E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Análisis, diseño y despliegue de una base de datos orientada a grafos para la investigación de Derivaciones de Responsabilidades

Grado en Ingeniería Informática

Trabajo Fin de Grado

Alumno: Iván Carabantes Guerrero

Director: Mikel Galar Idoate

Pamplona, 10 de junio del 2019

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen y palabras claves

La principal finalidad del proyecto es el desarrollo de un sistema de investigación de Derivaciones de Responsabilidades para el departamento de Gestión de Deudas de la empresa Tracasa Instrumental S.L que colabora con la Hacienda Tributaria de Navarra (HTN).

Actualmente, el proceso de investigación es manual y costoso y carece de una herramienta que lo facilite. El proceso se basa en la investigación de posibles sujetos de una derivación de responsabilidades, por lo que se necesita plasmar las relaciones (jurídicas, familiares...) entre las personas (físicas, jurídicas) junto con los indicadores de capacidad financiera de dichas personas.

Asimismo, para satisfacer esta necesidad se ha planteado el despliegue de una base de datos orientada a grafos. Esto implica elaborar una prospección de este tipo de bases de datos en el mercado, confeccionar un análisis y diseño nuevo del modelo de datos de Hacienda y originar un planteamiento del suministro de datos, entre otras muchas tareas.

Palabras claves: Base de datos orientada a grafos, Grafo, Neo4j, Cypher, Derivaciones de Responsabilidades.

Summary and keywords

The main purpose of this project is the development of an investigation system of Responsibility Derivations that will be used by the Debts Management Department of the company Tracasa Instrumental S.L, which collaborates with the Navarre Tax Treasury (Hacienda Tributaria de Navarra, HTN).

Currently, the research process is not very comfortable, since it is manual, costly and lacks a tool to facilitate it. The process is based on the investigation of possible subjects of a derivation responsibility, so it is necessary to get the relationships (legal, family…) between different people (natural, legal) in conjuction with the financial capacity of such people.

In this way, to meet this need, the deployment of a Graph Database has been considered. This implies elaborating a prospection of this type of databases on the market, elaborating an analysis and a new design of the data model and making a data supply plan, among other more tasks.

Keywords: Graph Database, Graph, Neo4j, Cypher, Responsibility Derivations.

Laburpena eta hitz gakoak

Proiektu honen helburu nagusia Ardura Desbideraketaren prozedura errazteko sistema baten garapena da, Tracasa Instrumental enpresako Zorra Kudeaketa Sailaren eta Nafarroako Zerga Ogasunaren lankidetzarekin.

Gaur egun, ikerketa hauetan erabiltzen dituzten prozesuak eta teknikak ez dira batere erosoak. Gainera oso prozesu neketsuak eta konplexuak dira eta ez dira inolako tresna espezifikoak erabiltzen eginbide hau errazteko. Aipatutako ikerketa, zorra baten erantzulearen bilaketan oinarritzen da, horretarako beharrezkoak dira pertsona juridikoen eta fisikoen arteko harremanak eta pertsona hauen finantza-ahalmenak.

Era berean, premia hori asetzeko, grafoen datu-baseak erabiltzea erabaki da. Guzti honek, merkatuko datu-baseen prospekzioa, datuen azterketa eta diseinua eta datu inportazio plana sortzea dakartza.

Hitz gakoak: Grafoen datu-baseak, Grafoak, Neo4j, Cypher, Ardura Deribazioak.

Agradecimientos

Agradezco toda la energía y el apoyo recibido por parte de mis padres, mi hermano y familia durante esta etapa académica.

A mi tutor Mikel Galar por involucrarse y estar siempre a mi disposición y por guiarme en este Trabajo Fin de Grado.

A mis amigos y compañeros de clase por amenizarme esta etapa y por estar tanto en los momentos buenos como en los malos.

Índice

Capítulo 1 - Introducción

En el presente documento se abarca una descripción detallada del Trabajo Fin de Grado realizado, cuyo principal objetivo ha sido el análisis, diseño y despliegue de una base de datos orientada a grafos, que facilita la investigación de Derivaciones de Responsabilidades realizada por el departamento de Gestión de Deudas en colaboración con la Hacienda Tributaría de Navarra (HTN). Este trabajo se ha llevado a cabo durante las prácticas en la empresa Tracasa Instrumental S.L.

Tracasa Instrumental S.L es una empresa pública del Gobierno de Navarra perteneciente a la categoría CNAE 8411 - Actividades generales de la Administración Pública. Uno de los servicios que Tracasa Instrumental presta a la Hacienda Tributaria de Navarra (HTN) es la colaboración con la Sección de Procedimientos Especiales del Servicio de Recaudación en la investigación y gestión de las Derivaciones de Responsabilidad. Concretamente, es el departamento de Gestión de Deudas *GEDESA* el que presta estos servicios.

1.1 - Motivación , necesidades y antecedentes

Este proyecto surge tras un autodiagnóstico de la situación respecto a las herramientas y los procedimientos aplicados en las investigaciones de Derivaciones de Responsabilidad. En él surge la necesidad de una herramienta que facilite todo este proceso. El proceso al cual se ven sometidos los empleados, es un proceso extenso, laborioso y manual. Sin embargo, a pesar de ello, los resultados de recaudación obtenidos por esta vía son excelentes. Además, la investigación de Derivaciones de Responsabilidades se considera un método altamente eficaz para la lucha contra el fraude fiscal.

Al carecer de una herramienta específica de investigación, los empleados se encuentran con los siguientes inconvenientes:

- Están obligados a investigar varias personas físicas o jurídicas para finalmente determinar la responsabilidad oportuna.
- La investigación es compleja y extensa (cantidad abundante de NIFs a revisar).
- Por el trabajo que genera y el grado de dispersión de la información, en ocasiones no se utiliza toda la información que pudiera existir.
- La información se consulta de manera individual por NIF y tipo de hecho tributario.

Es por todo esto que surge este Trabajo Fin de Grado, en el que se plantea desplegar una base de datos orientada a grafos con los datos de la Hacienda Tributaria de Navarra para cubrir todas las necesidades mencionadas.

Con este Trabajo Fin de Grado, lo primero que se quiere conseguir es una visualización gráfica de los datos y de las relaciones existentes que se puedan dar (familiares, socios, administradores, etc.). Este hecho supone un gran avance en la investigación de derivaciones, ya que las derivaciones y las relaciones a estudiar pueden verse fácilmente como grafos. La visualización en un grafo de estas relaciones es más directa que la visualización en tablas relacionales.

Por otra parte, se quiere centralizar toda la formación en un solo grafo. Es decir, se quiere combatir el grado de dispersión de la información disponible. De esta manera, los empleados solo tendrán que acudir a una única fuente de información.

De igual forma, se pretende automatizar lo máximo posible todo el proceso de investigación. Actualmente, se necesitan realizar múltiples pasos para obtener un caso de uso, sin embargo, con la base de datos orientada a grafos se quiere minimizar el número de pasos que se debe dar.

Finalmente, mediante esta tecnología se quiere obtener un mejor rendimiento en las búsquedas que necesitan realizar.

Asimismo, antes de adentrarse en los objetivos específicos del Trabajo Fin de Grado se cree oportuno aclarar ciertas definiciones y procesos del área tributaria.

Las derivaciones de responsabilidad tributaria quedan justificadas en el siguiente artículo extraído del documento ofrecido por la empresa:

La obligación fundamental de todo contribuyente consiste en el pago de la deuda tributaria. Y aunque el deudor principal o su sustituto van a ser los obligados en primer lugar; si no pagan, la Hacienda Pública podrá derivar su acción de cobro hacia otras personas declarándolas responsables de esa deuda.

En los estudios de dichas derivaciones, las tareas de obtención de sociedad sucesora, administradores y/o socios y sus bienes adquiere una importancia máxima, ya que, si procede, será a esas personas a las que se les derive la deuda correspondiente.

1.2 – Objetivos particulares

La estructura de este Trabajo Fin de Grado se divide en diferentes capítulos que reflejan los objetivos particulares que se han seguido para finalmente llegar al objetivo principal ya mencionado.

En el *Capítulo 2* se describen dos de los conceptos claves de este proyecto (grafos y base de datos orientada a grafos), junto con las bases de datos relacionales para posteriormente poder realizar una comparativa entre estas dos.

Seguidamente, en el *Capítulo 3*, se realiza una prospección exhaustiva de las bases de datos orientadas a grafos. En este capítulo se destacan las diferentes ventajas y

desventajas que supone la utilización de este tipo de base de datos en el proyecto, se realiza una búsqueda de las posibles referencias del mercado actual, se determinan las características claves para llevar a cabo la selección de la base de datos y finalmente se escoge la que mejor se adecua a los casos de uso.

Una vez llegado al *Capítulo 4*, se conoce la base de datos a utilizar, por lo que se procede a realizar el diseño y el modelo orientado a grafos de los datos. En esta etapa se incluye la generación de datos sintéticos generados aleatoriamente por dos motivos fundamentales. Por una parte, los datos provienen de la Hacienda Tributaria de Navarra, por lo que se consideran datos sensibles y confidenciales, y sería una infracción grave la utilización de éstos para el Trabajo fin de Grado. Por otra parte, la obtención de estos datos reales no es un proceso rápido e inmediato, se debe seguir un riguroso protocolo y se deben firmar diferentes contratos para cumplir con el reglamento de protección de datos, que retrasan la entrega de los mismos.

Después de la generación de los datos, en el *Capítulo 5* se realiza la importación de estos a la base de datos junto con la especificación de las diferentes técnicas utilizadas para optimizar el proceso. Tanto para los datos reales como para los datos sintéticos se sigue el mismo procedimiento.

Asimismo, una vez se tiene todos los datos importados, en el *Capítulo 6* se explota la base de datos orientada a grafos para cumplir con todos los objetivos, necesidades y supuestos planteados.

Para finalizar el Trabajo Fin de Grado, en el *Capítulo 7*, se determinan las diferentes conclusiones a las que se han llegado junto con las líneas futuras proyectadas.

Capítulo 2 – Preliminares

El objetivo principal de este capítulo es la aclaración y la descripción de conceptos claves empleados durante el desarrollo del proyecto. De esta manera se podrá hacer referencia a los mismos de una manera más accesible, comprensible y diáfana.

2.1 - La estructura Grafo

El comienzo de la teoría de grafos se sitúa en el año 1736 con el trabajo de Leonhard Euler en el cual se presenta el planteamiento del afamado problema de los puentes de Königsberg. Este problema consiste en hallar un camino que recorra los siete puentes cruzando una única vez por cada uno de ellos [1].

Este primer acercamiento a la teoría de grafos impulsa a otros físicos y matemáticos a indagar en la nueva filosofía de grafos. En 1847 Gustav Kirchhoff incorporó la teoría de grafos en el análisis de las redes eléctricas, estableciendo las leyes de Kirchhoff. Esta unión se considera el primer asentamiento de la teoría de grafos en la ingeniería [1].

El problema de los cuatro colores que ideó Francis Guthrie en el año 1852 pero que no se solventó hasta un siglo después, significó un gran avance en la teoría de grafos, ya que se definieron conceptos teóricos imprescindibles de los grafos [1].

Si bien el grafo ha sido transcendental durante años en diferentes sectores, en los últimos años la inclinación hacia él ha sido desmedida debido al buen comportamiento y los favorables resultados obtenidos.

Los grafos han sido utilizados en la mayoría de las áreas de la ingeniería y otras ramas de la ciencia: informática, ingeniería eléctrica, telecomunicaciones, química, economía, etc. Se han empleado para representar las arquitecturas de redes e Internet, para diseñar circuitos eléctricos complejos, para encontrar el camino óptimo a un problema estadístico, para la implementación de programas de recomendaciones y procesamiento de datos, etc.

Especialmente en el área de la ingeniería informática el uso de grafos está en auge, principalmente por lo fácil que resulta explotar los datos en forma de grafo cuando estos representan conceptos fácilmente representables como grafos, pero no de otra forma. Este es el caso de este Trabajo Fin de Grado donde se parte de la necesidad de una herramienta que facilite la visualización y la explotación de las diferentes relaciones que puede poseer una persona física o jurídica con otras entidades.

2.1.1 - Definición de grafo

Un grafo se define como un par de conjuntos finitos, G = (V,A). El primer conjunto V se denomina el conjunto de vértices y no puede ser vacio. Mientras que el segundo conjunto A se conoce como el conjunto de aristas.

Cada arista se considera un par de vértices ordenados o no. Se representa mediante una línea que une la pareja de nodos, puede ser dirigida o no. Un vértice o nodo es el segundo componente del grafo que se simboliza con un círculo. Finalmente, a la secuencia de vértices unidos por aristas se denomina camino.

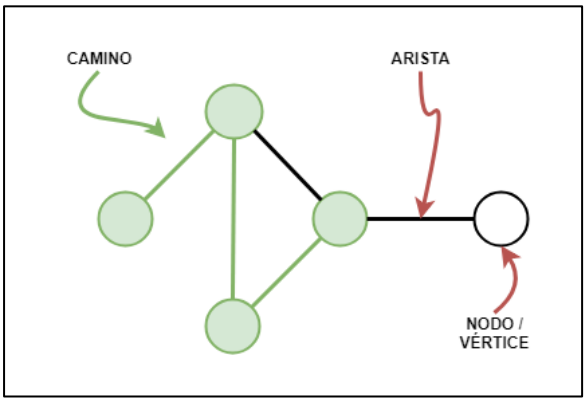


Figura 1 *- Representación de un grafo.*

Suponiendo que $G = (V, A)$ sea un grafo, se cumple:

- *V es finito y no es $\emptyset$*
- *A es finito*
- *$x, y \in V$*
- *$\{x, y\} \in A$*

Figura 2 *- Definición de grafo.*

Aristas

Se dice que dos aristas son **adyacentes,** si convergen sobre el mismo nodo. Las aristas **paralelas** (o multiples) son aquellas que comparten el mismo nodo inicial y final. Por último, las aristas **cíclicas** (o lazos) son aquellas en el que el nodo inicial y el nodo final son el mismo, véase **Figura 3**.

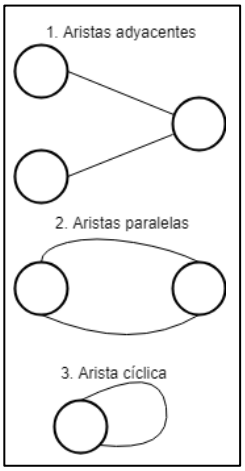


Figura 3 *- Tipo de aristas.*

Vértices

Aquel vértice que no incide con ninguna arista es un **vértice aislado**.

El **grado** o valencia de un vértice es la cantidad de aristas que inciden sobre el mismo, contando dos veces las aristas cíclicas. Siendo $v \in V$, el grado del vértice se representa con $\partial(v)$. En el caso de los grafos dirigidos, se tiene en cuenta tanto las aristas de salida como las de entrada.

Caracterización de grafos

Los grafos más importantes para este proyecto son los grafos **dirigidos** o **no dirigidos**. Los dos grafos están compuestos por un conjunto de vértices y un conjunto de aristas. Sin embargo, en el primer grafo las aristas siguen un orden y se conocen como aristas dirigidas. Mientras que en el segundo grafo los pares de vértices, las aristas no dirigidas, no siguen ningún orden.

Visualmente, los grafos dirigidos son aquellos cuyas aristas se representan mediante una flecha, ya que establecen una dirección. No obstante, las aristas de los grafos no dirigidos se consideran bidireccionales, ya que no especifican una dirección exacta. Estas últimas son lineas.

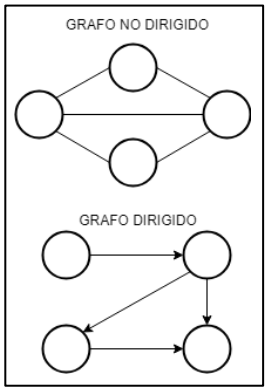


Figura 4 *- Grafo dirigido y no dirigido.*

2.2 - Bases de datos relacionales

El concepto de base de datos relacional (BDR) dio a lugar en el año 1970 por el científico informático Edgar Frank Codd. Se define como un conjunto de datos cuya estructura se establece mediante tablas relacionadas. Estas tablas están formadas por registros (líneas) y atributos (columnas) y por cada registro existe una clave que los identifica unívocamente.

Generalmente los tipos de entidades se representan mediante tablas, mientras que las instancias de los mismos y sus propiedades se modelan con registros y columnas respectivamente.

Por otro lado, se ha comentado que las tablas pueden estar relacionadas, este proceso se lleva a cabo con las claves primarias y foráneas.

La explotación y el mantenimiento de la base de datos se efectúa por medio del lenguaje de programación estándar SQL (*Structured Query Language*). Este lenguaje permite crear, leer, actualizar y borrar los datos.

