo con los datos sobre Twitter que GaggleCo ha recogido para su estudio, y veremos cómo podemos realizar consultas utilizando este mismo lenguaje. En el anexo B (apartado 8), se incluye un pequeño resumen de la sintaxis básica del lenguaje.

Neo4j permite traducir de manera directa el modelo lógico descrito como un grafo de propiedades a una base de datos.

5.2. Creación de la base de datos

El siguiente código permite crear el grafo de ejemplo del apartado anterior (figura 2) con Cypher. La creación de nodos y relaciones en Cypher se realiza utilizando el comando `CREATE`. Todo el contenido de la declaración se ejecutará como una única transacción en neo4j, de manera que una vez se ha

ejecutado podemos estar seguros de que o bien se ha creado el grafo entero, o bien la transacción ha fallado y ninguna parte del grafo ha sido creada.

```
CREATE
( catalan:Language{
      languageCode:'ca' } ),
( english:Language{
      languageCode:'en'} ),
( alice:RelevantTwitterUser:TwitterUser{
      userId:1 , userName:'Alice1990' , screenName:'Alice' ,
      location:'Wonderland' , createdAt:20060705 , verified:True ,
      geoEnabled:False , statusesCount:52 } ),
( bob:RelevantTwitterUser:TwitterUser{
      userId:2 , userName:'Robert' , screenName:'RobertTheGreat' ,
      location:'Cryptoland' , createdAt:20060701 , verified:True ,
      geoEnabled:True , statusesCount:500 } ),
( carol:TwitterUser{
      userId:3 } ),
( david:TwitterUser{
      userId: 4 } ),
( eve:TwitterUser{
      userId: 5 } ),
( barcelona:Location{
      locationId:1 , fullName: 'Barcelona' ,
      name: 'Bcn' , type:'City' , countryCode: 'ca',
      boundingBox:'
            {lat=41.3200423,lon=2.0524766}
            {lat=41.3200423,lon=2.2261223}
            {lat=41.4682346,lon=2.2261223}
            {lat=41.4682346,lon=2.0524766}' } ),
(t1:Reply:Tweet{
      tweetId:2 , text:'@Bob Hi, Bob, nice to see you in Twitter!' ,
      createdAt:20060703 , app:'web' } ) ,
(t0:Tweet{
      tweetId:1 , text:'This is my first Twitter message' ,
      createdAt:20060701 , app:'web' } ) ,
(t2:Retweet:Tweet:geoLocatedTweet{
      tweetId:3 , text:'RT: This is my first Twitter message' ,
      createdAt:20060704 , app:'smartphone' ,
      lon: 2.1589900 , lat: 41.3887900 } ) ,
(alice)-[:FOLLOWS]->(bob),
(bob)-[:FOLLOWS]->(alice),
(bob)-[:FOLLOWS]->(david),
(alice)-[:FOLLOWS]->(eve),
(alice)-[:FOLLOWS]->(david),
(alice)-[:HAS_AS_PROFILE_LANGUAGE]->(english),
(bob)-[:HAS_AS_PROFILE_LANGUAGE]->(catalan),
(bob)-[:HAS_WRITEN]->(t0),
(alice)-[:HAS_WRITEN]->(t1),
(t1)-[:IS_A_REPLY_OF]->(t0),
(carol)-[:HAS_WRITEN]->(t2),
(t2)-[:IS_A_RETWEET_OF]->(t0),
(t2)-[:IS_WRITEN_FROM]->(barcelona);
```

En el código de ejemplo podemos observar cómo, en primer lugar, se crean varios nodos de diferentes tipos (con sus respectivas propiedades) y, después, se crean relaciones etiquetadas entre estos nodos.

El primer nodo que se crea tiene la etiqueta `Language`. El nodo tiene una única propiedad (*languageCode*) con su respectivo valor (*ca*). Este primer nodo tiene un único tipo (`Language`). En cambio, el tercer nodo que se crea es un usuario de Twitter relevante y tiene, por lo tanto, dos etiquetas diferentes: `RelevantTwitterUser` y `TwitterUser`. De manera similar al primer nodo,

este también contiene un conjunto de propiedades con sus respectivos valores (el identificador de usuario es 1, el nombre de usuario es *Alice1990*, el nombre que aparece en el perfil es *Alice*, etc.).

El nodo recientemente creado tiene asignado el identificador *alice*. Este identificador (que no se debe confundir con el atributo *userId*) es temporal, y solo estará disponible dentro del ámbito de la instrucción de creación. Una vez ejecutada la instrucción, el identificador no será accesible, y no será posible hacer referencia al nodo a través de él.

El resto de los nodos se crean siguiendo la misma estructura, especificando un identificador temporal que permite hacer referencia a ellos dentro de la instrucción de creación, definiendo el tipo de nodo e incluyendo los valores de las propiedades que lo describen en formato clave:valor. Nótese que Cypher usa como símbolo para separar la clave del valor los dos puntos (:) en vez de el guión (-).

Para crear las relaciones entre nodos, aprovechamos los identificadores temporales que se han asignado a cada uno de los nodos. Así, podemos crear relaciones explicitando los dos nodos participantes, el tipo de relación y la dirección. Siguiendo con el código de ejemplo, la primera relación que se crea nos indica que el usuario con identificador temporal *alice* está siguiendo (`FOLLOWS`) al usuario con identificador temporal *bob*.

Una vez se ha creado el grafo, se podrán realizar las modificaciones que se desee. Así, por ejemplo, se podrán volver a ejecutar instrucciones `CREATE` para añadir nodos y relaciones al grafo ya existente. Además, se puede utilizar también la instrucción `CREATE UNIQUE` para asegurar que la subestructura que se está creando, que puede contener tanto nodos y relaciones ya existentes en el grafo como nodos y relaciones nuevas, sea la estructura que queda en el grafo después de la ejecución de la instrucción (sin que exista duplicación de nodos ni de aristas).

La estrategia usada para crear relaciones entre los diferentes nodos utiliza los identificadores temporales de los nodos para hacer referencia a ellos. En consecuencia, esta estrategia no será útil cuando queramos añadir relaciones entre nodos ya existentes del grafo, puesto que los identificadores temporales ya no estarán disponibles. El mismo problema aparecerá cuando se intente crear una grafo grande y sea necesario dividir la instrucción de creación del grafo en varias instrucciones más pequeñas. En estos casos, podemos hacer referencia a los nodos a partir de una (o varias) de sus propiedades utilizando la instrucción `MATCH`, y crear después la relación entre los nodos que se han encontrado. El código siguiente muestra un ejemplo de la creación de dos nodos y de la relación que los une en dos instrucciones independientes.

```
CREATE
( alice:TwitterUser{
      userId:1 , userName:'Alice1990' , screenName:'Alice' ,
      location:'Wonderland' , createdAt:20060705 , verified:True ,
      geoEnabled:False , statusesCount:52 } ),
( bob:RelevantTwitterUser:TwitterUser{
      userId:2 , userName:'Robert' , screenName:'RobertTheGreat' ,
      location:'Cryptoland' , createdAt:20060701 , verified:True ,
      geoEnabled:True , statusesCount:500 } );

MATCH (u1:TwitterUser{userId:1}), (u2:TwitterUser{userId:2})
CREATE (u1)-[:FOLLOWS]->(u2);
```

En este caso, la cláusula `MATCH` busca dos nodos de tipo `TwitterUser` que tengan como *userId* los valores 1 y 2. En los casos en los que sea necesario utilizar este tipo de instrucciones en repetidas ocasiones, será interesante crear primero un índice sobre la propiedad que utilizaremos para buscar los nodos, con el objetivo de agilizar el proceso.

5.3. Creación de índices

Las bases de datos basadas en grafos que disponen de capacidad de procesamiento nativo tienen una propiedad conocida como adyacencia libre de índice (*index-free adjacency*).

> La adyacencia libre de índice (*index-free adjacency*) es una propiedad de las bases de datos basadas en grafos con capacidad de procesamiento nativo. En las bases de datos con esta propiedad, cada nodo mantiene un puntero directo a sus nodos adyacentes, de manera que cada nodo actúa como un pequeño índice de los nodos vecinos.

Como consecuencia directa de la adyacencia libre de índice, el tiempo de ejecución de las consultas (que utilizan estos índices) es independiente del tamaño total del grafo. En concreto, el tiempo de ejecución pasará a depender de la proporción del grafo explorado por la consulta. Esto hace que los recorridos sobre el grafo sean extremadamente eficientes en estas bases de datos.