Con el fin de obtener un enfoque más práctico de este tipo de modelo, se ha planteado un posible modelo relacional para la red social *Twitter*. En este ejemplo, disponemos de usuarios, los cuales deben tener una cuenta de correo vinculada. Los usuarios disponen de un *nickname* único y, además, se almacena información como su nombre, apellidos y si el usuario está activo o no. Por otro lado, respecto a las cuentas de correo, se guarda el proveedor, el nombre de la cuenta, que debe ser único, y la fecha de vinculación de la cuenta con el usuario de *Twitter.* Al mismo tiempo, un usuario puede publicar uno o más *tweet* y le puede dar a *me gusta* a uno o más tweet. De estas dos acciones se guarda la fecha y la hora del suceso. Finalmente, en la entidad *tweet* se reúne el texto del *tweet* y un atributo que sirve para definir si el tweet está disponible o no.

Una vez aclarado el escenario, se ha diseñado un modelo relacional recogido en la **Figura 5**. Este modelo consta de cinco tablas para representar tres tipos de entidades y dos relaciones. Las entidades existentes son: el usuario, el correo y el tweet. Dentro de la primera tabla de usuarios se encuentran seis atributos (columnas), siendo el primero de ellos (*id*) la clave primaria de la tabla y el último de ellos una clave foránea que vincula la tabla de correo. A su vez, la segunda tabla de tweets posee tres atributos con la primera como clave primaria. Finalmente, la tabla de correos contiene cinco atributos. Por otro lado, se encuentran las dos tablas que representan las dos relaciones entre los usuarios y los tweets: *usuario escribe un tweet* y *usuario le gusta un tweet*. En estas dos relaciones existen dos claves foráneas que sirve para vincularlas con las tablas anteriormente mencionadas.

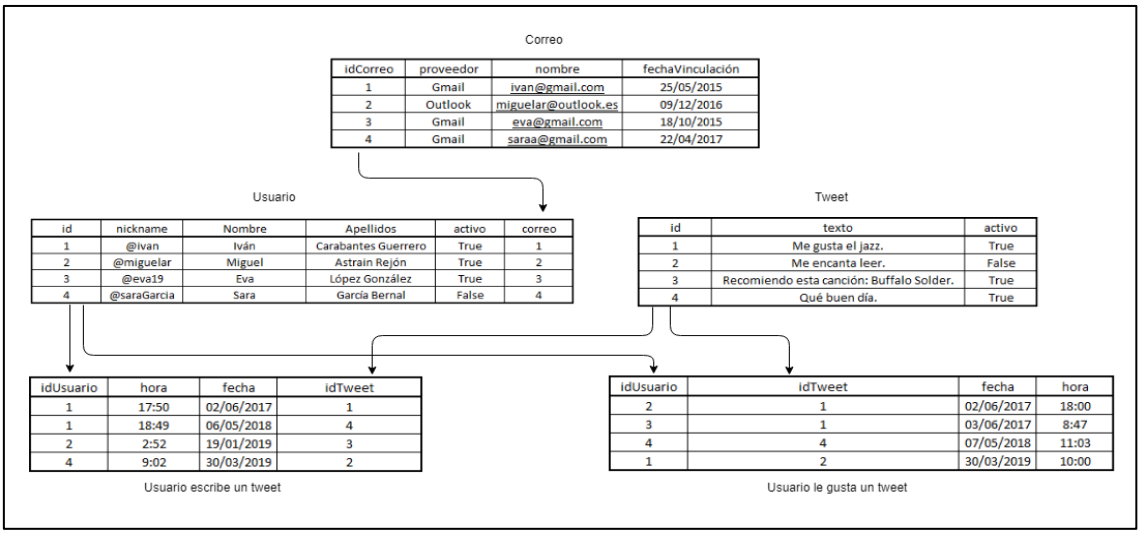


Figura 5 *- Ejemplo de un modelo relacional de Twitter.*

2.3 - Bases de datos orientadas a grafos

Si bien las bases de datos orientadas a grafos (BDG) se presentan como una tecnología novedosa, la realidad no es así. El comienzo de las BDG se sitúa a mediados del 1960 [2], cuando se desempeñaron los modelos de datos jerárquico y de red. Éstos se alejaban de la idea actual de las BDG, ya que, se necesitaba tener conocimientos muy concretos sobre el almacenamiento físico de los datos y trabajar con operaciones a muy bajo nivel [3]. Sin embargo, estos dos modelos fueron eclipsados por el modelo relacional una década después.

Posteriormente, en 1990 apareció el modelo semántico de bases de datos, en los que destacaban los estándares XML y RDF [3], que están fuertemente relacionado con el modelo de grafos.

Cabe destacar que el apogeo de las BDG comienza entre los años 2000 y 2010, con la aparición de las bases de datos orientadas a grafos con propiedades ACID y multimodales [2].

Las bases de datos orientadas a grafos (BDG) se posicionan dentro de las bases de datos NoSQL (*Not only SQL*). Estos gestores NoSQL, no utilizan *SQL* como lenguaje principal, aunque algunos soporten consultas de tipo *SQL*. Además, no necesitan un esquema o una estructura fija de los datos (libres de esquemas), permiten escalar horizontalmente y no todos garantizan las propiedades ACID [4].

Las BDG se benefician de las ventajas que supone utilizar la estructura de un grafo para el almacenamiento de los datos existentes, junto con las posibles relaciones entre los mismos.

Mientras que en las bases de datos relacionales las instancias se reflejan mediante las lineas de una tabla, en las orientadas a grafos se utilizan nodos. Las columnas que representan los atributos de las instancias en los relacionales, en este último se representan con las propiedades de los nodos. Las relaciones de entidades de un conjunto de datos se modelan mediante las aristas de un grafo.

Generalmente, este tipo de bases de datos vinculan físicamente los nodos, siendo así la propiedad más ventajosa de las mismas. Esto significa que permiten obtener las relaciones entre los diferentes conjuntos con una sola operación y de una forma rápida. A diferencia de las relacionales, las orientadas a grafos almacenan las relaciones en la propia base de datos.

En cuanto al mecanismo de almacenamiento, varía según la base de datos que se utilice. Una explicación más detallada de los diferentes mecanismos será expuesta más adelante en este documento, en posteriores capítulos.

Seguidamente, en la **Figura 6**, se puede apreciar el modelo de *Twitter* anteriormente visto para una base de datos orientada a grafos. Este modelo consta de doce nodos con

los correspondientes atributos y doce aristas con sus propiedades para representar las relaciones entre los nodos.

Como se puede apreciar, existen tres tipos de nodos: *Correo* (nodos verdes), *Usuario* (nodos azules) y *Tweet* (nodos rojos).

Cada nodo usuario tiene vinculada una cuenta de correo, la cual se representa mediante la arista verde junto con su propiedad *fecha de la vinculación*.

Asimismo, un nodo usuario puede estar relacionado con un nodo *tweet* de dos formas diferentes. Es por ello que existen dos aristas de diferentes colores (amarillas y rojas). Las aristas rojas reflejan que un cierto *usuario escribe un tweet*, mientras que las aristas amarillas reflejan que *a un usuario le gusta un tweet*.

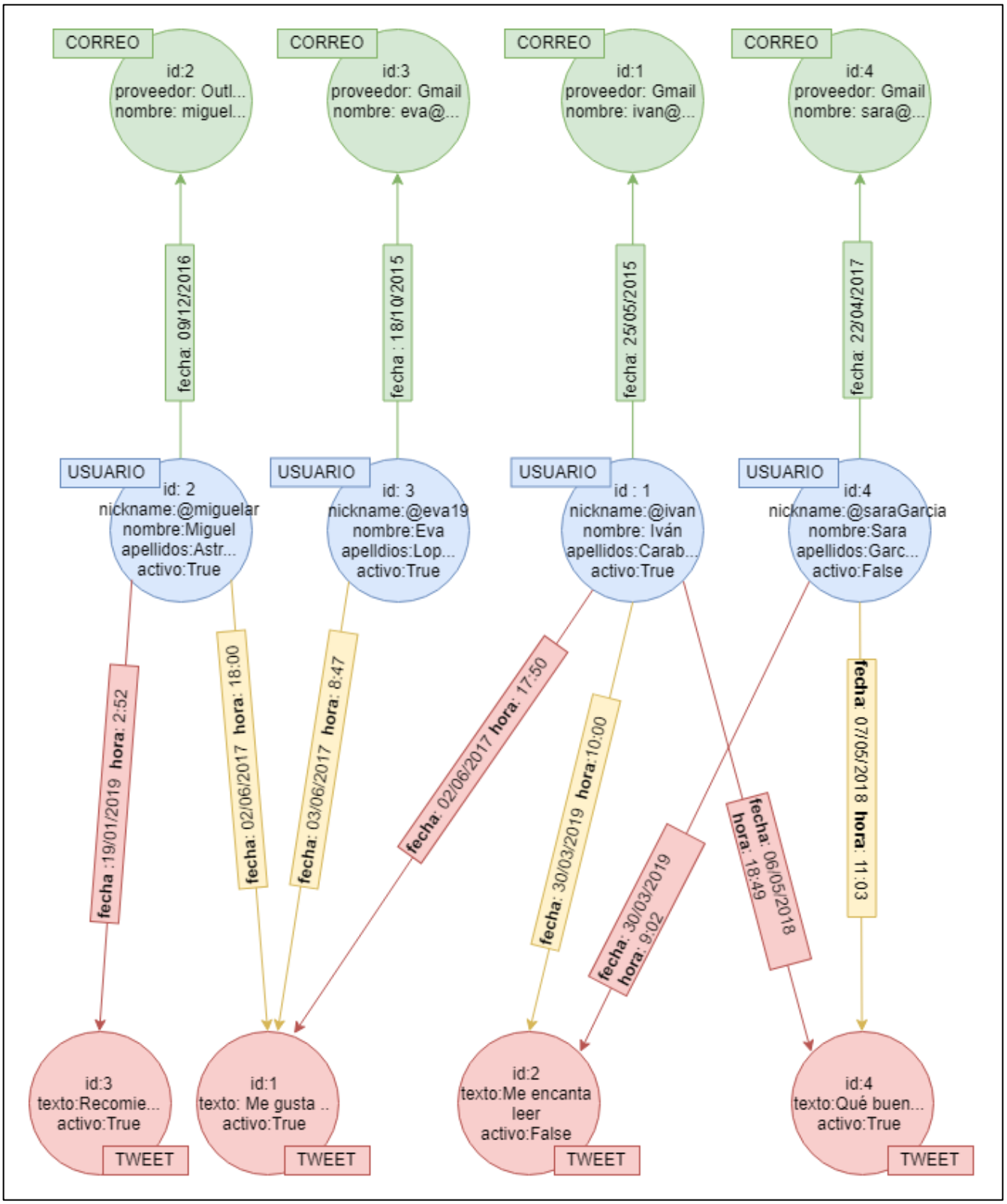


Figura 6 *- Ejemplo de un modelo orientado a grafos de Twitter.*

Capítulo 3 - Prospección de bases de datos orientadas a grafos

Este capítulo tiene como objetivo la exploración de las bases de datos orientadas a grafos, ofreciendo las ventajas y desventajas que éstas supondrían y los beneficios que traerían para este caso de uso (Derivaciones de Responsabilidad).

Del mismo modo, se expondrán las diferentes opciones de bases de datos orientadas a grafos que existen en el mercado actual, se realizará un análisis exhaustivo de las mismas y se seleccionará la más adecuada para el caso de uso.

3.1 - Ventajas y desventajas de BDG respecto a BDR

Unos de los principales motivos por los que se ha decidido emplear una base de datos orientada a grafos en este proyecto, es por la comodidad visual que los grafos ofrecen a la hora de analizar las relaciones entre diferentes entidades junto con la eficiencia para las consultas.

El proceso de investigación de derivaciones de responsabilidad que efectúa el departamento de Gestión de Deudas es un proceso muy lento, costoso y manual. Los empleados, trabajan con una herramienta poco gráfica, van analizando diferentes entidades de una en una y a mano. Y, además, disponen de diferentes fuentes de las cuales obtienen la información deseada.

Por ejemplo, unas de las tareas principales de los trabajadores de este departamento es la búsqueda de relaciones entre personas jurídicas que forman un grupo consolidado. Dentro de este grupo existen las sociedades dominantes (aquellas que dominan a otras sociedades) y las dominadas.

Actualmente, para la obtención de este grupo, los empleados deben seguir los siguientes pasos:

Operación 1: Para saber si una empresa X es dominante:

- Acción T1 → opción 2 → Insertar NIF de la sociedad X → insertar $año$ → en la casilla 560 se indica si es una sociedad dominante.

Operación 2: Para saber si una empresa X es dominada:

- Acción T1 → opción 2 → insertar NIF de la sociedad X → insertar año →en la casilla 561 se indica si la sociedad es dominada.

Operación 3: Para saber el grupo consolidado de una empresa X:

- Acción T1 → opción 2 → Insertar NIF de la sociedad X → insertar año → mayusuclas + PF11 → código 658, donde aparecen las sociedades que componen el grupo consolidado SIN DISTINGUIR CUAL DE ELLAS ES DOMINANTE O DOMINADA. Luego si se quiere saber cuáles de las sociedades obtenidas son dominantes y dominadas, por cada una de ellas se tiene que

volver a realizar las dos operaciones reflejadas arriba (*Operación 1* y *Operación 2*). En la **Figura 7** se ve reflejado el flujo de acciones que deben seguir.

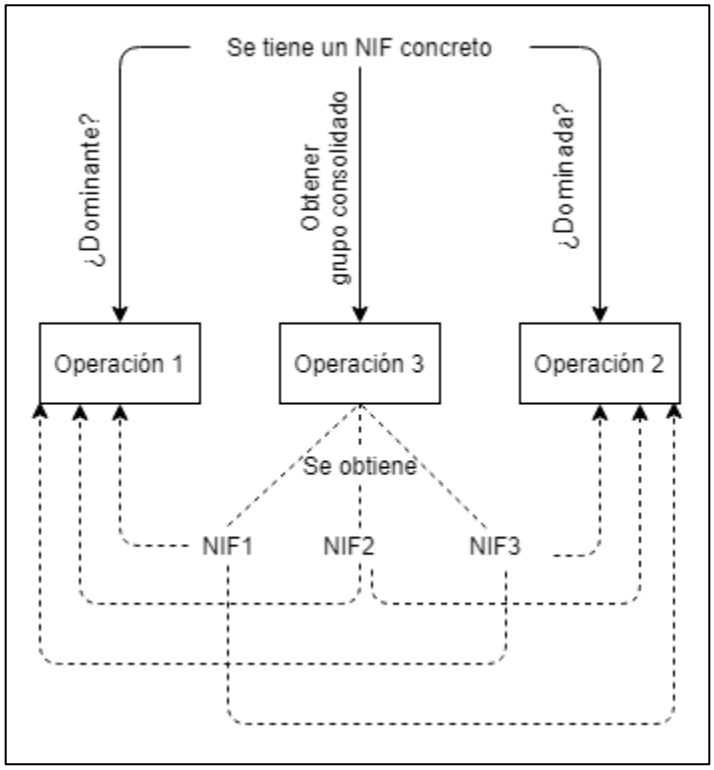


Figura 7 *- Flujo de las operaciones que se deben seguir para obtener un grupo consolidado.*

Como se puede apreciar, solo para obtener el caso de uso del grupo consolidado de sociedades se debe realizar múltiples consultas enlazadas. Ahora bien, para la investigación de las derivaciones, los empleados necesitan realizar muchos más casos de uso, en los que se necesita mantener la información de los anteriores para realizarlos. Además, si el conjunto de datos es grande, el coste y el esfuerzo aumenta considerablemente.

De esta manera, mediante las bases de datos orientadas a grafos se quiere facilitar y automatizar todo este proceso. Mediante un grafo se puede apreciar visualmente y obtener, con una sola consulta/operación, qué empresas están dominadas por otras y cuáles son las dominantes. Los nodos representarían las empresas mientras que las aristas dirigidas simularían la dominación. Al almacenarlo en un grafo su visualización es directa, no obstante, en las tablas relacionales habría que realizar consultas recursivas para construir este grupo consolidado de empresas, como bien se acaba de ver en la **Figura 7**.

Por otra parte, se quiere centralizar toda la información en un solo grafo. De esta manera se evita tener que acceder a diferentes fuentes para realizar los diferentes casos

de uso. Desde una sola operación, por ejemplo, se podría obtener un grafo que representase el grupo consolidado entero, junto con los diferentes bienes que posee cada sociedad. Así se conseguiría obtener una visualización simultanea de los dos casos de uso.

Asimismo, estos no son los únicos motivos por los cuales se ha decidido decantarse por una base de datos orientada a grafos. Seguidamente, se van a analizar aspectos más técnicos de los dos tipos de bases de datos que ayudan a entender la decisión tomada.