Creación de índices en neo4j

La creación de índices en neo4j no es inmediata. Al lanzar una instrucción de creación de un índice, se lanza un proceso en segundo plano que construye el índice. Aunque la operación de creación del índice retorna de manera inmediata, el proceso que está construyendo el índice puede necesitar más tiempo para realizar esta tarea y, por lo tanto, el índice puede no estar disponible de manera inmediata.

De todos modos, nos puede interesar realizar consultas que no consistan en navegar por el grafo, sino que se centren en las propiedades de los nodos. Por ejemplo, podríamos estar interesados en obtener todas las localizaciones dadas de alta en un país concreto (propiedad *countryCode* del nodo de tipo `Location`). Si solo dispusiéramos de los índices que apuntan a los nodos vecinos (y no tuviéramos etiquetas ni ningún tipo de índice), sería necesario recorrer todo el grafo para encontrar los nodos de tipo `Location` que tuvieran el valor de la propiedad *countryCode* deseada.

Neo4j permite la creación de índices sobre propiedades de nodos etiquetados. El siguiente código muestra cómo se crean índices para algunas de las propiedades sobre las que puede ser interesante realizar consultas en nuestra base de datos.

```
CREATE INDEX ON :Location(countryCode);
CREATE INDEX ON :Tweet(app);
CREATE INDEX ON :RelevantTwitterUser(userName);
CREATE INDEX ON :RelevantTwitterUser(screenName);
CREATE INDEX ON :RelevantTwitterUser(geoEnabled);
```

5.4. Creación de restricciones

Tal y como hemos descrito en el subapartado 4.3.3 es interesante añadir restricciones de unicidad sobre las propiedades que representan los identificadores de tuits, usuarios de Twitter, idiomas y localizaciones. Neo4j permite crear este tipo de restricciones sobre propiedades de nodos que tienen una etiqueta específica. Así pues, añadiremos las restricciones para los nodos de tipo `Tweet`, `TwitterUser`, `Language` y `Location`. El código siguiente nos muestra cómo se pueden crear estas restricciones.

```
CREATE CONSTRAINT ON (tw:Tweet) ASSERT tw.tweetId IS UNIQUE;
CREATE CONSTRAINT ON (us:TwitterUser) ASSERT us.userId IS UNIQUE;
CREATE CONSTRAINT ON (la:Language) ASSERT la.languageCode IS UNIQUE;
CREATE CONSTRAINT ON (lo:Location) ASSERT lo.locationId IS UNIQUE;
```

Una vez creadas las restricciones de unicidad, si intentamos añadir un nodo de un tipo concreto con el valor duplicado de la propiedad sobre la que hemos aplicado la restricción, obtendremos un error que nos informará que no es posible (ya que ya existe un nodo de ese tipo con ese valor para la propiedad en cuestión) y se descartarán automáticamente los cambios efectuados.

La creación de una restricción de unicidad sobre una propiedad supone implícitamente la creación de un índice sobre esta misma propiedad. Por lo tanto, no será necesario añadir índices en propiedades sobre las que hayamos credo una restricción de unicidad con anterioridad. La eliminación de una restricción de unicidad implicará también la eliminación del correspondiente índice, por lo que si deseamos mantener un índice sobre la propiedad y eliminar la restricción de unicidad, será necesario añadir el índice a posteriori.

SCHEMA

Se pueden listar las restricciones y los índices sobre etiquetas existentes en una base de datos con el comando SCHEMA (en la consola de neo4j, NEO4J-SHELL) o con el comando :SCHEMA (del navegador de neo4j, NEO4J-BROWSER).

5.5. Transacciones

Las transacciones en neo4j funcionan del mismo modo que en bases de datos relacionales. El conjunto de las instrucciones que pertenecen a una misma transacción forman una unidad de trabajo y se ejecutan de forma atómica.

Neo4j soporta las cuatro propiedades ACID: atomicidad, consistencia, aislamiento (*isolation*) y durabilidad (o definitividad).

Neo4j permite señalizar el inicio de una transacción mediante la instrucción `BEGIN`, finalizar la transacción de forma exitosa con `COMMIT` y cancelar la transacción con `ROLLBACK`.

Cuando se elimina un nodo (o una relación) del grafo, las propiedades de la entidad se eliminan de manera automática. En cambio, al eliminar un nodo las relaciones que este tiene con otros nodos del grafo no se eliminan. Dado que todas las relaciones en la base de datos tienen que tener un nodo de inicio y un nodo final válidos, no podemos eliminar un nodo si este participa en alguna relación. De todos modos, la restricción se aplica en el momento de finalizar la transacción, por lo que sí que se puede eliminar primero el nodo y después la relación, siempre que las dos operaciones se realicen dentro de una misma transacción. El código siguiente muestra un ejemplo de un intento de eliminar un nodo que participa en una relación. El resultado de la ejecución produce un error, ya que no se puede eliminar el nodo sin eliminar primero la relación.

```
CREATE
( u1: TwitterUser{userId:1} ),
( u2: TwitterUser{userId:2} ),
(u1)-[:FOLLOWS]->(u2);

MATCH (u1:TwitterUser {userId:1})
DELETE u1;
```

```
TransactionFailureException: Unable to commit transaction
```

Consistencia

La situación descrita se relaciona con la consistencia, una de las propiedades ACID, que exige que las transacciones deben preservar las restricciones de integridad definidas en la base de datos, a pesar de que, mientras la transacción está activa, la base de datos puede caer momentáneamente en un estado inconsistente.

En cambio, el próximo código muestra el comportamiento de la misma eliminación si esta está incluida en una transacción que también elimina las relaciones. En este caso, la transacción finaliza con éxito.

```
CREATE
( u1: TwitterUser{userId:1} ),
( u2: TwitterUser{userId:2} ),
(u1)-[:FOLLOWS]->(u2);

BEGIN
MATCH (u1:TwitterUser {userId:1})
DELETE u1;
MATCH ()-[r]->(:TwitterUser {userId:2})
DELETE r;
COMMIT
```

```
Nodes deleted: 1
Relationships deleted: 1
Transaction committed
```

Aunque el ejemplo de la eliminación de un nodo y sus aristas incidentes es útil para mostrar el funcionamiento de las transacciones en neo4j, este no es

el método que utilizaríamos habitualmente para realizar esta eliminación. El siguiente código muestra la secuencia de operaciones más frecuente.

```
CREATE
( u1: TwitterUser{userId:1} ),
( u2: TwitterUser{userId:2} ),
(u1)-[:FOLLOWS]->(u2);

MATCH (u1:TwitterUser {userId:1})-[r]->(:TwitterUser {userId:2})
DELETE r, u1;
```

```
Nodes deleted: 1
Relationships deleted: 1
```

6. Comparativa con una base de datos relacional

En este apartado, compararemos la base de datos creada en neo4j con una base de datos relacional capaz de almacenar los mismos datos. Para hacerlo, en primer lugar, realizaremos el diseño de la base de datos siguiendo el modelo relacional. Después, crearemos las consultas en Cypher (para neo4j) que permiten responder a algunas de las preguntas que se planteaban al inicio de este documento, así como las consultas equivalentes en SQL (para la base de datos relacional en MySQL). Por último, compararemos las consultas obtenidas en ambos lenguajes.