3.1.1 - Flexibilidad y escalabilidad

Las bases de datos orientadas a grafos ofrecen flexibilidad y escalabilidad al modelo, haciendo fácil la tarea de modificación de los tipos de nodos, así como la incorporación de nuevos nodos o relaciones. Al mismo tiempo, también es aplicable a las propiedades de los nodos y aristas.

Sin embargo, en los modelos relacionales no se consideran los esquemas libres, ya que se debe especificar por adelantado un esquema. Se debe definir el dominio de una forma exhaustiva, es decir, se debe especificar el tipo de propiedad, la cantidad de atributos a utilizar, etc. Esto supone un obstáculo para los casos en los que la base de datos está constantemente cambiando.

Mediante el siguiente ejemplo, se verá cómo afectaría el mismo cambio del modelo en una base de datos relacional y en una base de datos orientada a grafos. Para ello, se ha utilizado el caso de uso de *Twitter* que se ha visto en el *Capítulo 2*.

Según lo establecido, para una cuenta de *Twitter* sólo se puede tener una cuenta de correo electrónico vinculada. Sin embargo, se va a cambiar el diseño y se va a establecer que para una cuenta de *Twitter* se pueden tener varios correos electrónicos vinculados.

En el caso del modelo relacional, el diseñador se vería obligado a crear una nueva tabla (tabla *Vinculación* en la **Figura 8**) en la que se tuviese el *id* del usuario y el *id* del correo como clave primaria y clave foráneas. Esto supondría la creación de la nueva tabla junto con la migración de datos a la misma.

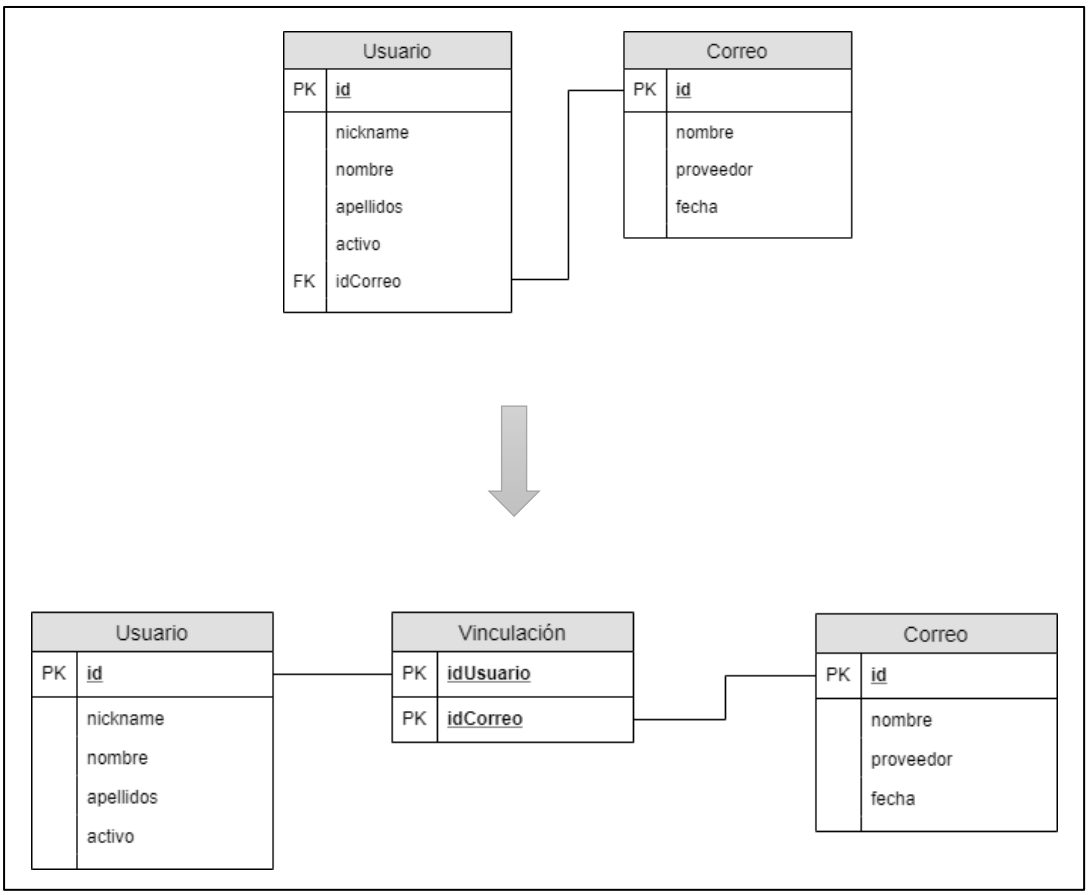


Figura 8 *- Transformación del modelo relacional.*

Mientras que en la base de datos orientado a grafos sería suficiente con añadir un nuevo nodo interconectado mediante una nueva arista. Una tarea independiente, ya que no afecta a ningún otro elemento del modelo, ni a la estructura del grafo, véase **Figura 9**.

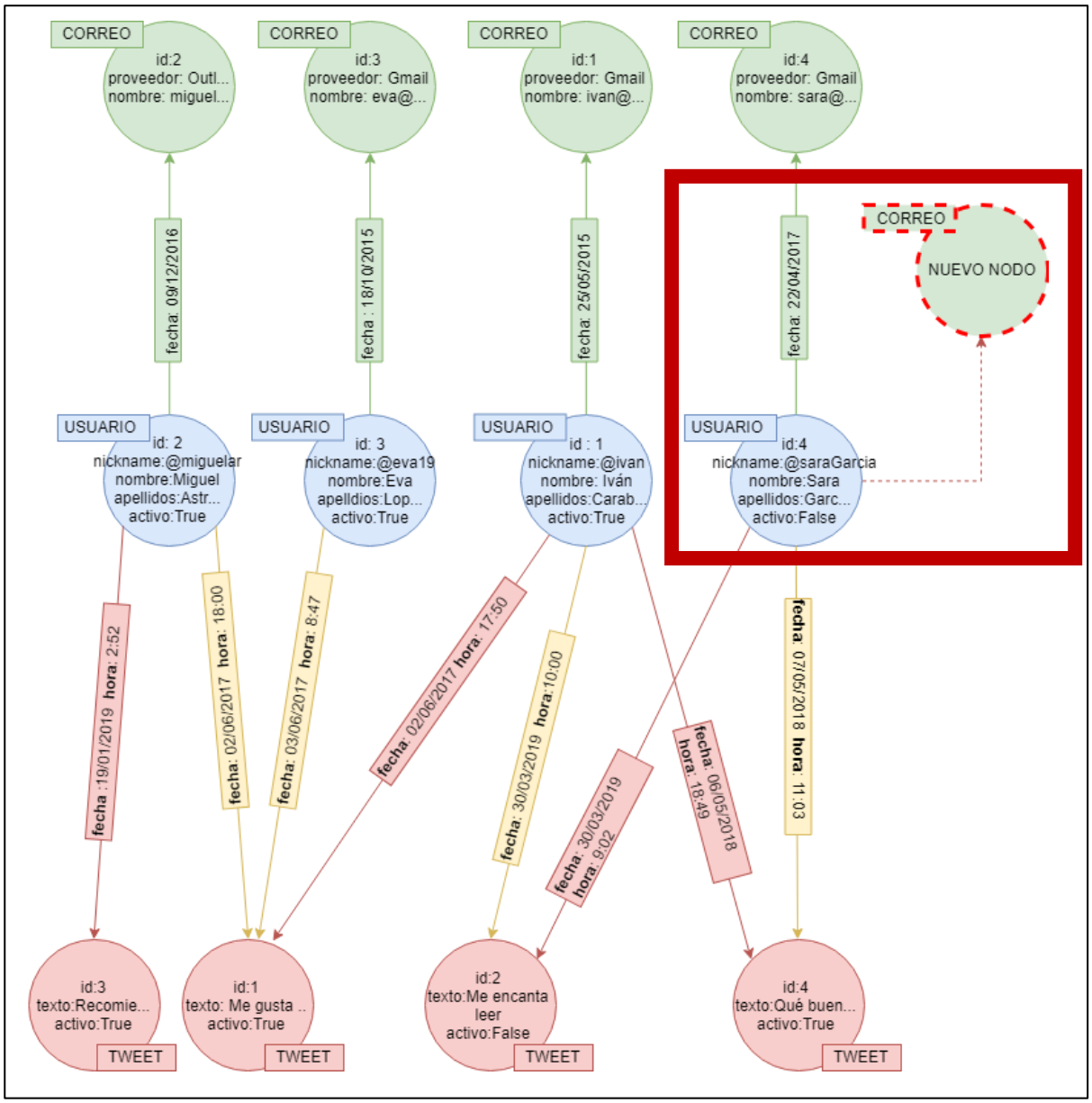


Figura 9 *- Transformación del modelo orientado a grafos.*

3.1.2 - Rendimiento y búsquedas transversales

El rendimiento es una característica preponderante a la hora de seleccionar el tipo de base de datos, ya que, dependiendo del caso de uso específico del proyecto, puede que sea más eficiente un tipo u otro.

Por ello, para casos en los que prevalece las consultas transversales, es decir, consultas en las que se necesitan dar varios saltos entre datos para obtener el resultado deseado, el rendimiento de una base de datos orientada a grafos es mayor que la de una relacional. Esto se debe gracias a la estructura de datos de la base de datos, ya que, los datos que están relacionados se apuntan directamente de una forma u otra, haciendo que las lecturas trasversales sean rápidas y poco costosas.

Sin embargo, en el modelo relacional, a medida que se van enlazando más cláusulas de join (avanzas en los niveles de profundidad de la búsqueda), mayor carga para el motor de búsqueda, por lo que concluye en un peor rendimiento del mismo.

No obstante, no todo son características ventajosas en las bases de datos orientadas a grafos. Por una parte, la ganancia del rendimiento de lectura trae como consecuencia una decadencia en el rendimiento de las escrituras. Por otra parte, el rendimiento empeora si no existen relaciones entre los datos (grafos nulos o grafos con pocas aristas) y si las consultas abarcan toda la base de datos entera. Además, no todas las bases de datos orientadas a grafos cumplen con las características ACID[1], lo que supone un rendimiento deficiente cuando se ejecutan transacciones.

3.1.3 - Indexación

La indexación no está presente en los dos modelos. En los modelos relacionales se habla de una indexación en tiempo de modelización. Cuando se efectúa la modelización de la base de datos, se establecen diferentes atributos (columnas) para el almacenamiento de los índices (claves foráneas) que relacionan los datos. Sin embargo, en las orientadas a grafos no existe la indexación, puesto que las aristas interconectan físicamente los datos relacionados. Por consiguiente, se obtiene un acceso más rápido a los datos, pero se necesita más capacidad de almacenamiento respecto a los modelos relacionales, ya que los punteros físicos ocupan un espacio de memoria extra.

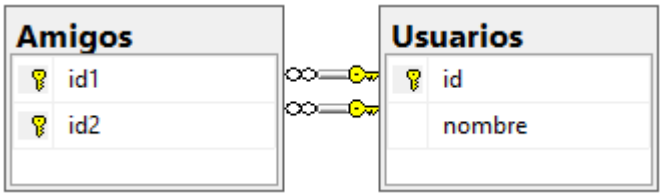
3.1.4 - Lenguaje

Para el análisis de esta característica se han considerado dos lenguajes. Por una parte, se encuentra el lenguaje SQL para los modelos relacionales y por otro lado el lenguaje *Cypher* para los modelos de grafo.

[1] Las características ACID son: Atomicidad, Consistencia, Aislamiento y Durabilidad.

Se ha presentado un mismo caso para los dos lenguajes: *se quiere obtener los amigos de un usuario en concreto, junto con los amigos de los amigos*. El usuario origen para la búsqueda será el usuario número uno.

En la **Figura 10**, se puede apreciar cómo se resumiría el modelo del caso de uso planteado en una base de datos relacional.



Amigos	
id1	id2
1	2
1	5
1	9
2	3
2	4
5	6
5	7
7	8

Usuarios	
id	nombre
1	Iker
2	Pepe
3	Sara
4	Antonio
5	Adrian
6	Pablo
7	Ainhoa
8	Estela
9	Inma

Figura 10 *- Modelo relacional del caso de uso a estudiar.*

Para realizar la búsqueda deseada, la consulta en el lenguaje SQL se figuraría de la siguiente forma:

```
SELECT amigos1.id2 AS "amigos (nivel 1)", amigos2.id2 AS "amigos de amigos (nivel 2)", amigos3.id2 AS "amigos de amigos (nivel 3)"
FROM (Amigos as amigos1 LEFT JOIN Amigos AS amigos2 on amigos1.id2 = amigos2.id1)
        LEFT JOIN Amigos AS amigos3 ON amigos2.id2 = amigos3.id1
WHERE amigos1.id1 = 1
```

Figura 11 *- Consulta en lenguaje SQL.*

El resultado obtenido se muestra en la **Figura 12**. El usuario con id igual a 1 consta de tres amigos directos (usuario 2, 5 y 9), y de cinco amigos de amigos (usuarios 3,4,6,7 y 8).

amigos	amigos de amigos (nivel 1)	amigos de amigos (nivel 2)
2	3	NULL
2	4	NULL
5	6	NULL
5	7	8
9	NULL	NULL

Figura 12 *- Resultado de la consulta SQL.*

Asimismo, se puede apreciar que el elemento primordial de la consulta es la cláusula *left join*, ya que establece la secuencia de vínculos entre los amigos y los amigos de los amigos. Para este caso se han utilizado dos *joins* que equivaldría a una búsqueda en profundidad de nivel 3. Luego si se quisiese ampliar la profundidad de búsqueda en n saltos, habría que enlazar $n - 1$ cláusulas de *joins*, lo cual hace que la consulta sea más laboriosa y menos accesible y legible.

Por lo contrario, en un lenguaje orientado a grafo, las consultas son más expresivas y siguen una estructura propia de un grafo. Se considera que es un lenguaje natural, ya

que, la propia consulta en sí imita o representa el subgrafo que se está buscando, véase **Figura 13** y **Figura 14**.

Los paréntesis representan los nodos, las flechas "-->" figuran las aristas y dentro de los corchetes se especifica el número de saltos (niveles) que se quiere dar. En este caso se quiere transitar desde el nivel 1 (amigos) hasta el máximo de niveles del grafo.

Además de ser un lenguaje más expresivo, el aumento de niveles en la búsqueda no supone ninguna modificación sustancial de la consulta. El único parámetro a cambiar sería el nivel de profundidad.

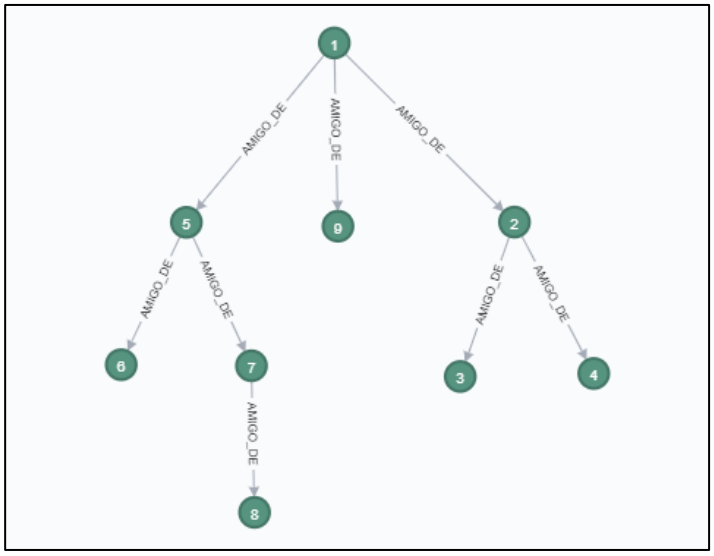


Figura 13 *- Modelo orientado a grafos del caso a estudiar.*

```
$ match (n:User{Id:1})-[*1..]->(amigos:User) return amigos
```

Figura 14 *- Consulta en un lenguaje propio de una base de datos orientada a grafos (lenguaje Cypher).*

3.1.5 - Casos de uso

Actualmente, las bases de datos orientadas a grafos son recomendadas para la resolución de problemas en los que es primordial obtener un buen rendimiento en las consultas en tiempo real. De igual manera, son destacables en algoritmos iterativos y búsquedas de profundidades considerables.

Áreas como la minería de datos, el aprendizaje automático y Big Data son principales consumidores de este tipo de base de datos.

3.2 - Búsqueda de referencias en el mercado

Una vez consideradas todas las características del apartado anterior, se ha decidido con qué tipo de base de datos se va a trabajar (orientada a grafos) y se procede a realizar una investigación de las diferentes referencias del mercado actual.

En el mercado se dispone de una multitud considerable de herramientas para trabajar con grafos, pero no todas trabajan de la misma forma o tienen las mismas características.

Para la selección de la base de datos orientada a grafos, se va a tener en cuenta los siguientes aspectos. En primer lugar, la manera de enfocar las diferentes bases de datos orientadas a grafos (nativa o no nativa). Y, en segundo lugar, el modelo de grafo que se quiere implementar (*Resource Description Framework* o *Labeled Property Graph*).

3.2.1 - Enfoque nativo y no nativo

Es uno de los aspectos más importantes para la selección de la base de datos orientada a grafos, ya que una de las alternativas obtiene mejores resultados que la otra.

3.2.1.1 - Enfoque no nativo

Este enfoque se considera no nativo puesto que no procesa de una manera óptima y coherente los grafos. Asimismo, abusan de otros componentes externos para la obtención de los datos. Generalmente estos componentes suelen ser sistemas manejadores de bases de datos (SGBD) relacionales, de columnas u otros NoSQL (sin incluir BDG). [5]

Dentro de este mismo enfoque se pueden distinguir dos perspectivas diferentes:

Capa de grafo [5]

En este tipo de enfoque se añade una capa superior a un SGBD no orientado a grafos. Esta capa a veces se presenta como una API de grafos o como con una interfaz de grafos. De esta manera, teniendo un SGBD relacional o de documentos, se puede ampliar a un SGBD multimodelo que incluye el modelo de grafos de una manera sencilla. En resumidas cuentas, es la adición de una capa que hace parecer que se está interactuando con un modelo de grafos, cuando realmente debajo de esta capa se dispone de un modelo relacional o de documentos.

Operadores de grafos [5]

Este enfoque no añade ninguna capa extra. La idea principal de este enfoque es extender el lenguaje de consultas del SGBD de documentos o relacional. Se incorporan nuevos operadores de grafos al lenguaje existente. Sin embargo, a pesar de ofrecer nuevos operadores con los que poder trabajar con un grafo, la optimización de este enfoque es muy reducida, ya que, se sigue trabajando sobre un SGBD relacional o de documentos sin ninguna modificación.

Ejemplos con un enfoque no nativo del mercado actual: *Apache Spark Graph X, Apache Giraph, MongoDB, Gephi, Graph-tool, etc.*

3.2.1.2 - *Enfoque nativo*

El enfoque nativo, sin embargo, es la antítesis del enfoque anteriormente mencionado. Este enfoque resuelve todas las deficiencias presentadas en el apartado anterior que concluyen en dos principios fundamentales: *almacenamiento y procesamiento nativo*.

Almacenamiento nativo (Native graph storage)

Las bases de datos con esta característica son aquellas que han sido optimizadas y diseñadas para almacenar y manejar grafos. Es decir, los nodos y las relaciones están almacenados en memoria cerca el uno del otro y además hacen uso del sistema de ficheros de una forma que beneficia a la estructura de los grafos. Lo contrario a estas bases de datos sería aquella que serializa el grafo en una base de datos relacional.

Procesamiento nativo (Native graph processing)

Esta característica sirve para describir el procesamiento de las operaciones y consultas. En otros términos, describe la forma en la que el motor de la base de datos planifica, optimiza y ejecuta las posibles consultas.

Pese a que existe una fuerte controversia en torno a este concepto, uno de los mayores aspectos a considerar es la adyacencia sin límites, fuertemente conocida como ***index-free adjacency*** en inglés.

Actualmente, no se tiene una definición oficial y estándar de lo que supone tener un procesamiento de grafos nativo. Por una parte, se sitúan los expertos que afirman que para considerar que una BDG satisface esta característica se debe cumplir la adyacencia sin límites *(index free adjacency).*

Sin embargo, existe otro colectivo el cual no considera la adyacencia sin límites para determinar si una BDG cumple con este aspecto nativo. Solo se centran en que el mecanismo a utilizar sea tan bueno como la técnica de *index free adjacency*.

Dejando de lado la presente discordia, la propiedad de *index-free adjacency* implica la supresión de la indexación global. Dicho de otro modo, no se utiliza una estructura o índice para relacionar datos. La relación es directa, mediante una dirección física de la memoria RAM, también conocido como puntero. Además, el almacenamiento o creación de estos punteros en RAM se realiza cuando se cargan los datos (no cuando se lanza una consulta) [6]. Es importante destacar que se disponen de punteros tanto para

los nodos vecinos como para las relaciones en sí, ya que estas pueden tener valor añadido como las propiedades, etc [7].

En definitiva, *index-free adjacency* se puede sintetizar en tres nociones básicas:

1. No se utilizan índices.
2. Se dispone de referencias directas.
3. Contraposición de los árboles de búsqueda binaria.

En la **Figura 15** se puede apreciar una representación abstracta y general del procesamiento de un grafo con *index free adjacency*, sin entrar en tecnicismos. Se considera que se quiere acceder desde el nodo A hasta el nodo B. En el propio nodo A se almacena la dirección física de los nodos vecinos (B y C), luego el acceso es rápido y directo.

Finalmente, cabe destacar, que la adyacencia sin límites supone una mejora de eficiencia, ya que es independiente del tamaño total del grafo (es proporcional al tamaño del subgrafo que se quiere analizar). Esta propiedad hace que las consultas sean más rápidas que en una base de datos relacional, por ejemplo.

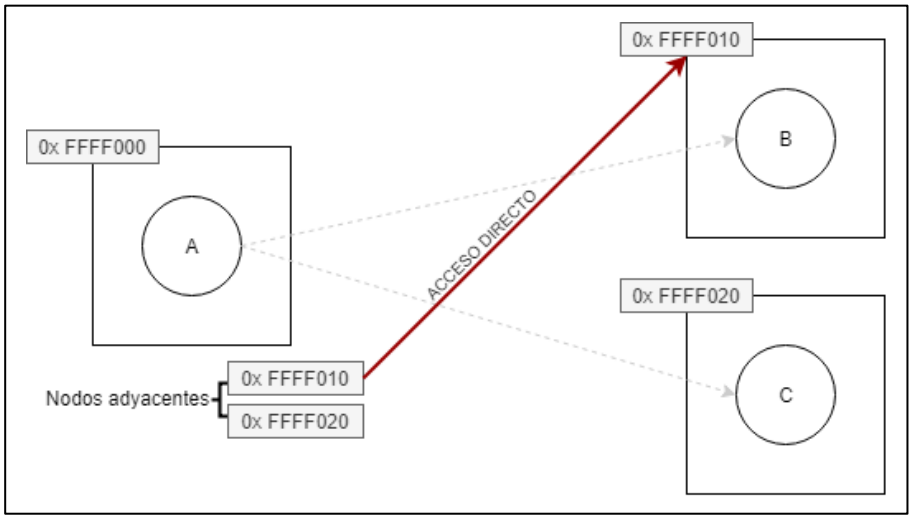


Figura 15 *- Representación gráfica de la propiedad index-free adjacency.*

3.2.2 - Resource Description Framework (RDF) y Labeled Property Graph (LPG)

Asimismo, además de las propiedades de almacenamiento y procesamiento (*native graph processing* y *native graph storage*), también se contempla el modelo de datos especifico. Los modelos principales son: RDF y LPG.

3.2.2.1 - RDF (Resource Description Framework)

El marco RDF es un estándar orientado a la web semántica[2]. De ahí a que las bases de datos orientadas a grafos RDF tambien sean conocidos como *semantic graph database*.

Este tipo de base de datos tiene como objetivo representar, publicar, clasificar y reportar la información sobre los recursos de la web estableciendo estándares. A pesar de estar orientado a los recursos web, es frecuente el uso de la base de datos como modelo de datos software en entornos locales. Son muy útiles a la hora de representar metadatos y referencias en web o conceptos de alta complejidad en local.

En cuanto arquitectura, el grafo RDF es un modelo *triple store*, cuyo núcleo es el triplete [subject-predicate-object]. El sujeto y el objeto son nodos y el predicado es una arista. El sujeto y el predicado representan los diferentes recursos. Sin embargo, el objeto puede representar un recurso o un valor literal. Un ejemplo de este modelo de grafo se puede ver en la **Figura 16**.

Con este modelo se consigue una completa descomposición de los datos, ya que, todo es representado mediante un nodo o arista, sin estructuras internas en cada uno de ellos. Por ejemplo, los atributos serán representados como nodos adicionales, a diferencia de los LPG.

Los ejemplos más señalados son: *Jena, Rya, AllegroGraph, DGraph.*

Ventajas

- ✓ Ofrece métodos de integración entre diferentes fuentes de información.
- ✓ Permite interoperabilidad entre distintas máquinas de procesamiento. Es decir, interoperabilidad entre los agentes inteligentes encargados de la búsqueda de información de forma automatizada.
- ✓ Permite seguir una semántica, esquema, reglas y razonamiento de los datos.

2 *[Extraído de Wikipedia] - https://es.wikipedia.org/wiki/Web_semántica*
Web semántica: Son actividades cuyo objetivo es añadir metadatos semánticos y ontológicos a la World Wide Web. Esta información añadida describe el contenido, el significado y la relación de los datos y debe de seguir una estándar para se pueda automatizar el trabajo de las máquinas de procesamiento. El objetivo es mejorar Internet ampliando la interoperabilidad entre los sistemas informáticos usando "agentes inteligentes".
Agentes inteligentes: Programas en las computadoras que buscan información sin operadores humanos.
Ontología: Es una definición formal de tipos, propiedades, y relaciones entre entidades.

Desventajas

- ☒ Rendimiento no tan optimo como en LPG.
- ☒ Complejidad alta.

Casos de uso

- Aplicaciones que buscan la integración de los datos.
- Knowledge graph[3].
- Aplicaciones cuya finalidad es la interoperabilidad semántica[4].
- Aplicaciones que buscan semántica y relaciones de los recursos web.

[3] **Knowledge graph:** Grafo que proporciona información de una forma estructurada y detallada, junto con las relaciones y enlaces existentes. Google hace uso de este grafo para ayudar al motor de búsqueda.
[4] **Interoperabilidad semántica**: La capacidad de intercambiar información de una manera significativa (con significado).

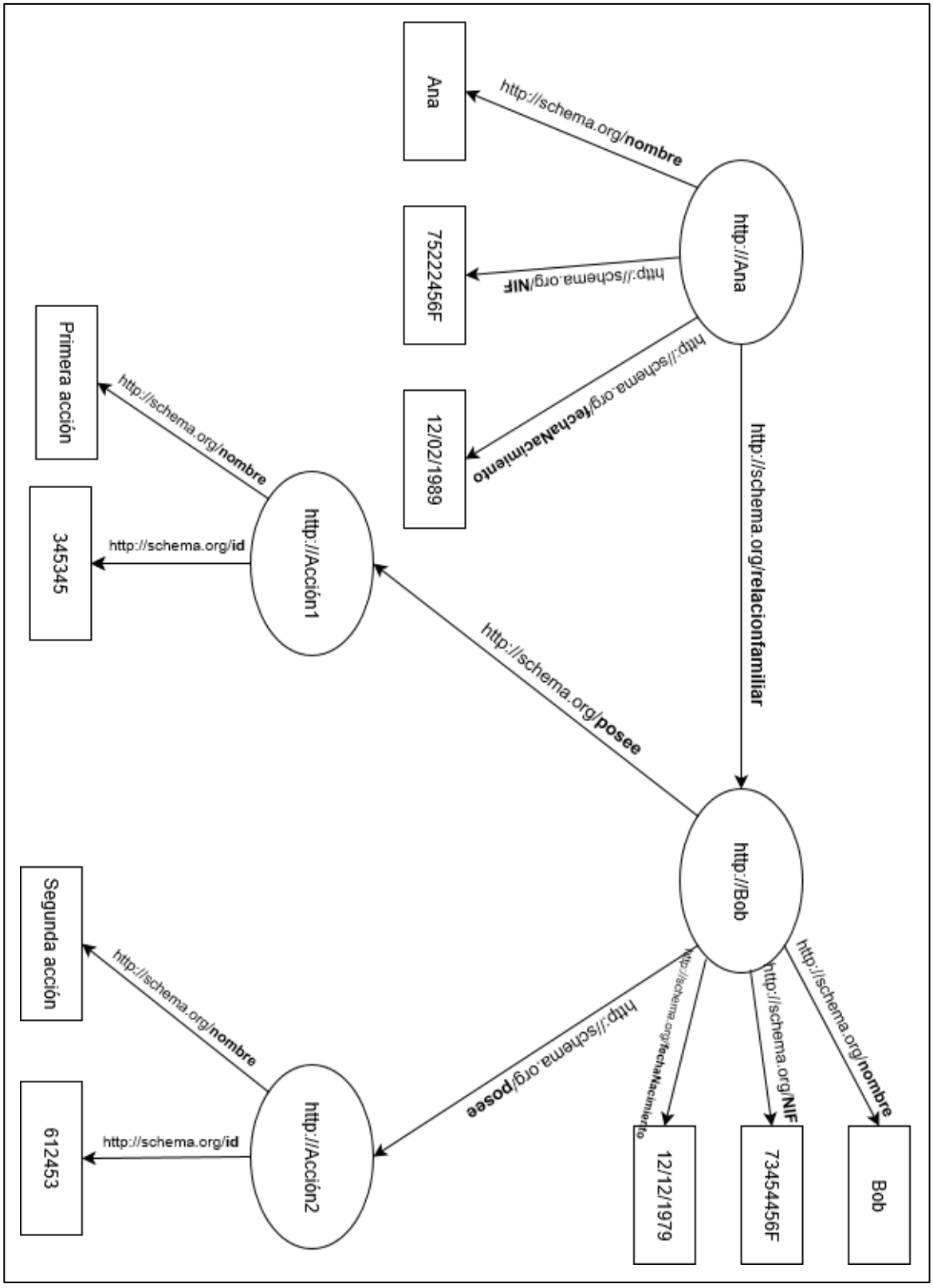


Figura 16 *- Ejemplo grafo RDF.*

3.2.2.2 - *LPG (Labeled Property Graph)*

Los grafos LPG es un modelo cuyo objetivo es obtener un almacenamiento eficiente con el que se pueda ejecutar consultas rápidas y permita un recorrido rápido a través del grafo. A diferencia de los grafos RDF, no buscan el intercambio o la publicación de los datos.

Como en todos los modelos de grafos, existen nodos que están relacionados mediante aristas. Tanto las relaciones (aristas), como los nodos, tienen un *id* para la identificación del mismo y un conjunto de clave-valor para representar las propiedades. Luego, como bien se ha mencionado anteriormente, cada elemento tiene una estructura interna (conjunto de clave-valor).

En los RDF cada atributo es representado por nodos, sin embargo, en los grafos LPG se almacenan en la estructura interna de cada nodo o arista. Esto supone una reducción considerable del tamaño del grafo, véase **Figura 17**.

Por ejemplo, si se quiere representar la relación entre dos personas y por cada persona se quiere almacenar dos atributos. El tamaño del grafo RDF sería de 6 nodos y 5 aristas (2 nodos para las entidades y 4 nodos para los atributos), mientras que el tamaño de LPG sería de 2 nodos y 1 arista (dos nodos solo para entidades, no se tienen nodos adicionales para los atributos).

Ejemplos: *Titan, Neo4J, Cosmos, Tiger Graph.*

Ventajas

- ✓ Flexible.
- ✓ Rápidos en funciones trasversales (rápidos para recorrer grafos y aplicar algoritmos).

Desventajas

- ☒ No hay interoperabilidad.
- ☒ Esquemas básicos.
- ☒ No hay estandarización.

Casos de uso

- Aplicaciones operacionales.
- Aplicaciones con consultas trasversales complejas.
- Empleo de algoritmos.
- Casos en los que se quiera guardar un número considerable de propiedad por cada nodo o relación.