6.1. Diseño para base de datos relacional

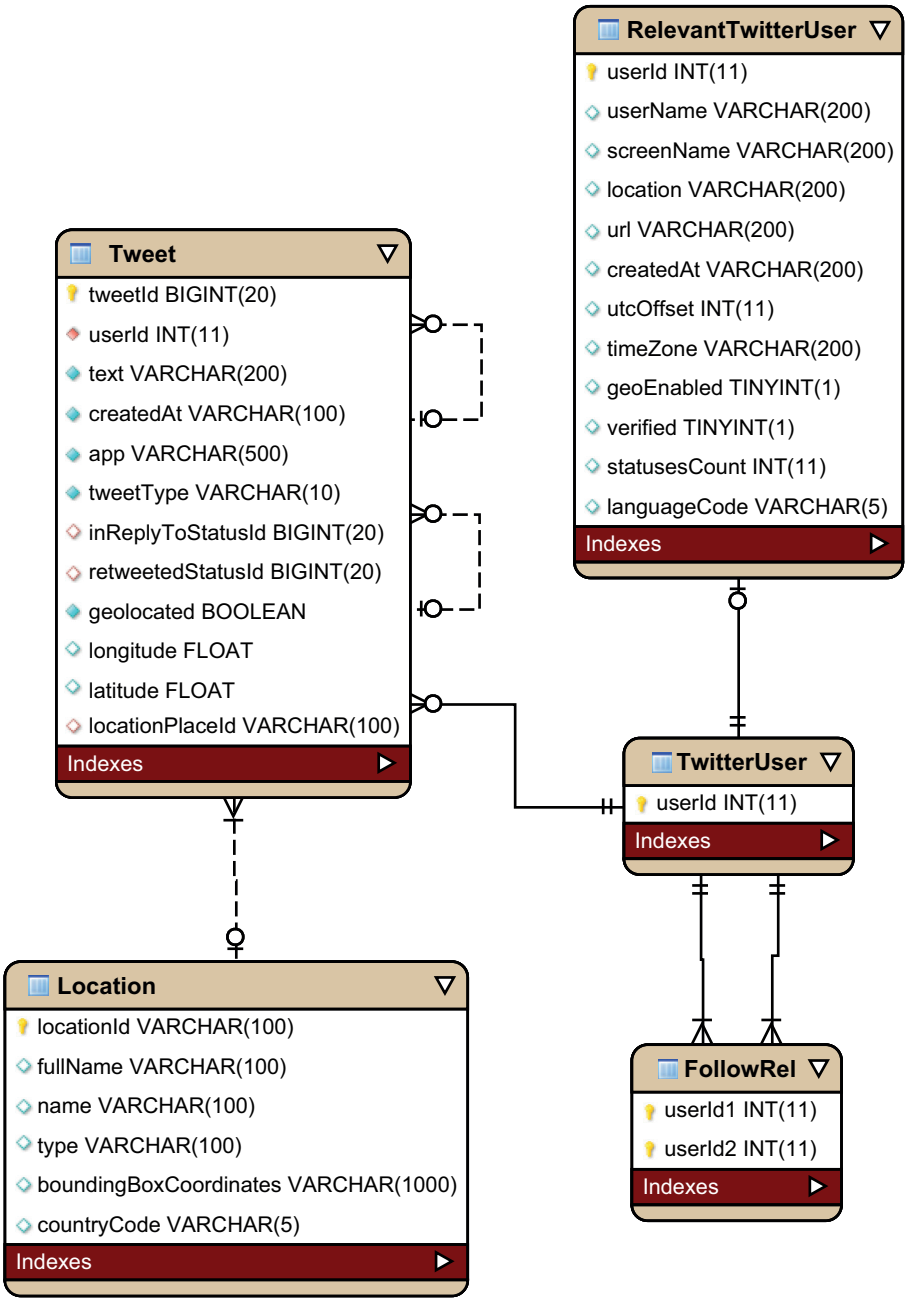
Para poder comparar la base de datos creada mediante neo4j con otra base de datos ajustada al modelo relacional, realizaremos primero el diseño de una base de datos relacional que permita representar la información modelada durante el diseño conceptual de nuestro dominio de interés.

La figura 6 muestra el diagrama del diseño para una base de datos relacional. El diseño incluye 5 tablas: `Tweet`, `TwitterUser`, `RelevantTwitterUser`, `Location` y `FollowRel`. Dado que el objetivo principal de este documento no es analizar la construcción de una base de datos relacional, repasaremos de manera muy rápida las decisiones tomadas durante su diseño.

Para representar las relaciones de generalización/especialización, hemos optado por dos estrategias distintas.

Dado que el número de usuarios de Twitter no relevantes será varios órdenes de magnitud superior al número de usuarios de Twitter relevantes y que los usuarios relevantes tienen muchos más atributos que los usuarios no relevantes, utilizaremos una tabla para representar todos los atributos comunes (que será únicamente el *userId*) y otra tabla para representar los atributos específicos de la subclase. Así, la tabla `RelevantTwitterUser` contendrá columnas para representar todos los atributos de los usuarios relevantes de Twitter y la tabla `TwitterUser` contendrá únicamente una columna con el identificador de usuario. Además, añadiremos una clave foránea en el identificador de usuario de la tabla de usuarios relevantes que hará referencia al identificador de usuario de la tabla de usuarios de Twitter.

Figura 6. Diseño de la base de datos relacional



En cambio, dado que los tuits geolocalizados (así como los *replies* y retuits) no tienen tantos atributos de más que los tuits que suponen actualizaciones de estado, ni la proporción de unos y otros es tan marcadamente superior, utilizaremos la estrategia de crear una única tabla con todos los atributos de las subclases para representar el resto de las relaciones de generalización/especialización. Así, crearemos una única tabla `Tweet` que representará las clases `Tweet`, `Retweet`, `Reply` y `GeoLocatedTweet`. La tabla tendrá, además de las columnas comunes a todas estas clases, unas columnas adicionales: *inReplyToStatusId* se utilizará por los *replies* para hacer referencia al tuit anterior al que responden (cuando este sea conocido); *retweetedStatusId* se utilizará por los retuits para hacer referencia al tuit que reenvían (cuando este sea conocido); *longitude* y *latitude* se utilizarán por los tuits geolocalizados para incluir la información de geolocalización; y *locationPlaceId* se utilizará también por los tuits geolocalizados que estén asociados a una localización concreta. Todas

estas columnas tomarán el valor *null* cuando no sean aplicables o cuando se desconozca la información.

Además, se añaden dos columnas adicionales: *tweetType* y *geolocated*. La columna *tweetType* contendrá una cadena de caracteres que definirá el tipo de tuit. Los valores posibles para esta columna serán tuit, retuit y *Reply*. Esto permitirá mantener la semántica asociada a las generalizaciones/especializaciones de tuits propuestos en el diseño conceptual. Por otro lado, la columna *geolocated* contendrá un valor booleano que indicará si el tuit en cuestión es o no un tuit geolocalizado.

Se añadirá también una clave foránea sobre la columna *locationPlaceId*, que hará referencia al identificador de la localización de la tabla `Location`, y dos claves foráneas sobre las columnas *inReplyToStatusId* y *retweetedStatusId*, que harán referencia a la columna *tweetId* de la misma tabla.

retweetedStatusId

Podrá darse el caso de que algunos retuits (respectivamente *replies*) hagan referencia a un tuit anterior que no se haya capturado. En este caso, la columna *retweetedStatusId* (respectivamente *inReplyToStatusId*) contendrá el valor *null*. Podremos diferenciar estos tuits de las actualizaciones de estado (que también tendrán la columna *retweetedStatusId* y *inReplyToStatusId* a *null*) gracias al contenido de la columna *tweetType*.

La relación de autoría (*authorship*) se ha convertido en una clave foránea entre la tabla `Tweet` y la tabla `TwitterUser`.

La relación de seguimiento entre usuarios (*Following*) es de $0..*$ a $0..*$, por lo que necesitamos una tabla adicional para poder representarla. La tabla `FollowRel` permite expresar esta relación, e incluye dos claves foráneas que harán referencia a la columna de identificador de usuario de la tabla de usuarios de Twitter (`TwitterUser`). La pareja de claves foráneas, a su vez, son la clave primaria de la tabla.

Por último, dado que la clase que representaba el idioma del perfil contiene un único atributo con el código del idioma y que este participaba en una relación $0..*$ a 1 con la clase de usuarios relevantes de Twitter, se ha añadido una columna *languageCode* a la tabla `RelevantTwitterUser` para representarla. Esto impide que se puedan tener idiomas que no sean usados por ningún usuario, pero permite simplificar la representación.