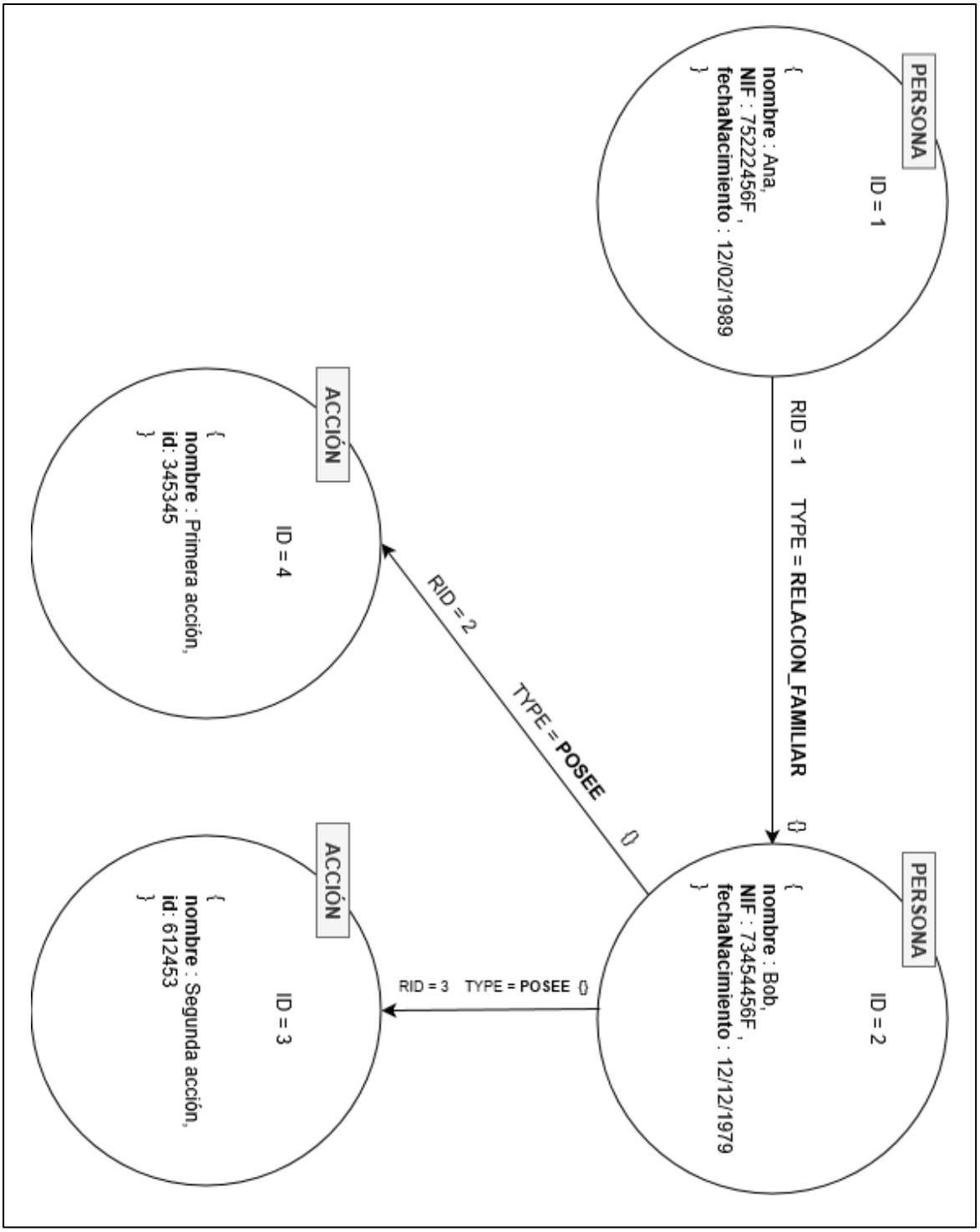


Figura 17 *- Ejemplo grafo LPG.*

3.2.2.3 - *Diferencias*

- RDF está orientado al intercambio y publicación de la información de la web, mientras que LPG busca un almacenamiento y procesamiento eficiente dejando alado la interoperabilidad que ofrece RDF. Es decir, los LPG son más pragmáticos/prácticos mientras que los RDF se centran más en la semántica, estandarización e integración de los datos.
- Los LPG son más aconsejables si se quiere recorrer en profundidad los grafos. Es decir, más óptimo para aplicar algoritmos. En RDF el coste de atravesar un nodo es logarítmico.
- El orden de magnitud de RDF es mucho más grande que el de LPG. Esto puede verse reflejado en la eficiencia y rapidez de las consultas.
- En los grafos RDF no se permite añadir propiedades a las aristas, en los LPG sí.
- En RDF no se puede tener el mismo tipo de relación para dos nodos, ya que cada arista no tiene un *id* único -se identifica mediante la URI que para las aristas de un tipo en concreto es el mismo-. En LPG si se permite tener dos aristas del mismo tipo para dos nodos.

En la siguiente figura se puede apreciar a qué grupo pertenecen alguna de las bases de datos del mercado actual (no se muestran todas). Además, cabe destacar que existen bases de datos que utilizan los dos tipos de modelos explicados, como *AWS* y *Stardog*.

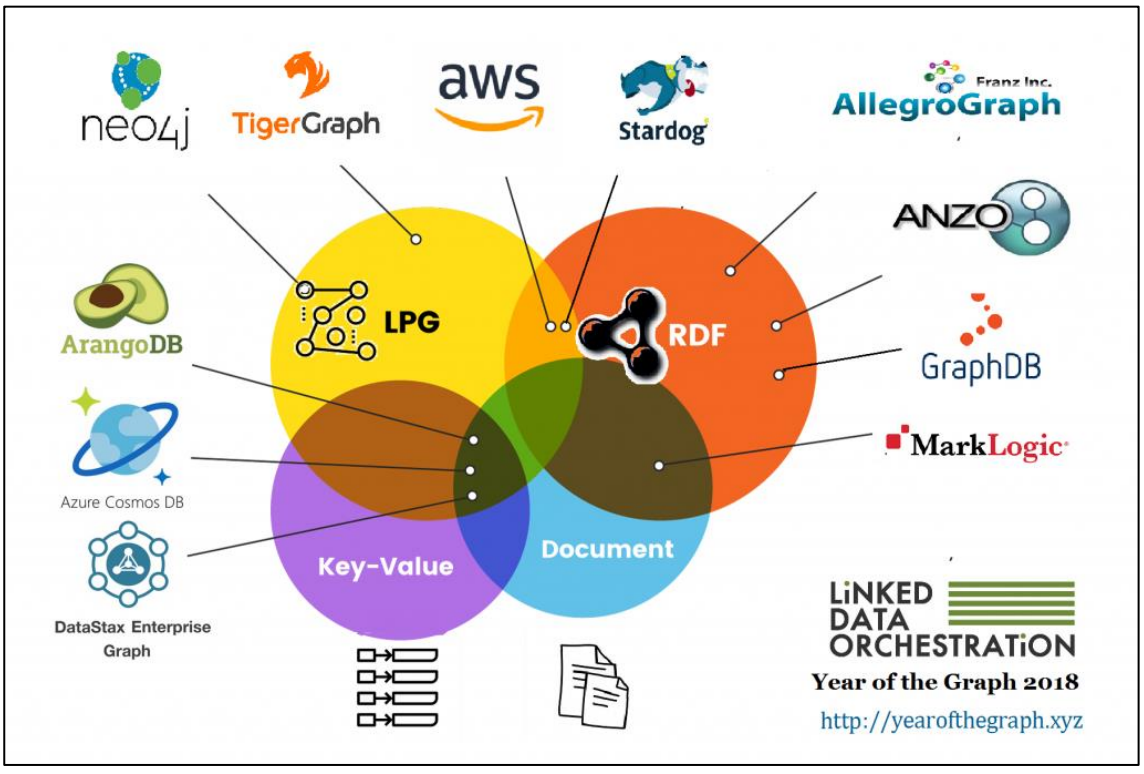


Figura 18 *- Clasificación de BDG del mercado. [20]*

3.3 - Selección de la base de datos

Una vez finalizada la investigación de las diferentes referencias existentes en el mercado actual junto con sus características, se debe seleccionar aquella base de datos que mejor se moldee a las necesidades del caso.

En primer lugar, se han descartado todas las bases de datos que no se consideran nativas. Por consiguiente, quedan descartados ejemplos como: *Apache Spark Graph X, Apache Giraph, MongoDB, Gephi, Graph-to, etc.*

Por otro lado, para la definición del conjunto de BDG nativas, se ha considerado como nativa todas aquellas que cumplen con los dos aspectos nativos (*native graph storage* y *native graph storage*), independientemente del acatamiento de *index-free adjacency.*

Por otra parte, se han contemplado las bases de datos cuyo modelo es el LPG. El caso de uso en el que se está trabajando, no busca la interoperabilidad o la integración de los datos, más bien se focaliza en analizar y procesar las diferentes relaciones que puede haber entre diferentes entidades, obligando al grafo a tener que recorrer una cierta cantidad de nodos, dando saltos entre ellos. Además, en la fase de explotación es posible la aplicación de algoritmos iterativos sobre el grafo.

De esta manera, se ha llegado a dos conjuntos de referencias: nativas y grafos con modelos LPG. Tras procesar la intersección entre los elementos de los dos conjuntos, las posibles candidatas son: ArangoDB, Neo4j, TigerGraph y OrientDB.

3.3.1 - Comparación de alternativas y decantación

Escoger una base de datos de las cuatro alternativas propuestas no es un cometido sencillo, ya que, éstas tienen unas características y ventajas muy similares y cualquiera de estas cuatro bases de datos lograrían grandes resultados en nuestro caso de uso.

Por ello, el criterio utilizado para la determinación de la misma es personal e hipotético y los argumentos elaborados se han tenido que aferrar a diferencias mínimas y tal vez indiferentes.

Una vez aclarado todo, la base de datos que se ha seleccionado para llevar a cabo el proyecto ha sido ***Neo4j***. Las razones por las que se ha elegido esta alternativa son las siguientes.

Index free adjacency

Es una de las principales razones que han hecho decantarse por *Neo4j*. Mientras que *Neo4j* se beneficia de esta propiedad, *ArangoDB*, *OrientDB* y *TigerGraph* no lo hacen. Esto no quiere decir que estas tres última no sean óptimas y que no tengan un acceso rápido a los datos; utilizan otras técnicas como mapeos (*índices hash*) que obtienen un tiempo de acceso de $O(1)$.

No obstante, desde el punto de vista personal, se cree que esta propiedad (*index free adjacency*) puede ser un extra, puesto que, parece que se amolda mejor a los principios y fundamentos de los grafos. En resumidas cuentas, los grafos son conjuntos de datos que apuntan a otros conjuntos de datos, que es sumamente equivalente a decir que un nodo (conjunto de datos) es apuntado por otro nodo (otro conjunto de datos) mediante un puntero (arista), que es prácticamente el mecanismo que sigue *Neo4j*.

Rankings y popularidad

DB-Engines Ranking clasifica en un ranking las bases de datos según su popularidad. Este ranking es actualizado cada mes y está disponible en su página web.

Como se puede ver en la **Figura 19**, **Figura 20** y **Figura 21**, *Neo4j* ocupa la primera posición de popularidad entre todas las bases de datos orientadas a grafos. Seguidamente, *OrientdDB* ocupa el tercer puesto, *ArangoDB* el cuarto puesto y finalmente *TigerGraph* el puesto treceavo.

Luego, este hecho ha sido un aspecto a considerar a la hora de seleccionar la base de datos. Además, al tener un gran índice de popularidad existe una mayor cantidad de documentación tanto por parte de *Neo4J* como por parte de la comunidad, lo que supone una gran ventaja para la hora de desarrollar el proyecto.

include secondary database models — 31 systems in ranking, April 2019

Rank Apr 2019	Rank Mar 2019	Rank Apr 2018	DBMS	Database Model	Score Apr 2019	Score Mar 2019	Score Apr 2018
1.	1.	1.	Neo4j	Graph	49.49	+0.91	+8.59
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	26.28	+1.45	+9.09
3.	3.	3.	OrientDB	Multi-model	6.19	+0.06	+0.55
4.	4.	4.	ArangoDB	Multi-model	4.29	+0.03	+0.49
5.	5.	5.	Virtuoso	Multi-model	3.31	+0.12	+1.51
6.	↑ 8.	↑ 7.	Amazon Neptune	Multi-model	1.39	+0.36	+0.70
7.	↓ 6.	↑ 13.	JanusGraph	Graph	1.38	+0.06	+1.09
8.	↓ 7.	↓ 6.	Giraph	Graph	1.20	+0.16	+0.16
9.	↑ 13.	↑ 17.	Dgraph	Graph	1.08	+0.39	+0.94
10.	↓ 9.	10.	GraphDB	Multi-model	0.97	+0.04	+0.51
11.	↓ 10.	↓ 8.	AllegroGraph	Multi-model	0.89	+0.02	+0.31
12.	12.	↓ 9.	Stardog	Multi-model	0.81	+0.04	+0.29
13.	↓ 11.	↑ 15.	TigerGraph	Graph	0.81	+0.00	+0.64
14.	↑ 17.	↓ 12.	Sqrrl	Multi-model	0.59	+0.10	+0.20
15.	↓ 14.	↓ 11.	Graph Engine	Multi-model	0.56	0.00	+0.16
16.	↓ 15.	↑ 19.	Blazegraph	Multi-model	0.56	+0.01	+0.43
17.	↑ 18.	↑ 22.	InfiniteGraph	Graph	0.40	+0.01	+0.29
18.	↓ 16.	↑ 21.	FaunaDB	Multi-model	0.37	-0.15	+0.26
19.	↑ 20.	↓ 18.	FlockDB	Graph	0.27	+0.01	+0.14
20.	↑ 23.	↑ 25.	AgensGraph	Multi-model	0.23	+0.06	+0.20
21.	↓ 19.	↓ 20.	InfoGrid	Graph	0.22	-0.05	+0.11
22.	22.		AnzoGraph	Graph, Multi-model	0.22	+0.04	
23.	↓ 21.	↓ 16.	HyperGraphDB	Graph	0.21	+0.03	+0.06
24.	24.	↓ 14.	Sparksee	Graph	0.15	+0.03	-0.06
25.	↑ 27.	↑ 27.	GRAKN.AI	Multi-model	0.11	+0.01	+0.09
26.	↑ 28.	26.	GraphBase	Graph	0.11	+0.03	+0.08
27.	↓ 25.	↓ 24.	TinkerGraph	Graph	0.10	+0.00	+0.06
28.	↓ 26.	↓ 23.	VelocityDB	Multi-model	0.08	-0.02	+0.01
29.	↑ 30.	29.	HGraphDB	Graph	0.05	+0.04	+0.03
30.	↓ 29.		Memgraph	Graph	0.05	+0.00	
31.	31.		DataChemist	Graph, Multi-model	0.02	+0.02	

Figura 19 *- Ranking de popularidad de las BDG. [25]*

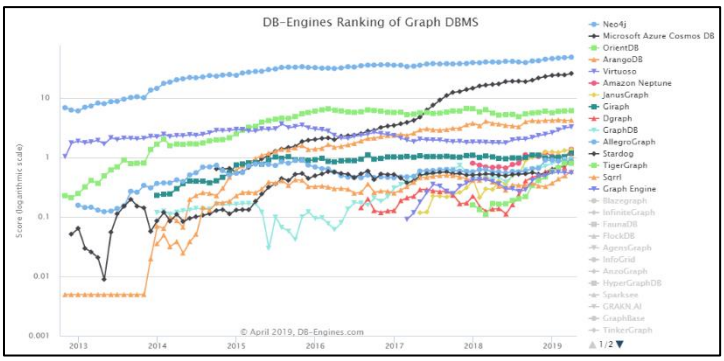


Figura 20 *- Ranking de tendencias de las BDG. [26]*

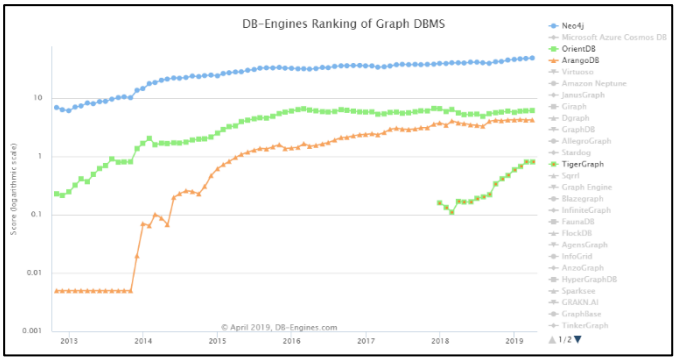


Figura 21 *- Ranking de tendencias de las 4 BDG estudiadas. [26]*

Lenguaje propio de consultas

Otro de los hechos que han incentivado a la base de datos *Neo4j* es el hecho de tener un lenguaje propio de consultas (Cypher) el cual es gráfico, fácil de entender y adecuado para las consultas de grafos. De las cuatro alternativas es el lenguaje que mejor refleja la estructura de un grafo y el que más se aleja de la sintaxis de SQL. Los lenguajes de las otras bases de datos son: *SQL extendido* (*OrientDB*), *AQL* (*ArangoDB*) y *GSQL* (*TigerGraph*)

Otros aspectos

Entre otras cosas, existen diversas herramientas analíticas y de visualización que se pueden enlazar con *Neo4j* y las cuales pueden ser de gran utilidad en un futuro. Además, posee un rango amplio de lenguajes de programación que pueden interactuar con el driver de *Neo4j*, entre ellos Python. También puede accederse a la base de datos a través de la API desde Java o mediante RESTful HTTP, etc.

Para finalizar, se quiere recalcar que no es un resultado determinante y que cualquiera de las tres alternativas restantes llegaría a desempeñar una función tan buena como la elegida o incluso mejor. Sin embargo, por criterios propios se ha seleccionado *Neo4j*.

Características	Neo4j	ArangoDB	OrientDB	TigerGraph
Modelo	Grafo	Multi modelo: - Grafo - Document - Clave-valor	Multi modelo: - Grafo - Document - Clave-valor	Grafo
Año lanzamiento	2007	2012	2010	2017
Lenguaje de implementación	Java, Scala	C, C++, JavaScript	Java	C++
Nativa	Sí	Sí	Sí	Sí
Index free adjacency	Sí	No	No	No
Transacciones	ACID	ACID	ACID	ACID
Lenguaje de consultas	Cypher	AQL	SQL extendido	GSQL
Ranking popularidad	1º	4º	3º	13º
SO disponibles	Linux OS X Solaris Windows	Linux OS X Raspbian Solaris Windows	Todos SO que tengan JAVA JDK con una versión superior a 6	Linux

Tabla 1 *- Comparativa de propiedades de las cuatro BDG. [8] [9]*

3.3.2 - Profundizando en Neo4j y Cypher

Neo4j es una base de datos NoSQL orientada a grafos e implementada en *Java*. Fue desarrollada por *Neo Technology Inc* en 2002 y su primera publicación se realizó en 2007.

Modelo del grafo

Como bien se ha adelantado, el modelo de base de datos en el cual se basa *Neo4j* es el *Labeled Property Graph* (LPG). No obstante, gracias a la flexibilidad de la misma, es posible diseñar un grafo RDF.

Por otro lado, *Neo4j* obliga a trazar un grafo dirigido, por lo que todas aristas deberán tener una dirección. No obstante, a la hora de ejecutar una consulta, las aristas pueden ser utilizadas bidireccionalmente sin ningún coste extra, por lo que no es necesario duplicar las aristas con la dirección opuesta [10].

Tanto los nodos como las relaciones pueden ser etiquetadas y, además, pueden disponer de un conjunto de propiedades (*clave-valor*).

Organización física de los datos

Neo4j almacena cada elemento del grafo en diferentes registros que se agrupan en distintos ficheros de almacenamiento.

Registros

- Registros de nodos
- Registros de relaciones
- Registros de propiedades

Ficheros más relevantes

- *neostore.nodestore.db*: Almacena los registros de los nodos.
- *neostore.relationshipstore.db*: Almacena los registros de las relaciones.
- *neostore.propertystore.db:* Almacena los registros de las propiedades.

En cuanto a los **registros**, *Neo4j* los establece con un tamaño fijo que puede variar dependiendo de la versión de *Neo4j* que se esté utilizando. Disponer de un tamaño fijo ayuda a obtener un acceso y una búsqueda rápida de los mismos en el archivo que los almacena. Supongamos que cada registro ocupa 15 bytes y que por cada registro se dispone de un identificador que los identifica unívocamente. De esta manera, si se tiene el nodo con id igual a 100, se sabe concretamente que este nodo (registro) comienza en

el byte 1500 (100*15) del archivo de almacenamiento. Con esta técnica, el sistema gestor de bases de datos puede calcular directamente la posición que ocupa el nodo y acceder a él con un coste de $O(1)$, a diferencia de las búquedas de árboles binarios, las cuales obtienen un coste de $O(\log n)$ [11].

Es por ello, que se dice que *Neo4j* almacena punteros. Aunque el valor del identificador (ID) que se almacena no sea una dirección de memoria en sí, mediante un cálculo simple y rápido se obtiene la dirección de memoria. Por consiguiente, cada vez que se hable de punteros, realmente son identificadores con el cual se obtiene una dirección de memoria.

Por otra parte, se emplean los términos ***cadena de relaciones*** **y** ***cadena de propiedades***. Para cada nodo existe una cadena de relaciones y otra de propiedades. La primera de ellas es una lista doblemente ligada de registros de relaciones, mientras que la segunda es una lista simple de registros de propiedades. Sin embargo, en los registros de los nodos no se almacenan las dos listas completas, sino que se almacena el primer elemento de cada una de ellas. Como los elementos de las listas están ligados, desde el primer elemento (y desde cualquier otro) se puede recorrer la lista de una forma directa y sencilla.

La distribución de los bytes en los **registros de nodos** se puede apreciar en la **Figura 22**. El primer byte se reserva para el *flag in-use* que sirve para detectar si un registro está siendo utilizado para almacenar un nodo o no. Los siguientes cuatro bytes almacenan el puntero de la primera relación de la *cadena de relaciones*. Los siguiente cuatros bytes están reservados para el puntero a la primera propiedad de la *cadena de propiedades* del nodo específico. Los siguientes cinco bytes se utilizan para las etiquetas. Finalmente, existe un byte extra que se reserva para diferentes *flags* [11]. Uno de estos *flags* sirve para identificar aquellos nodos que tienen un grado/valencia alto, es decir, nodos densos.

Por otro lado, los **registros de relaciones** disponen de 34 bytes, véase **Figura 22**. Cada registro de relación contiene un byte para el *flag in-use*, ocho bytes para los punteros de los nodos origen y destino respectivamente, un puntero al tipo de relación, punteros a la relación *anterior* y *siguiente* de la cadena de relaciones para cada uno de los nodos de origen y destino, un puntero a la primera propiedad de la cadena de propiedades de la relación, y, finalmente, un *flag* extra para saber si la relación es el inicio de la *cadena de relaciones* [11]. Los punteros de las relaciones *anterior* y *siguiente* sirven para componer la *cadena de relaciones* del nodo origen y del nodo destino.

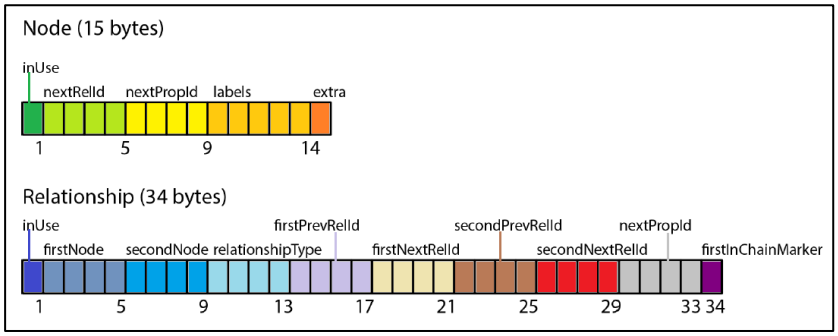


Figura 22 *- Estructura de los registros de nodos y relaciones. [11]*

En la **Figura 23** se puede ver una demostración práctica de toda esta estructura. En el *nodo 1* se dispone de un puntero a la primera propiedad y un puntero a la primera relación de la *cadena de relaciones* del *nodo 1*. A su vez, en el *nodo 2* se dispone de un puntero al primer elemento de la cadena de relaciones del *nodo 2* y un puntero para la primera propiedad del *nodo 2*.

En el caso de que se quiera leer una propiedad de un nodo X, se accederá al puntero de la primera propiedad y se recorrerá toda la lista ligada hasta encontrar la propiedad deseada.

Lo mismo sucede con las relaciones, en el caso de que se quiera buscar una relación, se accederá al puntero de la primera relación de la cadena de relaciones y se recorrerá toda la lista doblemente ligada hasta encontrar la relación que se busca. Recorrer las listas no supone mucho tiempo, ya que, como bien se ha visto antes, se puede acceder a los registros de una forma directa mediante el identificador.

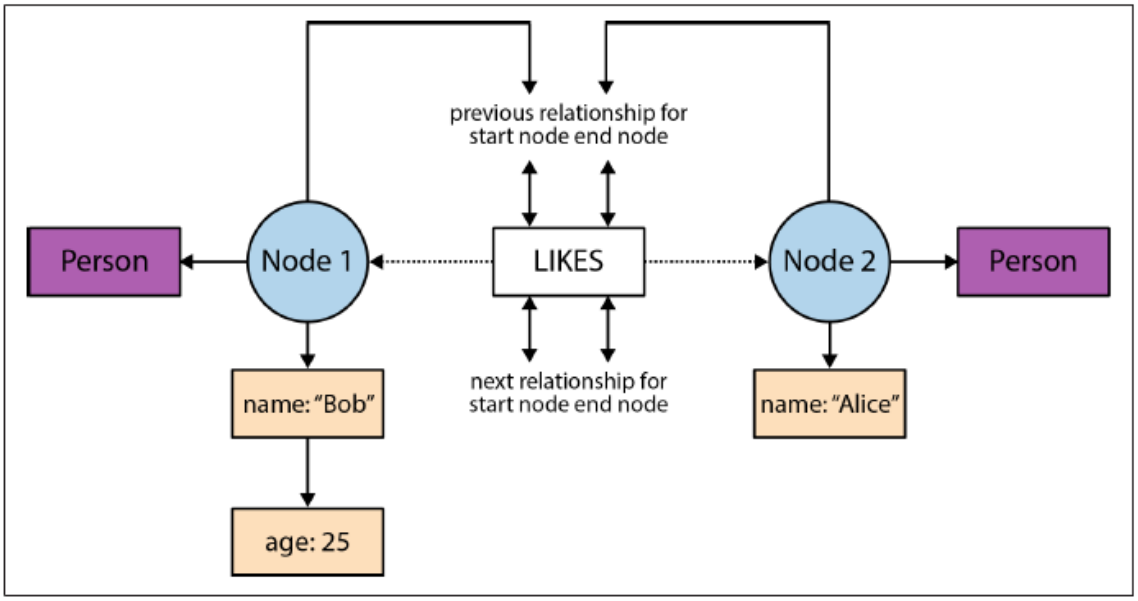


Figura 23 *- Caso práctico del funcionamiento físico de Neo4j. [11]*

Por todo esto se dice que *Neo4j* cumple con la propiedad de *index-free adjacency*, ya que no utiliza ninguna indexación global, sino que calcula la posición en la que se encuentra cada registro.

Suscripciones

Neo4j consta de dos niveles de suscripciones. Por una parte, se encuentra la edición ***community*** que es una licencia gratuita. Y por la otra, se encuentra la edición ***enterprise*** con una licencia de pago.

No obstante, mediante unas licencias autorizadas y específicas y con unas restricciones establecidas por *Neo4j*, es posible hacer uso de la versión *enterprise* de una forma gratuita.

En la edición *enterprise* se dispone de diferentes ventajas reflejadas en la siguiente referencia [12] . Principalmente, las características más destacables de esta edición son:

- Aplicación de escritorio, en donde se puede administrar diferentes proyectos y añadir diferentes plugins.
- Mayor rendimiento.
- Características extras en la configuración de la base de datos.
- Múltiples tipos de restricciones (constraints).
- No tiene una limitación en la cantidad de nodos, relaciones y propiedades

Asimismo, durante el desarrollo del documento, se irán mencionando las diferencias más significantes para el proyecto y, además, se realizará una comparación experimental de las dos, tanto en la importación como en la explotación de los datos.

3.3.2.1 - *Lenguaje Cypher*

Neo4j tiene un lenguaje de consultas propio que se llama *Cypher*. Éste es un lenguaje que proporciona una sintaxis intuitiva y declarativa. La propia consulta representa mediante "ASCII art" los patrones de los grafos que se quieren obtener o crear. Como se puede ver en el **Código 1**, representa los nodos mediante paréntesis y las relaciones mediante flechas "-->".

```
CREATE  (:Persona {nombre : "Iván"})-[:ES_AMIGO_DE]->(:Persona {nombre : "Iker"})
[...]
MATCH   p =  (:Persona )<-[:ES_PAREJA_DE]-(:Persona )-[:ES_AMIGO_DE]->(:Persona)
RETURN  p
```

***Código 1** - Ejemplo de consulta en lenguaje Cypher.*

Creación de nodos y relaciones

La creación de nodos y relaciones nuevas se realiza mediante la cláusula **CREATE**. Por ejemplo, siguiendo el modelo de *Twitter* explicado en el *Capítulo 2 – Apartado 2.2*, se quiere insertar y crear un nuevo usuario en la base de datos. Para ello, se deberá crear dos nodos (*Usuario* y *Correo*) con sus propiedades y una arista con una propiedad de *fecha* que representa la vinculación.

```
CREATE (variable_alicia:Usuario {nickname: "@alicia", nombre: "Alicia", apellidos: "Santos...", activo:True})
CREATE (variable_correo:Correo  {proveedor: "gmail", nombre: "alicia@gmail.com"})
CREATE (variable_alicia)-[:TIENE_VINCULADO {fecha:03/05/2019}]->(variable_correo)
```

***Código 2** - Creación de dos nodos, una arista y siete propiedades en Cypher.*

En el **Código 2** se tiene reflejado como podría realizarse el escenario planteado. En primer lugar, se ha creado un primer nodo con la etiqueta *Usuario* y cuatro propiedades. Esta se ha almacenado en la variable *variable_alicia*. Seguidamente, se ha creado el segundo nodo con la etiqueta *Correo* y las propiedades correspondientes. Este segundo nodo también se ha guardado en otra variable *variable_correo*. Finalmente, se han relacionado los dos nodos mediante una arista con una etiqueta *TIENE_VINCULADO* y la propiedad *fecha*.

Los nodos pueden disponer de más de una etiqueta, mientras que a las relaciones sólo se les puede asignar a lo sumo una etiqueta, véase **Código 3**.

```
CREATE (:Animal:Mamifero)
```

***Código 3** - Creación de nodo con dos etiquetas en Cypher.*

Extracción de datos o subgrafos

Mediante la cláusula **MATCH** se puede extraer un subgrafo determinado, un conjunto de nodos, un conjunto de aristas, un camino, un conjunto de propiedades, etc. Si se quiere filtrar el conjunto de datos o el subgrafo obtenido mediante MATCH, se utilizará la cláusula **WHERE**. Siempre que se llame a estas cláusulas, seguidamente se deberá llamar a **RETURN** para devolver lo que se desea. En el lenguaje de *SQL*, el comando SELECT-WHERE tiene una función similar a la del MATCH-WHERE.

Para analizar mejor el funcionamiento de estas cláusulas se han propuesto diferentes escenarios.

En primer lugar, se quiere obtener el nombre y apellidos del usuario con nickname *@eva19*. Los dos códigos presentados son equivalentes y en los dos se devuelve el mismo conjunto de atributos. En estas consultas se realiza un filtrado de nodos por propiedad y etiqueta, véase **Código 4**.

```
MATCH (nodoFiltrado:Usuario {nickname: "@eva19"})
RETURN nodoFiltrado.nombre, nodoFiltrado.apellidos
```

```
MATCH (nodoFiltrado:Usuario)
WHERE  nodoFiltrado.nickname = "@eva19"
RETURN nodoFiltrado.nombre, nodoFiltrado.apellidos
```

***Código 4** - Filtrado de nodos con Cypher.*

En segundo lugar, se quiere extraer todos los nodos relacionados al nodo anteriormente filtrado (se quiere obtener un subgrafo centrado en un nodo en concreto), véase **Código 5**. Para ello, no se especifica ni la etiqueta de la relación, ni la etiqueta del nodo destino. Así se consigue que se devuelva todos los nodos que estén relacionados con él sin importar el tipo de relación ni la etiqueta del nodo. En este caso, en vez de guardar los nodos y las relaciones en variables diferentes, se ha guardado todo el subgrafo completo, por lo que, en la variable *subgrafo* se dispone de todos los nodos, las propiedades y las relaciones del subgrafo.

```
MATCH  subgrafo = (:Usuario {nickname: "@eva19"})-[]->()
RETURN subgrafo
```

***Código 5** - Extracción de un subgrafo con Cypher.*

Si no se tuviese en cuenta la dirección de la arista, es decir, se quisiese realizar una búsqueda bidireccional, se realizaría la siguiente consulta:

```
MATCH  subgrafo = (:Usuario {nickname: "@eva19"})-[]-()
RETURN subgrafo
```

***Código 6** - Búsqueda bidireccional.*

Para continuar con la explicación de esta cláusula, se quiere realizar una búsqueda en profundidad. Este tipo de consultas es una de las más destacables y ventajosas de este tipo de bases de datos. Supongamos que, en el modelo de *Twitter* explicado anteriormente, se añade la posibilidad de que un Usuario sea amigo (*seguir*) de otro Usuario. De esta manera, se podría obtener el círculo de amigos de una persona en concreto, es decir, obtener los amigos del usuario y los amigos de los amigos. Como se puede deducir, es una consulta en la que se debe de realizar diferentes saltos entre los nodos. Actualmente, mediante este tipo de consultas se intenta agilizar los sistemas de recomendaciones de las redes sociales.