El siguiente código muestra la creación de las tablas de la base de datos para MySQL.

```
CREATE TABLE `TwitterUser` (
  `userId` int(11),
  PRIMARY KEY (`userId`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE `RelevantTwitterUser` (
  `userId` int(11),
  `userName` varchar(200) DEFAULT NULL,
  `screenName` varchar(200) DEFAULT NULL,
  `location` varchar(200) DEFAULT NULL,
  `url` varchar(200) DEFAULT NULL,
  `createdAt` varchar(200) DEFAULT NULL,
  `utcOffset` int(11) DEFAULT NULL,
  `timeZone` varchar(200) DEFAULT NULL,
  `geoEnabled` tinyint(1) DEFAULT NULL,
```

```
  `verified` tinyint(1) DEFAULT NULL,
  `statusesCount` int(11) DEFAULT NULL,
  `languageCode` varchar(5) DEFAULT NULL,
  PRIMARY KEY (`userId`),
  CONSTRAINT `fk_publicTwitterUser_TwitterUser`
      FOREIGN KEY (`userId`) REFERENCES `TwitterUser` (`userId`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE `Location` (
  `locationId` varchar(100),
  `fullName` varchar(100) DEFAULT NULL,
  `name` varchar(100) DEFAULT NULL,
  `type` varchar(100) DEFAULT NULL,
  `boundingBoxCoordinates` varchar(1000) DEFAULT NULL,
  `countryCode` varchar(5) DEFAULT NULL,
  PRIMARY KEY (`locationId`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE `Tweet` (
  `tweetId` bigint(20),
  `userId` int(11) NOT NULL,
  `text` varchar(200) NOT NULL,
  `createdAt` varchar(100) NOT NULL,
  `app` varchar(500) NOT NULL,
  `tweetType` varchar(10) NOT NULL,
  `inReplyToStatusId` bigint(20) DEFAULT NULL,
  `retweetedStatusId` bigint(20) DEFAULT NULL,
  `geolocated` boolean NOT NULL,
  `longitude` float DEFAULT NULL,
  `latitude` float DEFAULT NULL,
  `locationPlaceId` varchar(100) DEFAULT NULL,
  PRIMARY KEY (`tweetId`),
  CONSTRAINT `Tweet_ibfk_1`
      FOREIGN KEY (`locationPlaceId`) REFERENCES `Location`
    (`locationId`),
  CONSTRAINT `Tweet_ibfk_2`
      FOREIGN KEY (`userId`) REFERENCES `TwitterUser` (`userId`),
  CONSTRAINT `Tweet_ibfk_3`
      FOREIGN KEY (`inReplyToStatusId`) REFERENCES `Tweet` (`tweetId`),
  CONSTRAINT `Tweet_ibfk_4`
      FOREIGN KEY (`retweetedStatusId`) REFERENCES `Tweet` (`tweetId`),
  CHECK ( (tweetType='Tweet' AND inReplyToStatusId is NULL)
    OR (tweetType='Retweet' AND inReplyToStatusId is NULL)
    OR (tweetType='Reply') ),
  CHECK ( (tweetType='Tweet' AND retweetedStatusId is NULL)
    OR (tweetType='Reply' AND retweetedStatusId is NULL)
    OR (tweetType='Retweet') ),
  CHECK ( (geolocated=True AND latitude is not NULL AND longitude is
    not NULL)
    OR (geolocated=FALSE AND latitude is NULL AND longitude is NULL) )
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

CREATE TABLE `FollowRel` (
  `userId1` int(11),
  `userId2` int(11),
  PRIMARY KEY (`userId1`,`userId2`),
  CONSTRAINT `fk_followRel1_TwitterUser`
      FOREIGN KEY (`userId1`) REFERENCES `TwitterUser` (`userId`),
  CONSTRAINT `fk_followRel2_TwitterUser`
      FOREIGN KEY (`userId2`) REFERENCES `TwitterUser` (`userId`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Cláusulas CHECK

La tabla tuit contiene cláusulas CHECK que intentan asegurar que no se producirá una falta de consistencia en la base de datos. Estas cláusulas intentan evitar, por ejemplo, que puedan existir tuits con *tweetType* que tengan un valor de *retweetedStatusId* diferente de *null*. De todos modos, la versión actual (5.5.37) de MySQL omite estas cláusulas, permitiendo que sean infringidas las restricciones impuestas por los CHECKS.

6.2. Ejemplos de consultas

En este subapartado, intentaremos responder a algunas de las preguntas que se planteaban durante la descripción del dominio. Por un lado, crearemos las consultas en Cypher sobre la base de datos neo4j que permiten responder a ellas y, por otro lado, describiremos las consultas SQL sobre la base de datos

Véase también

Recordad que la descripción del dominio se realiza en el subapartado 1.4 de este módulo.

relacional en MySQL que permitirían responder a estas mismas preguntas. Esto nos permitirá comparar ambos enfoques y analizar la complejidad de expresarlas en los dos lenguajes.

6.2.1. La actividad en la red

GaggleCo había seleccionado un conjunto de preguntas sobre la actividad de los usuarios relevantes en Twitter. La primera pregunta que se planteaba era: "¿Cuántos tuits envían de media los usuarios?". La base de datos contiene todos los tuits enviados por los usuarios relevantes durante el periodo de recopilación de datos. Por lo tanto, podemos contar cuántos tuits ha enviado cada usuario relevante y calcular la media de este valor utilizando la función `avg` disponible tanto en Cypher como en SQL. El siguiente código muestra la consulta que nos permite calcular la media del número de tuits de los usuarios relevantes, así como el resultado de la consulta.

```
MATCH (u:RelevantTwitterUser)-[:HAS_WRITEN]->(t:Tweet)
WITH u, count(t) AS numTweetsPerUser
RETURN avg(numTweetsPerUser);
```

```
SELECT avg( a.numTweetsPerUser ) FROM
      (SELECT count(*) AS numTweetsPerUser
      FROM Tweet AS t, RelevantTwitterUser AS u
      WHERE u.userId = t.userId GROUP BY u.userId) AS a;
```

```
518.6226
```

Como se puede ver, la sintaxis de la consulta es más sencilla en Cypher, ya que solo es necesario describir el patrón que queremos buscar, contar el número de tuits por usuario y hacer la media. El patrón que hemos de buscar en este caso es simple: indicamos que buscamos nodos de tipo `RelevantTwitterUser` que tengan una arista de tipo `HAS_WRITEN` hacia un nodo de tipo `Tweet`. Después, para cada usuario relevante, contamos el número de tuits encontrados y asignamos el identificador `numTweetsPerUser` a este resultado y, por último, calculamos la media de los valores `numTweetsPerUser`.

La siguiente pregunta que se planteaba era: ¿Qué proporción de los tuits enviados son retuits? Una manera fácil de expresar la consulta en Cypher es utilizar dos cláusulas `MATCH`, una para contar el número de retuits que existen y la otra para contar el número de tuits. Utilizaremos `WITH` para indicar que necesitaremos el identificador `numRetweets` en el `RETURN`.

Limitaciones de Cypher

En el momento de escribir estas líneas, Cypher no permite hacer conversión de tipos ni existe un operador para definir la división real entre dos valores enteros. Por tanto, forzamos la conversión de entero a real al multiplicar el numerador por 100.0, de manera que obtengamos la división real entre los dos valores, que es el resultado que buscamos.

```
MATCH (r:Retweet) WITH count(r) AS numRetweets
MATCH (t:Tweet) WITH count(t) AS numTweets,
      numRetweets AS numRetweets
RETURN (numRetweets*100.0)/numTweets AS retweetPercentage;
```

```
SELECT t2.numRetweets/t1.numTweets*100 AS retweetPercentage
FROM
      ( SELECT count(*) AS numTweets FROM Tweet ) AS t1,
      ( SELECT count(*) AS numRetweets FROM Tweet
      WHERE tweetType = 'Retweet' ) AS t2
```

```
61.1528
```

Las consultas que calculan la proporción de *replies* y tuits geolocalizados se pueden calcular de manera análoga, y dejaremos su realización como ejercicio para el estudiante.

Se preguntaba también cuántos tuits están asociados a una localización de Twitter. Podemos responder a esta pregunta en Cypher describiendo el patrón de los tuits geolocalizados que están escritos desde una localización dada de alta en Twitter:

```
MATCH (g:GeoLocatedTweet)-[:IS_WRITEN_FROM]->(:Location)
WITH count(g) AS numTweetsWithLoc
MATCH (t:Tweet) WITH count(t) AS numTweets,
      numTweetsWithLoc AS numTweetsWithLoc
RETURN (numTweetsWithLoc*100.0)/numTweets AS
      numTweetsWithLocPercentage;
```

```
SELECT t2.numTweetsWithLoc/t1.numTweets*100 AS
   numTweetsWithLocPercentage
FROM
      ( SELECT count(*) AS numTweets FROM Tweet ) AS t1,
      ( SELECT count(*) AS numTweetsWithLoc FROM Tweet
      WHERE geolocated = True AND locationPlaceId IS NOT NULL) AS t2;
```

```
0.6180
```

6.2.2. Las aplicaciones que interactúan con Twitter

Antes de realizar consultas destinadas a intentar aumentar el número de usuarios de TwitterGaggle, es importante analizar cuál es la situación actual del mercado de clientes de Twitter.

En este sentido, se planteaban preguntas relacionadas como ¿cuál es el cliente de Twitter más usado? o ¿cuál es el ranquin de herramientas usadas?

Tanto en Cypher como en SQL podemos utilizar las instrucciones `ORDER BY` para ordenar la salida en función de una propiedad o columna y `LIMIT` para limitar el número de resultados obtenidos. El siguiente código muestra cómo obtener (en Cypher y SQL) las cinco aplicaciones más usadas, entendiendo como más usadas aquellas que se han utilizado para enviar un mayor número de tuits.