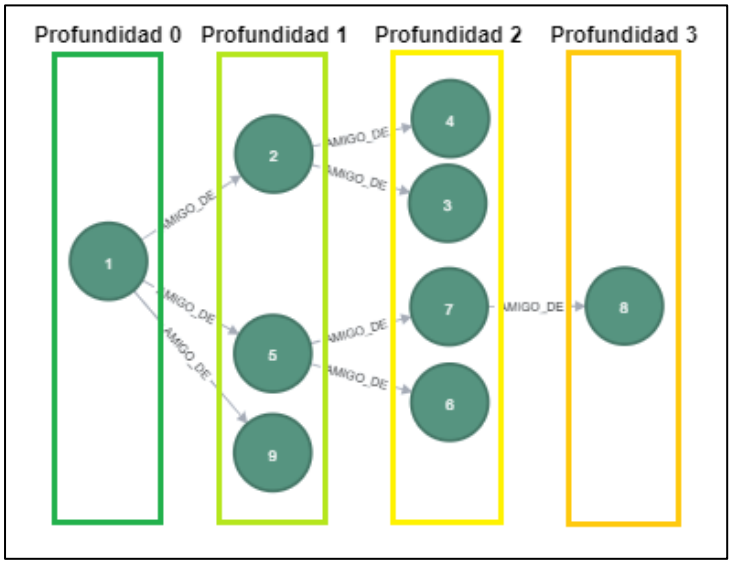


Figura 24 *- Grafo dividido en niveles de profundidad.*

Para realizar una búsqueda en profundidad, se utiliza el símbolo *asterisco* para determinar el nivel de profundidad dentro de las relaciones. En los ejemplos que se muestran a continuación, el nodo de partida será el *nodo 1*, véase **Figura 24**. Además, cuando se habla de niveles, se está hablando realmente del número de saltos.

En el **Código 7** se muestra una búsqueda en la que se quiere obtener todos los nodos que estén a un nivel de profundidad del nodo origen, es decir, todos los nodos accesibles mediante un único salto entre aristas del tipo *AMIGO_DE*. En este caso se obtendrían todos los nodos del nivel de profundidad 1, véase **Figura 24**. En el modelo de *Twitter*, es equivalente a obtener los amigos de un usuario concreto. Cuando se quiere realizar un único salto no es necesario introducir el asterisco junto con el nivel de profundidad.

```
MATCH (:Usuario {nickname: "@eva19"})-[relaciones:AMIGO_DE]-(amigos:Usuario)
RETURN relaciones, amigos
```

Código 7 *- Búsqueda en profundidad de nivel 1.*

El caso del **Código 8**, es similar al anterior, con la diferencia que en éste también se añade la profundidad 0, es decir, también se añade el usuario de partida (*@eva19*). Como se quiere hacer una búsqueda de más de un nivel (salto), se hace uso del símbolo asterisco junto con el rango de niveles **minNivel..maxNivel* .

```
MATCH (:Usuario {nickname: "@eva19"})-[relaciones:AMIGO_DE*0..1]-(amigos:Usuario)
RETURN relaciones, amigos
```

Código 8 *- Búsqueda en profundidad de nivel 0 y 1.*

Para obtener todo el círculo de amigos y los amigos de los amigos de un usuario, se utiliza una búsqueda en profundidad desde el nivel 0 hasta el nivel máximo existente, véase **Código 9**. En este caso, se empezaría obteniendo los nodos de nivel 0, luego los de nivel 1, posteriormente los del nivel 2 y así hasta el nivel máximo, que en este caso sería el nivel 3, véase **Figura 24**. Para ello, no hace falta definir el nivel máximo en el rango de niveles.

```
MATCH (:Usuario {nickname: "@eva19"})-[relaciones:AMIGO_DE*0..]-(amigosYconocidos:Usuario)
RETURN amigosYconocidos
```

Código 9 *- Búsqueda desde nivel 0 hasta el nivel máximo.*

Para realizar una búsqueda con un número concreto de saltos, se especifica de la siguiente manera, véase **Código 10**. En este caso solo se mostrarían los nodos de la profundidad 2 de la **Figura 24,** es decir, aquellos nodos que están a dos saltos del nodo origen.

```
MATCH (:Usuario {nickname: "@eva19"})-[relaciones:AMIGO_DE*2]-(amigosDeAmigos:Usuario)
RETURN relaciones, amigosDeAmigos
```

Código 10 *- Búsqueda en un nivel 2 de profundidad.*

En definitiva, el lenguaje *Cypher* permite realizar diferentes búsquedas dando una cantidad determinada de saltos de una forma rápida, sencilla y gráfica. En los casos mostrados se han utilizado saltos de relaciones con una etiqueta en concreto (:AMIGO_DE), sin embargo, esto también es aplicable a relaciones sin etiquetas.

Por otra parte, se encuentra la cláusula **OPTIONAL MATCH**, que en el lenguaje *SQL* es equivalente a la cláusula OUTER JOIN.

Para el caso de la **Figura 25**, con la consulta del **Código 11**, se obtiene el nombre de los usuarios que tienen al menos un amigo, junto con el nombre del amigo. En el caso de que un usuario no tuviese amigos no saldría en la tabla (como es el caso del usuario 10), véase el resultado del **Código 11**.

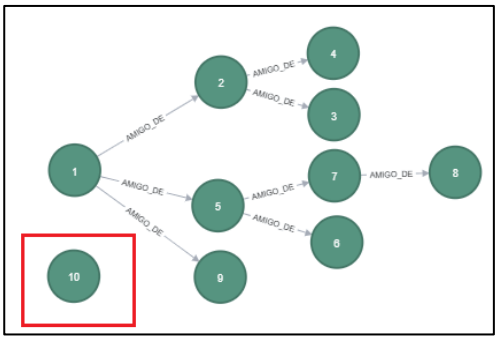


Figura 25 *- Ejemplo de un grafo de nodos de usuarios y relaciones de amigos*

Sin embargo, si se quiere obtener el nombre de todos los usuarios junto con sus amigos independientemente de si tiene amigos o no, se debe utilizar **OPTIONAL MATCH**, véase **Código 12**.

En esta última consulta, con la primera parte de la misma *(u1:Usuario)* , se obliga a que los nodos cumplan este mismo patrón. No obstante, con la segunda parte de la consulta **OPTIONAL MATCH**, puede que los nodos tengan alguna coincidencia con el patrón *(u1)-[:AMIGO_DE]-(u2:Usuario)* o puede que no los tengan. Luego, en el caso de que el nodo *u1* cumpliese con el patrón *(u1)-[:AMIGO_DE]-(u2:Usuario)*, en la variable *u2* se almacenaría el resultado, en caso contrario, la variable *u2* se establecería a *NULL*.

```
MATCH (u1:Usuario)-[:AMIGO_DE]-(u2:Usuario)
RETURN u1.Id AS usuario, collect(u2.Id) AS amigos
```

"usuario"	"amigos"
1	[9,5,2]
2	[4,3,1]
3	[2]
4	[2]
5	[7,6,1]
6	[5]
7	[8,5]
8	[7]
9	[1]

Código 11 *- Ejemplo de la cláusula MATCH y resultado.*

```
MATCH (u1:Usuario)
OPTIONAL MATCH (u1)-[:AMIGO_DE]-(u2:Usuario)
RETURN u1.Id AS usuario, collect(u2.Id) AS amigos
```

"usuario"	"amigos"
1	[9,5,2]
2	[4,3,1]
3	[2]
4	[2]
5	[7,6,1]
6	[5]
7	[8,5]
8	[7]
9	[1]
10	[]

Código 12 *- Ejemplo de OPTIONAL MATCH y resultado.*

Modificación de nodo, arista, propiedad o etiqueta

Para modificar un dato de un elemento existente, se debe utilizar la cláusula **SET** combinándolo con **MATCH**. Se utilizará **MATCH** para obtener el elemento o el conjunto de elementos que se quiere modificar y **SET** para determinar el cambio que se quiere realizar.

En el primer comando del **Código 13** se modifica el atributo *activo* para todos los nodos con la etiqueta *Usuario*. En el segundo comando, la actualización se centra en un solo nodo y se modifican dos propiedades.

```
MATCH (usuarios:Usuario)
SET usuarios.activo = False

MATCH (usuarioACambiar:Usuario {nickname: "@eva19"})
SET usuarioACambiar.nickname = "@eva", usuarioACambiar.nombre = "Eva María"
```

Código 13 *– Modificación de propiedades en un conjunto de nodos y en un nodo.*

Se sigue el mismo procedimiento para actualizar una propiedad de una relación, véase el **Código 14**.

```
MATCH (:Usuario {nickname: "@eva"})-[relacionACambiar:TIENE_VINCULADO]->(:Correo)
SET relacionACambiar.fecha = 04/05/2019
```

Código 14 *- Actualización de una propiedad de una relación.*

Si se quiere clonar un nodo, es decir, copiar todas las propiedades de un nodo en concreto, se realiza mediante: `SET nodoDestino = nodoACopiar`

Finalmente, para añadir una o varias etiquetas a los nodos o relaciones, se debe lanzar la siguiente consulta. En ésta se han añadido dos nuevas etiquetas al usuario *@ivan*.

```
MATCH (usuarioACambiar:Usuario {nickname: "@ivan"})
SET usuarioACambiar:Administrador:Tecnico
```

Código 15 *- Consulta para añadir dos nuevas etiquetas.*

Por otro lado, se encuentra la cláusula **MERGE**, la cual, verifica si el patrón especificado existe y coincide con la BDG. Si coincide extrae el patrón como lo haría la cláusula **MATCH**. Si no encuentra coincidencias, crea el patrón especificado como lo haría la función **CREATE**. Además, permite la opción de realizar diferentes acciones dependiendo del camino tomado. Mediante **ON MATCH SET**, se puede establecer una o más propiedades o etiquetas en los nodos y relaciones extraídos. A través de **ON CREATE SET** se puede establecer propiedades y etiquetas en los nodos y relaciones creados.

```
MERGE (u:Usuario {nickname: "@ivan"})
ON CREATE SET u.nickname = "@ivan_new"
ON MATCH  SET u.nickname = "@ivan_update"
```

Código 16 *- Ejemplo de la cláusula MERGE en Cyoher.*

Eliminación de nodo, arista, propiedad o etiqueta

El lenguaje *Cypher* dispone de dos clausulas para llevar a cabo la operación de eliminación: **REMOVE** y **DELETE**.

REMOVE: Establecida para la eliminación de etiquetas y propiedades.

DELETE: Establecida para eliminar nodos y relaciones.

En el **Código 17** se reflejan las tres consultas más comunes con la cláusula **REMOVE**. En la primera y en la segunda se elimina una propiedad de un nodo y de una relación respectivamente. En la tercera, se elimina una etiqueta de un nodo.

```
MATCH (u:Usuario {nickname: "@eva"})
REMOVE u.nickname

MATCH (:Usuario {nickname: "@eva"})-[relacion:TIENE_VINCULADO]->(:Correo)
REMOVE relacion.fecha

MATCH (u:Usuario {nickname: "@ivan"})
REMOVE u:Tecnico
```

Código 17 *- Consultas con la cláusula REMOVE.*

Finalmente, en la siguiente consulta se refleja la operación de eliminación de un nodo usuario junto con sus relaciones y los nodos relacionados (**DELETE**). Si se quiere eliminar un nodo que contiene relaciones, también se deberán eliminar estas relaciones. Si no es así, *Neo4j* lanzará un error.

```
MATCH (u:Usuario {nickname: "@ivan"})-[r]->(n)
DELETE u,r,n
```

Código 18 *- Eliminación de nodos y relaciones.*

Funciones de agregación

En *Cypher*, al igual que en *SQL*, se dispone de diferentes funciones de agregación que se muestran a continuación [13]:

- **COUNT** : Cuenta las filas o valores nulos.
- **SUM** : Ejerce la suma de las propiedades numéricas de los nodos resultantes.
- **AVERAGE** : Realiza la media de las propiedades numéricas de los nodos resultantes.
- **MAX** | **MIN** : Obtiene el máximo o el mínimo de un conjunto de números.
- **STDEV** : Para realizar la derivación estándar.
- **COLLECT** : Para almacenar todos los valores en un array.

Combinación de resultados

Para combinar dos resultados de dos consultas diferentes, se utiliza la cláusula **UNION** de la siguiente forma. Supongamos que se quiere obtener el número de usuarios y el número de correos de la base de datos. Primero se realiza la consulta para los usuarios y luego se combina con la consulta de los correos mediante **UNION**, véase **Código 19**.

```
MATCH (u:Usuario)
RETURN "Usuarios" as tipo, COUNT(u) as cantidad
UNION ALL
MATCH (c:Correo)
RETURN "Correos" as tipo, COUNT(c) as cantidad
```

"tipo"	"cantidad"
"Usuarios"	3
"Correos"	3

Código 19 *- Clausula UNION junto al resultado.*

Finalmente, existe la cláusula **WITH** que sirve para separar diferentes partes de una consulta. Por ejemplo, se quiere obtener los amigos del usuario que tenga la mayor cantidad cuentas de correo vinculadas, véase **Código 20**. Primero se obtienen todos los usuarios junto con la cantidad de cuentas de cada uno. Seguidamente, todos los usuarios se ordenan descendientemente según la cantidad de cuentas y se selecciona el primero de ellos (**LIMIT 1**). Una vez realizado esto, la consulta seguirá sólo para este usuario extraído, del cual se conseguirán los amigos.

```
MATCH (u:Usuario)-[:TIENE_VINCULADO]->(c:Correo)
WITH u, COUNT(c) as cantidad
ORDER BY cantidad DESC
LIMIT 1
OPTIONAL MATCH (u)-[:AMIGO_DE]->(amigos:Usuario)
RETURN u, COLLECT(amigos)
```

Código 20 *- Ejemplo de consulta con la cláusula WITH.*

Índices y restricciones

Los únicos índices que soporta *Neo4j* son los **índices de etiqueta** y los **índices de propiedades** [13]. Por una parte, al asignar una etiqueta a un nodo, internamente *Neo4j* establece un índice para la misma. Por otra parte, *Neo4j* ofrece la oportunidad de crear índices en las propiedades para un acceso más rápido. Para crear un índice en una determinada propiedad se debe lanzar el siguiente comando:

```
CREATE INDEX ON :Usuario(nickname)
```

Asimismo, *Neo4j* también ofrece tres tipos de restricciones [14]. No obstante, dos de ellas solo están disponibles en la edición *enterprise* de *Neo4j*:

- **Restricciones de unicidad de propiedades (*Unique property constraints*):**

  Estas restricciones sirven para asegurar que los valores de ciertas propiedades son únicos para todos los nodos con una etiqueta especifica. No obstante, todos los nodos de la misma etiqueta no están obligados a disponer de esta propiedad [14].

```
CREATE CONSTRAINT ON (u:Usuario) ASSERT u.nickname IS UNIQUE
DROP   CONSTRAINT ON (u:Usuario) ASSERT u.nickname IS UNIQUE
```

Código 21 *– Creación y eliminación de una restricción de unicidad.*

- **Restricción de existencia de propiedad (*Property existence constraints*)**

  [EDICIÓN ENTERPRISE]

  Este tipo de restricción garantiza que existe una propiedad para todos los nodos de una etiqueta especifica o para todas las relaciones de un tipo concreto. Si se intenta crear un nodo o relación sin la propiedad o si se quiere eliminar la propiedad que tiene este tipo de restricción, *Neo4j* lanza un error [14].

```
CREATE CONSTRAINT ON (u:Usuario) ASSERT EXISTS(u.activo)
DROP   CONSTRAINT ON (u:Usuario) ASSERT EXISTS(u.activo)
```

Código 22 *– Creación y eliminación de una restricción de existencia de propiedad.*

- **Claves de nodos (*Node Keys*)**

  [EDICIÓN ENTERPRISE]

  Este tipo de restricción asegura que para cierta etiqueta y un conjunto de propiedades [14]:

  - El conjunto de propiedades existe en todos los nodos con la etiqueta especifica.
  - La combinación de los valores de las propiedades es única.

```
CREATE CONSTRAINT ON (u:Usuario) ASSERT (u.nickname, u.nombre, u.apellidos) IS NODE KEY
DROP   CONSTRAINT ON (u:Usuario) ASSERT (u.nickname, u.nombre, u.apellidos) IS NODE KEY
```

Código 23 *- Creación de restricción NODE KEY.*

Cuando se crea una restricción de unicidad (*unique constraint*) sobre una propiedad, *Neo4j* automáticamente establece un índice en esa propiedad. Asimismo, al crear una restricción *Node Key* sobre un conjunto de propiedades, *Neo4j* crea un índice sobre el conjunto de propiedades.