```
MATCH (t:Tweet)
WITH t.app AS app, count(*) AS numTweetsPerApp
WHERE app <> 'web'
RETURN app, numTweetsPerApp
ORDER BY numTweetsPerApp DESC LIMIT 5;
```

```
SELECT app, count(*) AS numTweetsPerApp
FROM Tweet
WHERE app != 'web'
GROUP BY app ORDER BY numTweetsPerApp DESC LIMIT 5;
```

```
+-------------------------+-----------------+
| app                     | numTweetsPerApp |
+-------------------------+-----------------+
| Twitter for iPhone      |          342772 |
```

```
| TweetDeck                |          231687 |
| Twitter for BlackBerry   |          143236 |
| Twitter for Android      |          127499 |
| Echofon                  |           68744 |
+--------------------------+-----------------+
5 rows
```

6.2.3. Detección de usuarios importantes

Otro de los aspectos que cabe analizar se centra en la detección de usuarios importantes, que pueden ser la clave para propagar información sobre TwitterGaggle. Así, por ejemplo, nos preguntábamos cuáles son los usuarios relevantes con más seguidores directos y aquellos con más seguidores de segundo y tercer nivel. Los siguientes tres pares de consultas muestran cómo podríamos obtener los cinco usuarios con más seguidores directos, de segundo y de tercer nivel con Cypher y con SQL:

```
MATCH (:TwitterUser)-[:FOLLOWS]->(u:RelevantTwitterUser)
WITH u, count(*) AS numFollowers
RETURN u.userId, numFollowers
ORDER BY numFollowers DESC LIMIT 5;
```

```
SELECT u.userId, count(*) AS numFollowers
FROM FollowRel AS r, RelevantTwitterUser AS u
WHERE r.userId2 = u.userId
GROUP BY u.userId ORDER BY numFollowers DESC LIMIT 5;
```

```
+-------------------------+
| u.userId | numFollowers |
+-------------------------+
| 8627932  | 79           |
| 14919552 | 74           |
| 20536157 | 72           |
| 6809022  | 65           |
| 9206682  | 63           |
+-------------------------+
5 rows
```

```
MATCH (:TwitterUser)-[:FOLLOWS]->(:TwitterUser)-
      [:FOLLOWS]->(r:RelevantTwitterUser)
WITH r, count(*) AS numFollowers2nLevel
RETURN r.userId, numFollowers2nLevel
ORDER BY numFollowers2nLevel DESC LIMIT 5;
```

```
SELECT u.userId, count(*) AS numFollowers2nLevel
FROM FollowRel AS r1, FollowRel AS r2, RelevantTwitterUser AS u
WHERE r1.userId2 = r2.userId1 AND r2.userId2 = u.userId
GROUP BY u.userId ORDER BY numFollowers2nLevel DESC LIMIT 5;
```

```
+--------------------------------+
| r.userId | numFollowers2nLevel |
+--------------------------------+
| 8627932  | 1976                |
| 14919552 | 1703                |
| 4029671  | 1646                |
| 9206682  | 1640                |
| 3233231  | 1556                |
+--------------------------------+
5 rows
```

```
MATCH (:TwitterUser)-[:FOLLOWS]->(:TwitterUser)-
      [:FOLLOWS]->(:TwitterUser)-[:FOLLOWS]->(r:RelevantTwitterUser)
WITH r, count(*) AS numFollowers3rdLevel
RETURN r.userId, numFollowers3rdLevel
ORDER BY numFollowers3rdLevel DESC LIMIT 5;
```

```
SELECT u.userId, count(*) AS numFollowers3rdLevel
FROM FollowRel AS r1, FollowRel AS r2, FollowRel AS r3,
      RelevantTwitterUser AS u
WHERE r1.userId2 = r2.userId1 AND r2.userId2 = r3.userId1
      AND r3.userId2 = u.userId
GROUP BY u.userId ORDER BY numFollowers3rdLevel DESC LIMIT 5;
```

```
+---------------------------------+
| r.userId | numFollowers3rdLevel |
+---------------------------------+
| 8627932  | 57228                |
| 14919552 | 49615                |
| 9206682  | 47068                |
| 4029671  | 46780                |
| 3233231  | 44069                |
+---------------------------------+
5 rows
```

Una alternativa a esta última consulta en Cypher sería especificar la longitud del camino que se permite, en vez de describir de manera explícita todo el camino:

```
MATCH (:TwitterUser)-[:FOLLOWS*3]->(r:RelevantTwitterUser)
WITH r, count(*) AS numFollowers3rdLevel
RETURN r.userId, numFollowers3rdLevel
ORDER BY numFollowers3rdLevel DESC LIMIT 5;
```

Como se puede apreciar, la sintaxis de Cypher es directa e intuitiva: el hecho de que la representación en forma de grafo sea muy próxima al modelo mental que tenemos de los datos hace que podamos expresar lo que queremos obtener de un modo natural. En cambio, aunque la consulta SQL no es mucho más compleja, sí que requiere pensar en la representación subyacente de los datos para construirla. Además, al ir aumentando la distancia permitida entre los usuarios, la consulta en SQL crece (en número de tablas involucradas en el cálculo de la solución y en número de *joins*). En cambio, la consulta en Cypher solo requiere modificar el tamaño del camino permitido.

Los usuarios relevantes con más seguidores son importantes porque sus mensajes son leídos, potencialmente, por más gente. Por lo tanto, GaggleCo estará interesada en que estos usuarios hablen de su plataforma, ya que esta información llegará así a más usuarios. Existen otras medidas de importancia sobre nodos en una red que nos permiten cuantificar cuáles son los usuarios más bien conectados. En concreto, puede ser interesante saber cuáles son los usuarios más próximos al resto de los usuarios de la red o cuáles tienen mayor poder de control sobre la información que fluye por la red. En teoría de grafos, se definen las métricas centralidad de proximidad y centralidad de intermediación, que permiten responder, respectivamente, a las dos cuestiones planteadas. Ambas métricas requieren calcular el camino más corto entre dos nodos. Veamos cómo podemos hacer este cálculo usando Cypher:

```
MATCH p = shortestPath( (u1:RelevantTwitterUser{userId:216130820})
      -[r:FOLLOWS*]->(u2:RelevantTwitterUser{userId:214103395}) )
WITH EXTRACT(n IN NODES(p) |n.userId) AS nodes
RETURN nodes
```

```
[216130820,113061647,457733,214103395]
```

Cypher también nos permite expresar de manera sencilla el cálculo del camino más corto desde un usuario relevante de Twitter hacia el resto de los usuarios relevantes:

```
MATCH p = shortestPath( (u1:RelevantTwitterUser{userId:216130820})
      -[r:FOLLOWS*]->( u2:RelevantTwitterUser ) )
WHERE u1 <> u2
WITH EXTRACT(n IN NODES(p) |n.userId) AS nodes
RETURN nodes
```

```
+-------------------------------------------------+
| nodes                                           |
+-------------------------------------------------+
| [216130820,74251247,220031636]                  |
| [216130820,74251247,220072518]                  |
| [216130820,150024686,124181497,220365837]       |

...

| [216130820,113061647,457733,209004862]          |
| [216130820,74251247,209501198]                  |
| [216130820,113061647,457733,214103395]          |
+-------------------------------------------------+
499 rows
```

Calcular el camino más corto entre dos nodos usando SQL no es trivial. Posibles soluciones pasan por crear tablas adicionales, utilizar evaluación recursiva de consultas o utilizar procedimientos almacenados.

6.2.4. Segmentación de comportamiento

El tercer tema que hay que investigar con los datos recogidos va enfocado a la segmentación del comportamiento de los usuarios en la red. En este sentido, es interesante detectar usuarios con comportamientos similares.

Por ejemplo, puede resultar útil saber si dos usuarios relevantes siguen a los mismos usuarios de Twitter. Esto indica que están interesados en la misma información. La siguiente consulta permite calcular el número de usuarios comunes que siguen cada par de usuarios relevantes de Twitter, mostrando los cinco resultados con mayor número de usuarios comunes.

```
MATCH (u1:RelevantTwitterUser)-[:FOLLOWS]->(:TwitterUser)
      <-[:FOLLOWS]-(u2:RelevantTwitterUser)
RETURN u1.userId, u2.userId, count(*) AS numCommonFriends
ORDER BY numCommonFriends DESC LIMIT 5;
```

```
SELECT r1.userId1, r2.userId1, count(*) AS numCommonFriends
FROM FollowRel AS r1, FollowRel AS r2, RelevantTwitterUser AS u1,
```

```
      RelevantTwitterUser AS u2
WHERE r1.userId2 = r2.userId2 AND r1.userId1= u1.userId
      AND r2.userId1=u2.userId AND u1.userId != u2.userId
GROUP BY r1.userId1, r2.userId1
ORDER BY numCommonFriends DESC LIMIT 5;
```

```
+-----------------------------------------+
| u1.userId | u2.userId | numCommonFriends |
+-----------------------------------------+
| 44992826  | 6790852   | 2277             |
| 6790852   | 44992826  | 2277             |
| 12925072  | 8039442   | 2075             |
| 8039442   | 12925072  | 2075             |
| 17800797  | 16334724  | 1658             |
+-----------------------------------------+
5 rows
```

De una manera similar, se puede identificar a los usuarios que hacen retuits de los mismos tuits. Esto permite clasificar a los usuarios en función del contenido que encuentran interesante (o, al menos, digno de reenviar). La siguiente consulta permite calcular el número de retuits sobre un mismo tuit que dos usuarios concretos han realizado.

```
MATCH (:RelevantTwitterUser{userId:44992826})-[:HAS_WRITEN]->(:Retweet)
      -[:IS_A_RETWEET_OF]->(t:Tweet)<-[:IS_A_RETWEET_OF]
      -(:Retweet)<-[:HAS_WRITEN]-(:RelevantTwitterUser{userId:6790852})
RETURN count(distinct(t)) AS numCommonRetweets ;
```

```
SELECT count(distinct(t1.retweetedStatusId))
FROM Tweet AS t1, Tweet AS t2
WHERE t1.userId = 44992826 AND t1.retweetedStatusId IS NOT NULL AND
      t2.userId = 6790852 AND t2.retweetedStatusId IS NOT NULL AND
      t1.retweetedStatusId = t2.retweetedStatusId;
```

```
0
```

7. Anexo A: Guía básica de instalación y uso de neo4j

7.1. Instalación de neo4j

Neo4j está disponible para múltiples plataformas. Este anexo recoge las instrucciones básicas de instalación para Linux, Windows y MacOSX. El manual de neo4j, así como la página de descargas (http://www.neo4j.org/download), contienen información detallada sobre las diferentes alternativas de instalación.

7.1.1. Linux

Existe un repositorio de paquetes que incluye neo4j para distribuciones basadas en *debian*. Las siguientes instrucciones permiten instalar neo4j desde el repositorio.

```
# Start root shell
sudo -s
# Import signing key
wget -O - http://debian.neo4j.org/neotechnology.gpg.key |apt-key add -
# Create an Apt sources.list file
echo 'deb http://debian.neo4j.org/repo stable/' >
      /etc/apt/sources.list.d/neo4j.list
# Find out about the files in the repository
apt-get update
# Install Neo4j, community edition
apt-get install neo4j
# Start neo4j server
neo4j start
```

Aparte de usar gestores de paquetes para instalar neo4j, también existe la alternativa de descargar los binarios de la última versión desde la página oficial de descargas. En este caso, se pueden utilizar las siguientes instrucciones para instalar neo4j (sustituyendo X por la versión deseada).

```
# Start root shell
sudo -s
# Download tarball
wget http://download.neo4j.org/artifact?edition=community&version=X
      &distribution=tarball
# Extract the contents
tar -cf neo4j-community-X-unix.tar.gz
# Change directory to the top-level extracted directory
cd neo4j-community-X-unix
# Alternative 1: Launch the neo4j-console
./bin/neo4j console
# Alternative 2: Install neo4j as a linux service and start the service
sudo ./bin/neo4j-installer install
sudo service neo4j-service start
```

7.1.2. Windows

Para instalar neo4j en entornos Windows, se puede descargar un instalador oficial desde la página de descargas. Una vez descargado, es necesario hacer doble clic sobre el ejecutable y seguir las instrucciones del instalador.

Alternativamente, también se pueden descargar los binarios para Windows. En este caso, desde la misma página de descargas, es necesario bajarse el fichero comprimido .zip con los binarios y extraer el contenido del fichero. Después, existen dos alternativas: utilizar la aplicación de consola de neo4j o instalar neo4j como un servicio. Para la primera alternativa, ejecutaremos el fichero `\bin\Neo4j.bat`, que se encuentra en la carpeta que hemos descomprimido. Si optamos por la segunda alternativa, será necesario abrir una consola con permisos de administrador y ejecutar `\bin\Neo4jInstaller.bat install`.

7.1.3. MAC OSX

Se puede instalar neo4j en MAC OSX a través de Homebrew, lanzando el siguiente comando.

```
brew install neo4j && neo4j start
```

Después de la instalación, se puede ejecutar neo4j como un servicio o desde una consola. Encontraréis información adicional sobre la instalación de neo4j en sistemas MAC OSX en la página de descargas.

7.2. Uso de neo4j

Existen dos herramientas básicas que permiten interactuar con un servidor de neo4j: `neo4j-browser` y `neo4j-shell`.

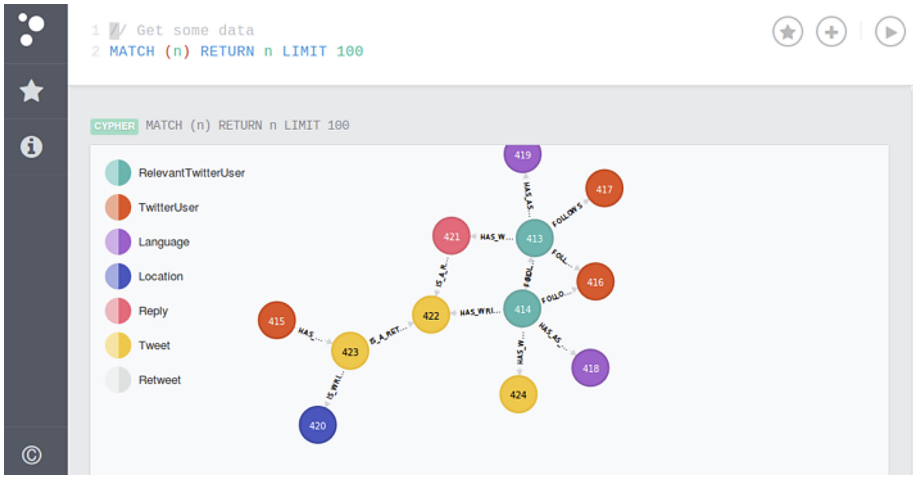
7.2.1. El navegador de neo4j

Neo4j instalado como servicio nos ofrece una interfaz web para interactuar con el servidor. Los parámetros por omisión permiten acceder al servidor a través del puerto 7474 (`http://localhost:7474/`).

La interfaz web de neo4j ofrece una interfaz gráfica para interactuar con el servidor de base de datos, además de una herramienta de visualización que nos puede ayudar a interpretar el grafo almacenado. Por su facilidad de uso y por su contenido, la interfaz web es una herramienta ideal para aprender a utilizar neo4j.

La figura 7 muestra la interfaz web de neo4j. Como se puede apreciar, el navegador nos permite ejecutar consultas con Cypher, introduciéndolas en el recuadro superior. Podemos elegir entre obtener como resultado de la consulta los datos en formato tabular o una visualización gráfica del grafo resultante. En la imagen podemos ver el resultado de esta última opción. Además, el grafo obtenido como resultado de la consulta es interactivo, de manera que podemos tanto mover los nodos para lograr una visualización más adecuada a nuestros propósitos como hacer clic sobre los nodos o las relaciones para obtener información sobre las propiedades que contienen y sus valores.

Figura 7. Interfaz web de neo4j



El navegador nos permite también almacenar consultas (usando el icono con una estrella justo a la derecha del recuadro para introducirlas), de manera que las podamos volver a ejecutar en el futuro de manera rápida.

El menú lateral ofrece múltiples funcionalidades adicionales. Haciendo clic sobre el logotipo de neo4j, nos muestra las etiquetas de los nodos y los tipos de relaciones que existen en el grafo, así como las propiedades que aparecen en los diversos tipos de nodos y relaciones. Además, podemos ver el *path* de los ficheros de la base de datos en el sistema y el tamaño que estos ocupan. Siguiendo el icono de la estrella podemos acceder a las consultas guardadas anteriormente, así como a un conjunto de consultas básicas que vienen incorporadas. Por último, el icono de información nos muestra un conjunto de referencias básico para trabajar con neo4j.

7.2.2. La consola de neo4j

Neo4j tiene también una consola que permite lanzar consultas con Cypher sobre una base de datos tanto local como remota. El puerto por omisión es el 1337 y podemos acceder a la consola con el comando `neo4j-shell`. El comando admite algunos parámetros para configurar detalles de la ejecución. El conjunto específico de parámetros depende de si la conexión es o no remota.

Para una conexión a un servidor local, `neo4j-shell` admite:

`-path PATH`: indica la localización de la base de datos.

`-pid PID`: especifica el PID del proceso al que conectarse.

`-readonly`: acceso a la base de datos en modo de solo lectura.

`-c COMMAND`: indica el comando a ejecutar. Después de la ejecución, se cierra la consola.

`-file FILE`: detalla el fichero a leer y ejecutar. Después de la ejecución, se cierra la consola.

`-config CONFIG`: indica el fichero de configuración que usar.

Para una conexión a un servidor remoto, `neo4j-shell` admite:

`-port PORT`: indica el puerto de conexión.

`-host HOST`: especifica el *host* de conexión.

`-name NAME`: indica el nombre RMI.

`-readonly`: acceso a la base de datos en modo de solo lectura.

Algunas de las opciones ofrecidas por la consola son de especial interés. Por ejemplo, podemos abrir una consola en modo de solo lectura (`-readonly`), de manera que nos aseguramos de que no se harán modificaciones del grafo. Este modo puede ser muy útil para realizar pruebas o durante la fase de aprendizaje de neo4j. La opción de especificar el comando que ejecutar (`-c COMMAND`) se puede utilizar para lanzar tareas en segundo plano que ejecuten alguna consulta en Cypher y luego finalicen la consola. A veces también se utiliza para ejecutar consultas que tengan una salida muy larga o para importar un volcado de alguna otra base de datos. Para esta última tarea, también se utiliza la opción de lectura de fichero (`-file FILE`), que permite leer y ejecutar las instrucciones de un fichero de manera más rápida que utilizando tuberías (*pipes*) para redireccionar la entrada.

Además de permitir ejecutar sentencias de Cypher, la consola de neo4j nos permite navegar por el grafo con una sintaxis similar a los comandos de exploración del sistema de archivos de la consola de unix.

Al iniciar la consola, no nos encontramos situados en ninguna entidad del grafo, pero podemos utilizar el comando `cd` para posicionarnos y empezar a recorrer el grafo. Algunos de los usos del comando `cd` son los siguientes:

`cd <node-id>`: atraviesa una arista hasta el nodo con el identificador indicado. El nodo debe estar a distancia 1 del nodo actual.

`cd -a <node-id>`: realiza un cambio de *path* absoluto, es decir, nos permite situarnos en el nodo indicado independientemente de la posición actual.

`cd -r <relationship-id>`: permite moverse a una arista en vez de a un nodo. La arista debe ser incidente al nodo actual.

`cd -ar <relationship-id>`: permite moverse a una arista haciendo un cambio de *path* absoluto, es decir, permite situarnos en la arista sin que el nodo actual sea incidente a esta.

`cd ..`: realiza un paso atrás en la navegación, es decir, nos sitúa en la localización anterior.

En cualquier momento, podemos usar el comando `pwd` para consultar el *path* actual.

El comando `ls` permite mostrar el contenido del nodo o la relación actual. El comando admite también argumentos que permiten filtrar la salida (`-f`), listar únicamente las propiedades (`-p`) o las relaciones (`-r`), mostrar un breve resumen del contenido (`-b`), entre otros. El comando `man` permite obtener el listado de todas las funciones de un comando concreto (por ejemplo, `man ls` nos muestra las opciones del comando `ls`).

El comando `schema` muestra una lista de todos los índices y las restricciones de integridad creadas sobre el grafo.

Esto es solamente un pequeño resumen de las funcionalidades que nos ofrece `neo4j-shell`. Para una revisión más completa de las funcionalidades, podéis consultar el manual de neo4j.

8. Anexo B: Sintaxis de Cypher

Cypher es un lenguaje declarativo que permite realizar consultas sobre un grafo almacenado en una base de datos neo4j. Las consultas en Cypher son fáciles de interpretar por los humanos, ya que la sintaxis del lenguaje es muy visual (está basada en ASCII art). Así, los nodos se representan con paréntesis `()` y las aristas como flechas entre los nodos, `()-[]->()`. A la vez que simple y visual, Cypher permite expresar consultas complejas sobre el grafo, siempre minimizando la complejidad de expresión de estas.

Al ser un lenguaje declarativo, Cypher está centrado en expresar lo que se quiere obtener del grafo (y no intenta indicar cómo obtenerlo). Cypher está inspirado en otros lenguajes de consulta de bases de datos ya existentes. Así, encontraremos palabras clave incorporadas de SQL como, por ejemplo, `WHERE`, `ORDER BY`, `LIMIT` y `COUNT`.

Del mismo modo, la estructura de una consulta en Cypher también deriva de la estructura de una consulta SQL. Las consultas en Cypher están formadas por cláusulas, que se pueden encadenar.

En este anexo, repasaremos la sintaxis básica de Cypher. Para un resumen esquemático de la sintaxis de Cypher, se puede consultar Neo4j Cypher Refcard 2.0*.

***http://docs.neo4j.org /refcard/2.0/**

8.1. Consultas de lectura

La sintaxis general de una consulta de lectura en Cypher es la siguiente:

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

La cláusula `MATCH` puede utilizarse para seleccionar el patrón y las restricciones sobre este:

```
// Los patrones permiten expresar nodos y relaciones:
MATCH ()-[]->()

// Los patrones pueden contener etiquetas y propiedades:
MATCH (:RelevantTwitterUser{userName:'Alice'})-[:FOLLOWS]->()
```

```
// Los patrones pueden contener identificadores que permiten
// filtrar las propiedades y retornar los valores:
MATCH (u1:RelevantTwitterUser)
WHERE u1.userName = 'Alice'

// Podemos permitir relaciones de cualquier tipo:
MATCH ()-->()

// Podemos permitir relaciones de cualquier tipo
// y en cualquier dirección:
MATCH ()--()

// Se pueden incluir operaciones de OR lógico
// sobre las propiedades y etiquetas:
MATCH ()<-[:IS_A_RETWEET_OF|:IS_A_REPLY_OF]-()

// Podemos expresar relaciones un poco más complejas:
MATCH (:RelevantTwitterUser)-[:HAS_WRITEN]->(:Tweet)<-
      [:IS_A_REPLY_OF]-(:Reply)

// Se pueden expresar caminos de longitud variable:
MATCH (:TwitterUser)-[r:FOLLOWS*1..3]->(:TwitterUser)

// Podemos seleccionar un camino más corto entre dos nodos:
MATCH (u1:TwitterUser{userId:1}),(u2:TwitterUser{userId:2}),
      p = shortestPath((u1)-[*..15]-(u2))

// Usando un patrón opcional, se asignan NULL
// a las partes no encontradas:
OPTIONAL MATCH (u1)-[r]->(u2)
```

Un `WHERE` puede formar parte de las cláusulas `MATCH`, `OPTIONAL MATCH`, `START` y `WITH` y permite especificar las restricciones (usada con `MATCH` y `OPTIONAL MATCH`) o filtrar los resultados (`START` y `WITH`):

```
// Filtrar por etiqueta:
WHERE u:RelevantTwitterUser

// Filtrar por propiedad:
WHERE u.statusesCount > 300

// Utilizar expresiones regulares:
WHERE u.userName =~ 'Alic.*'

// Utilizar expresiones regulares sin distinguir entre
// mayúsculas y minúsculas:
WHERE u.userName =~ '(?i)ali.*'

// Admite operaciones booleanas:
WHERE u.screenName = 'Neal' XOR
      (u.statusesCount < 10 AND u.name = 'Stephenson')

// Filtrar por el tipo de relación:
WHERE type(r)=~ 'IS_A_.*'

// Filtrar dependiendo de la existencia de una propiedad:
WHERE HAS (u.userName)

// Comprobar si un elemento existe en una colección:
WHERE u.screenName IN ["Alice", "Bob"]

// Utilizar NOT para excluir patrones:
MATCH (u:TwitterUser)
WHERE NOT (u)-[:HAS_WRITEN]->(:Tweet)
```

Las propiedades no existentes en un nodo o relación evalúan a NULL y las comprobaciones de igualdad sobre la propiedad evalúan a falso. El siguiente ejemplo demuestra el comportamiento descrito:

```
CREATE
( alice:RelevantTwitterUser:TwitterUser{ userId:10 ,
      userName:'Alice1990' , screenName:'Alice' ,
      location:'Wonderland' , createdAt:20060705 ,
      verified:True , geoEnabled:False , statusesCount:52 } ),
( bob:TwitterUser{ userId:11 } );
```

```
MATCH (u:TwitterUser)
WHERE u.geoEnabled = False
RETURN u;
```

```
Node[3755
{statusesCount:52,geoEnabled:false,location:"Wonderland",
screenName:"Alice",createdAt:20060705,userId:10,verified:true,
userName:"Alice1990"}
1 row
```

```
MATCH (u:TwitterUser)
WHERE u.geoEnabled = False OR u.geoEnabled IS NULL
RETURN u;
```

```
Node[3755]{statusesCount:52,geoEnabled:false,location:"Wonderland",
screenName:"Alice",createdAt:20060705,userId:10,verified:true,
userName:"Alice1990"}

Node[3756]{userId:11}
2 rows
```

La cláusula `RETURN` permite especificar el resultado a devolver:

```
// Retorna el valor de todos los identificadores:
RETURN *

// Utiliza un alias:
RETURN u AS SelectedTwitterUser

// Retorna valores únicos:
RETURN DISTINCT u

// Ordena el resultado por la propiedad userName:
ORDER BY u.userName

// Ordena el resultado por la propiedad userName de forma
// descendente:
ORDER BY u.userName DESC

// Omite el primer resultado:
SKIP 1

// Limita el número de resultados:
LIMIT 1
```

La cláusula `RETURN` admite funciones de agregación:

```
// Devuelve el número de coincidencias:
RETURN count(*)

// Agrupa el resultado utilizando la clave n y cuenta el número
// de coincidencias de cada valor de la clave
```

```
RETURN n, count(*)

// Retorna la suma de la propiedad especificada:
RETURN sum(u.statusesCount)

// Retorna la media de la propiedad especificada:
RETURN avg(u.statusesCount)

// Se pueden calcular los valores máximos y mínimos de una propiedad:
RETURN min(u.statusesCount), max(u.statusesCount);

// Calcula el percentil indicado sobre una propiedad
// (el percentil se indica con valores entre 0 y 1):
RETURN percentileDisc(u.statusesCount, 0.25)

// Crea una lista con todos los valores (ignora los NULL):
return collect(u.userName);
```

8.2. Consultas de creación o modificación

La sintaxis general de una consulta de creación o modificación del grafo es:

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

Se pueden crear o borrar nodos y relaciones usando `CREATE` y `DELETE`, y asignar valores a las propiedades y añadir o eliminar etiquetas con `SET` y `REMOVE`. Veamos algunos ejemplos:

```
// Crear un nodo con una etiqueta y una propiedad:
CREATE ( bob:TwitterUser{ userId:11 } );

// Crear una relación entre dos nodos existentes:
MATCH (u1:TwitterUser{userId:1}), (u2:TwitterUser{userId:2})
CREATE (u1)-[:FOLLOWS]->(u2);

// Eliminar un nodo:
MATCH (u1:Tweet{tweetId:1})
DELETE u1;

// Crear una propiedad:
MATCH (u:TwitterUser{userId:1})
SET u.selected = True;

// Eliminar una propiedad:
MATCH (u:RelevantTwitterUser{userId:2})
SET n.screenName = NULL;

MATCH (u:RelevantTwitterUser{userId:2})
REMOVE n.screenName

// Eliminar una etiqueta:
MATCH (t:Reply{tweetId:1})
REMOVE t:Reply;

// Marcamos todos los nodos que se encuentran en el camino
// entre los nodos seleccionados
MATCH p =(u1:TwitterUser{userId:3})-[*]-(u2:TwitterUser{userId:4})
FOREACH (n IN nodes(p) |SET n.marked = TRUE )
```

Resumen

En este documento se ha realizado el diseño de una base de datos para analizar la actividad de un conjunto de usuarios en Twitter.

El documento repasa las principales fases del diseño de una base de datos: la descripción del contexto que engloba el diseño de la base de datos, de los datos a almacenar y de las consultas que se prevén; la creación de un modelo conceptual que permita representar el dominio de interés; el análisis de las principales alternativas tecnológicas que se adecuan a las necesidades del caso estudiado; el diseño lógico de la base de datos orientada a grafos, y, por último, el diseño físico de la base de datos. A continuación, el documento analiza algunas de las preguntas a las que se pretende responder con el análisis de los datos obtenidos de la red social, detallando las consultas que permiten contestarlas tanto en Cypher (sobre neo4j) como en SQL (sobre MySQL). Esto permite realizar una pequeña comparativa de la capacidad expresiva y de la complejidad de ambos lenguajes.

Adicionalmente, se incluyen también dos anexos: una pequeña introducción al uso de neo4j, acompañada de las instrucciones básicas de instalación, y una guía de sintaxis del lenguaje Cypher.

Actividades

1. Cread una consulta en Cypher para obtener el porcentaje de tuits que son *replies*.

2. Cread una consulta en Cypher para obtener el porcentaje de tuits que están geolocalizados.

3. Cread una consulta en Cypher para obtener los 5 usuarios más activos de la red, es decir, aquellos que hayan enviado más tuits durante el periodo de recogida de datos. Mostrad el identificador del usuario y el número de tuits que han enviado.

4. Cread una consulta en Cypher que devuelva los cinco lugares (*Locations*) desde los que se han enviado más tuits, así como el número de tuits enviados desde estos lugares.

5. Cread una consulta en Cypher que permita obtener el identificador y el nombre de usuario relevante que utiliza como idioma de perfil el inglés y que ha enviado más tuits con contenido propio geolocalizados. Mostrad también el número de tuits con contenido propio geolocalizados que ha enviado este usuario.

6. Cread una consulta en Cypher que permita obtener el identificador y el nombre de usuario de los usuarios relevantes que no han enviado ningún tuit con contenido propio pero que han hecho al menos un retuit.

7. Cread una consulta en Cypher que retorne el idioma más utilizado para enviar tuits desde Barcelona. Escribid también la consulta que permita obtener el idioma más utilizado desde Madrid. Dado que nuestra base de datos no contiene el idioma de cada tuit, asumiremos que cada tuit está escrito en el idioma del perfil de su autor.

8. Cread una consulta en Cypher que retorne las fechas en las que se han escrito más tuits que la media de tuits por día. Mostrad tanto la fecha como el número de tuits enviados durante el día en cuestión.

9. Cread una consulta en Cypher para calcular con cuántos usuarios diferentes mantiene conversaciones mediante *replies* cada usuario. Mostrad los 5 usuarios que mantienen más conversaciones junto al número de estas.

10. Esta última consulta resulta bastante pesada de describir (incluso en Cypher). Proponed alguna modificación del grafo almacenado que permita optimizarla, tanto en lo que tiene que ver con la expresión de la consulta como respecto al tiempo necesario para ejecutarla. Realizad la modificación propuesta y cread la consulta en Cypher que permite aprovechar la modificación.

11. Cread una consulta en Cypher que retorne los cinco usuarios con más seguidores directos e indirectos a cualquier distancia, es decir, considerando tanto los seguidores directos como los seguidores de los seguidores, los seguidores de los seguidores de los seguidores, etc.

Nota

Esta última actividad plantea un problema similar a las dos últimas consultas que se comentaban en el subapartado 6.2.3. Así pues, será sencillo crear la consulta en Cypher que permita resolver la actividad, pero no será posible hacerlo de forma directa con SQL.

Bibliografía

Freeman, L.C. (1977). *A Set of Measures of Centrality Based on Betweenness*. USA: American Sociological Association.

Robinson, I.; Webber, J.; Eifrem, E. (2013). *Graph Databases*. USA: O'Reilly Media, Inc. (ISBN: 978-1-449-35626-2)

The Neo4j Team. *The Neo4j Manual*. [Fecha de consulta: mayo del 2014]. En línea: http://docs.neo4j.org/chunked/2.0.3/

The Neo4j Community. *Learn Cypher - the Neo4j query language*. [Fecha de consulta: mayo del 2014]. En línea: http://www.neo4j.org/learn/cypher

The Neo4j Team. *Neo4j Cypher Refcard 2.0*. [Fecha de consulta: mayo del 2014]. En línea: http://docs.neo4j.org/refcard/2.0/.

Twitter. *Twitter developers site*. [Fecha de consulta: mayo del 2014]. En línea: https://dev.twitter.com/.

Wasserman, S. (1994). *Social Network Analysis: Methods and Applications*. UK: Cambridge University Press.