Migración de datos

Una de las cláusulas más significativas para este proyecto es **LOAD CSV**. Mediante este comando se consigue cargar e importar todos los datos desde un fichero *.csv* a la base de datos de *Neo4J*. Los ficheros *csv* pueden tener una ubicación local o se puede acceder mediante una *URL*.

```
LOAD CSV WITH HEADERS FROM 'URL o direccionLocal' AS csvLine
CREATE (:Usuario {nickname:csvLine.nickname, nombre:csvLine.nombre, apellidos:csvLine.apellidos})
```

Código 24 *- Importación de datos desde ficheros CSV.*

Capítulo 4 – Diseño, modelo y generación de datos

La finalidad de este capítulo es determinar cómo se ha llevado a cabo el proceso del diseño, modelado y generación de datos que vamos a utilizar durante el desarrollo de este proyecto.

4.1 – Diseño y modelo

Actualmente, los empleados del departamento de Gestión de Deudas utilizan muchas bases de datos relacionales para la investigación de las derivaciones. Estas bases de datos se encuentran muy dispersas y, aunque almacenan información que se puede cruzar, no toda la información se comunica ni es tan directa. Precisamente, es una de las necesidades que se quiere cubrir con las bases de datos orientadas a grafos, el poder centralizar toda la información relacionada en un mismo grafo.

Por otra parte, en el presente proyecto no se han tenido en cuenta todas las entidades, relaciones y atributos existentes en las mismas. Se trata de un modelo específico para los casos de usos solicitados, no es un modelo extendido para todos los casos de uso de la Hacienda Tributaria de Navarra.

Además, se trata de un modelo relacional en el que la información que se va a presentar a continuación realmente existe, pero se presenta dispersa por las diferentes bases de datos disponibles. Es por ello que idealmente el modelo relacional se figuraría de la siguiente forma, aunque en la realidad no es todo tan sencillo.

Para este modelo se presentan diferentes entidades que se relacionan entre sí como pueden ser:

- **Personas físicas**

  Según el diccionario español jurídico se entiende como persona física a *aquel individuo con capacidad jurídica para ser titular de derechos y cumplir con obligaciones* [15].

  En el modelo actual estas personas son identificadas unívocamente mediante el *Número de Identificación Fiscal* (NIF).

- **Personas jurídicas**

  Según la definición del diccionario español jurídico, una persona jurídica es una *institución dotada de personalidad propia e independiente y plena capacidad para el cumplimiento de sus fines, creada por las leyes o conforme a lo establecido en las mismas* [16].

  Estas entidades también son identificadas mediante el NIF correspondiente.

- **Modelos**

  Se le llama *modelo*, a los modelos tributarios que son utilizados para informar a Hacienda de la actividad económica de las personas físicas o jurídicas. Existen modelos para saldar los impuestos y para informar de las acciones realizadas durante un periodo de tiempo (salarios, cuentas bancarias, etc.).

  De esta entidad se almacenará la clave identificadora del modelo junto con el importe del mismo y un número representativo del tipo de modelo (e.g. Modelo 180).

- **Declaraciones**

  Se conoce como declaración a la *comunicación de un sujeto privado concerniente a una determinada actuación regulada por las normas o precisada por la Administración para su control* [17].

  En las declaraciones se precisa de un identificador, el importe correspondiente y el tipo de declaración.

- **Deudas**

  Según el diccionario español jurídico una deuda tributaria es una *cantidad a ingresar en la Hacienda pública como consecuencia de la obligación tributaria principal y de los pagos a cuenta que está integrada por la cuota y, en su caso, el interés de demora, los recargos por declaración extemporánea, los recargos del período ejecutivo y los recargos exigibles sobre la base o sobre la cuota a favor del Tesoro o de otros entes públicos* [18].

  En este tipo de entidad se almacena el identificador junto con el importe.

- **Direcciones, teléfonos y móviles.**

  Otras de las entidades de este proyecto son las direcciones, los teléfonos fijos y los móviles. Las direcciones son identificadas mediante la referencia catastral de la misma. Las dos restantes se identifican mediante el propio número, puesto que es único.

Asimismo, las entidades detalladas se relacionan entre sí y, además, pueden ser de diferentes tipos como:

- **Relaciones familiares**

Las personas físicas se pueden relacionar con otras personas físicas de diferentes formas. Por una parte, se encuentran las relaciones entre padres e hijos. Por otra parte, se encuentra la relación entre cónyuges. Y finalmente, se encuentran las relaciones de asistidos. Una persona asistida es aquella que depende de otra persona física sea cual sea el motivo (enfermedad, minusvalías, etc.).

- **Relaciones empresariales**

  En cuanto a las relaciones empresariales, existen dos tipos de relaciones.

  En primer lugar, una persona jurídica puede tener una participación en otra persona jurídica. A la primera se le conoce como empresa *dominante*, mientras que a la segunda se le conoce como empresa *dominada*.

  Y, en segundo lugar, una persona física puede ser socio o administrador de una o más personas jurídicas. En caso de ser socio, por cada persona existe un porcentaje de participación.

- **Relación de posesión**

  Una persona física puede disponer o no de direcciones, teléfonos fijos y móviles. Además, varias personas físicas pueden relacionarse con una misma dirección o teléfono fijo. Esto puede dar indicios para detectar posibles fraudes en un futuro.

  En el caso de las direcciones, una persona física puede indicar que *reside* en una de ellas o también puede *tener derecho* sobre una de ellas (alguien que alquila, usufructos, etc.)

- **Relación entre deuda y persona física o jurídica**

  Una persona tanto física como jurídica puede contraer una o más deudas.

- **Relación entre declaración y persona física o jurídica**

  Una persona física o jurídica puede relacionarse de tres modos con una declaración.

  En primer lugar, se encuentran aquellas personas que son *causantes* de una declaración. Unos de los ejemplos más significativos de este tipo de relación son las herencias, los causantes serán aquellas personas que provocan esa herencia.

  En segundo lugar, se encuentra las personas que disponen o heredan un bien que deben declarar. También se conocen como *sujetos pasivos*.

En tercer lugar, existen las personas *transmitentes* de declaraciones. Un ejemplo de este último serían aquellas personas que realizan una donación.

- **Relación entre modelos y persona física o jurídica**

  Tanto a las personas físicas como a las personas jurídicas, se les imputan modelos que ayudan a Hacienda a tener controlado todo tipo de acciones realizadas por las personas. Además, estos modelos son imputados por diferentes personas jurídicas.

4.2 – Transformación a un modelo orientado a grafos

Una vez se ha estudiado el modelo actual de la base de datos relacional del departamento de Gestión de Deudas, lo siguiente a realizar es la transformación de ese modelo a un modelo orientado a grafos.

Lo primero que se debe tener en cuenta es el tipo de modelo de grafo que se quiere implementar. Como bien se ha estudiado en el *Capítulo 3,* la base de datos seleccionada para este proyecto ha sido *Neo4j.* En ella se implementa un modelo de grafo LPG, (Labeled Property Graph), lo que supone que existirán nodos y relaciones que podrán ser etiquetadas y podrán tener un conjunto de propiedades.

Todas las entidades identificadas en el modelo relacional (*Apartado 4.1*) pasarán a ser nodos con sus correspondientes etiquetas:

- **Nodo con etiqueta *Persona_fisica***

  Existirá un tipo de nodo para las personas físicas que dispondrá de una propiedad para el NIF de tipo *String*.

- **Nodo con etiqueta *Persona_juridica***

  Se establecerá un tipo de nodo para las personas jurídicas que, al igual que las físicas, establecerá una propiedad NIF de tipo *String*.

- **Nodo con etiqueta *Declaración***

  Las declaraciones se figurarán mediante otro tipo de nodo junto con sus tres propiedades: *id* , *tipo* e *importe*.

- **Nodo con etiqueta *Modelo***

  Los modelos seguirán el mismo procedimiento que el de las declaraciones, pero con sus correspondientes atributos: *id*, *modelo* (número del modelo) e *importe*.

- **Nodo con etiqueta *Deuda***

  Se dispondrá de otro tipo de nodo para las deudas con dos propiedades: *id* e *importe*.

- **Nodo con etiqueta *Direccion***

  En este tipo de nodo solo se necesitará la propiedad *direccion* que será un *String* para la referencia catastral.

- **Nodo con etiqueta *Telefono_fijo***

  Los teléfonos fijos se representarán mediante otro tipo de nodo con una propiedad *numero.*

- **Nodo con etiqueta *Telefono_movil***

Finalmente, existirá otro tipo de nodo con una propiedad *numero* para los móviles existentes.

Por otro lado, todas las relaciones identificadas anteriormente serán representadas mediante aristas con una etiqueta específica:

- Arista con etiqueta *ES_ASCENDENTE*[5] para representar las relaciones padre-hijo.
- Arista con etiqueta *ES_CONYUGE* para las relaciones de parejas.
- Arista con etiqueta *ES_ASISTIDO_POR* para representar las relaciones de las personas asistidas y las personas que asisten.
- Arista con etiqueta *TIENE_DERECHO* para determinar que una persona tiene derechos sobre una dirección.
- Arista con etiqueta *HA_INDICADO* que representa que una persona reside en una dirección.
- Arista con etiqueta *DOMINA* que figura la participación de una persona jurídica en otra persona jurídica.
- Arista con etiqueta *TIENE_ADMINISTRADOR* que representa que una persona jurídica dispone de un administrador (persona física).
- Arista con etiqueta *TIENE_SOCIO* que se utiliza para determinar los diferentes socios de una persona jurídica. Esta relación dispone de una propiedad *porcentaje* para el porcentaje de participación.
- Arista con etiqueta *IMPUTA* que representa que una persona jurídica imputa un modelo. Es decir, ofrece una declaración informativa a Hacienda.
- Arista con etiqueta *ES_IMPUTADO* que representa que a una persona física se le asigna un modelo.
- Arista con etiqueta *ES_CAUSANTE* para representar que una persona física o jurídica ha sido causante de una declaración.
- Arista con etiqueta *ES_SP* que se utiliza para representar los sujetos pasivos de las declaraciones.
- Arista con etiqueta *ES_TRASMITENTE* que representa las transmisiones de declaraciones de una persona física o jurídica.
- Arista con etiqueta *TIENE_CONTRAIDA* que sirve para determinar que una persona física o jurídica contrae una deuda.

[5] En *Neo4j* se siguen unas reglas de estilo para todos los proyectos. Las etiquetas de las relaciones estarán en mayúsculas, mientras que las etiquetas de los nodos, solo la primera letra será mayúscula.

En la **Figura 26** se representa el resultado de transformar el modelo relacional a un modelo orientado a grafos. Este modelo será el que se tenga de referencia para todo el proyecto restante. En esta figura los nombres que aparecen en los nodos son las etiquetas de los nodos, mientras que los nombres de las aristas son las etiquetas de las aristas.

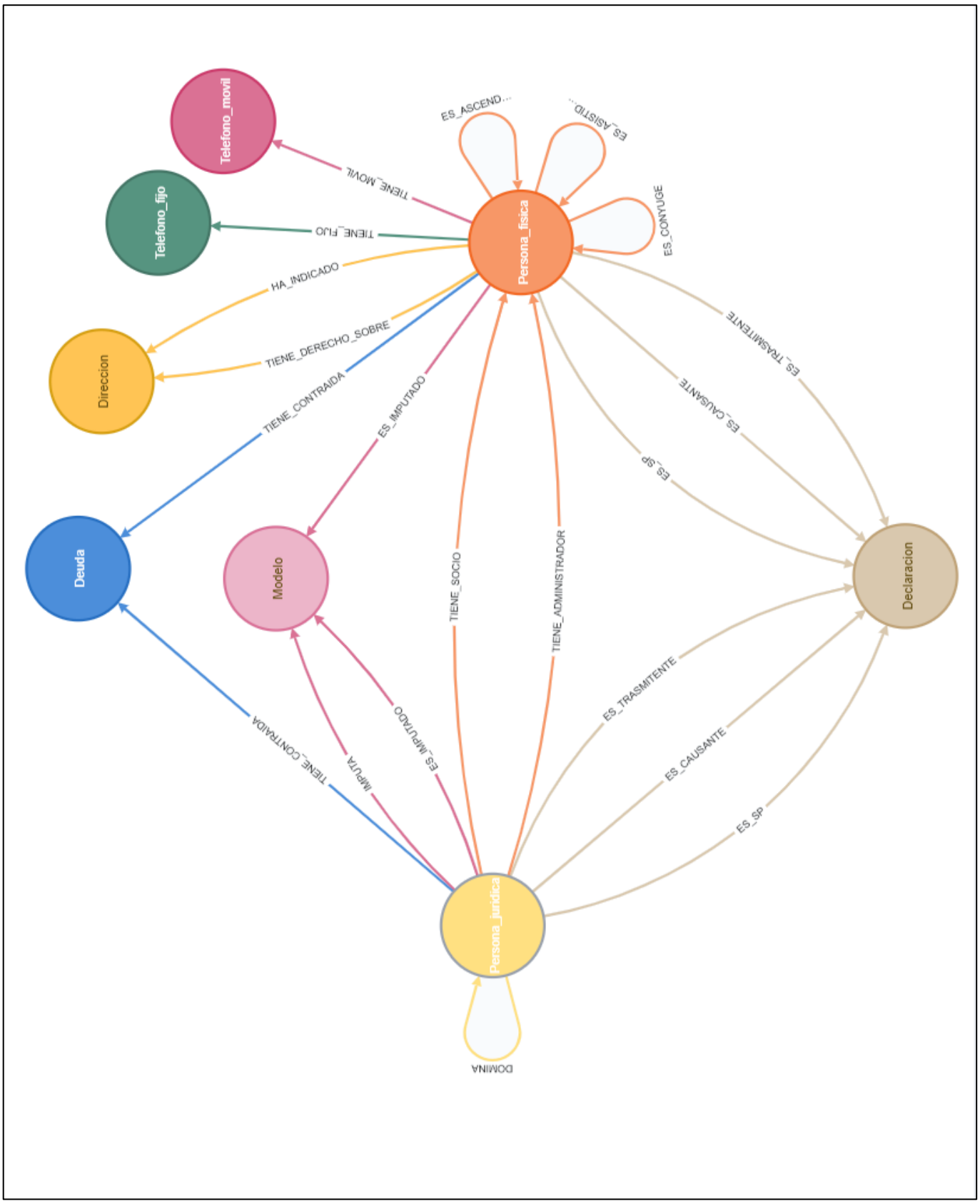


Figura 26 *– Modelo orientado a grafo.*

4.3 – Obtención de datos

Unos de los objetivos principales de este proyecto es la extracción y la importación de los datos obtenidos de las diferentes referencias utilizadas en el proceso de investigación. Estos datos, son datos oficiales de la Hacienda Tributaria de Navarra, disponibles por la empresa Tracasa debido a un convenio entre las dos entidades.

Dada la importancia y la confidencialidad de los mismos, se debe cumplir con el Reglamento General de Protección de Datos (RGPD). Para el tratamiento de los datos, tanto el director de este proyecto, como el autor, deben de haber firmado diferentes contratos de protección, los cuales, deben de pasar un proceso de validación y consentimiento por parte de Tracasa y por parte del Gobierno de Navarra. No obstante, este proceso es lento, por lo que se ha decidido generar datos sintéticos lo más parecidos posibles a los datos reales.

Esta generación de datos se ha llevado a cabo de una forma aleatoria, pero obteniendo resultados que reflejen la realidad lo más fielmente posible. Si este proceso se considerase totalmente aleatorio, se obtendrían datos incoherentes e inservibles para los estudios de los diferentes supuestos y casos de uso. Es por ello, se ha decidido generar datos aleatorios, que se ajusten perfectamente al modelo especificado, pero con ciertas restricciones para obtener un conjunto de datos congruente.

Los datos empleados en el Trabajo Fin de Grado son datos sintéticos, dado que no se quiere incumplir la RGPD y, además, gran parte del trabajo ha sido desarrollado con estos datos, ya que, no han llegado a tiempo los permisos correspondientes. No obstante, como se ha conseguido que sean los más afines a los datos reales, la utilización de uno u otro no supondría gran diferencia. El único inconveniente del mismo sería la obtención de patrones de los datos. Al generar datos aleatorios, la probabilidad de existencia de patrones útiles para los casos de uso disminuye.

4.3.1 – Proceso de generación de datos sintéticos

El proceso de generación de datos aleatorios se ha llevado a cabo con el lenguaje *Python* junto con las librerías *NumPy* y *Pandas* y el entorno interactivo *Jupyter Notebook*. El conjunto de datos creados se almacena en diferentes ficheros *csv* que, posteriormente, serán utilizados para importar los datos a *Neo4j*.

Los módulos implementados se han intentado realizar lo más parametrizables posibles para poder generar diferentes muestras de datos, con diferentes tamaños y diferentes probabilidades de ocurrencias de las relaciones.

Este sistema se divide en dos grandes bloques. Por una parte, se encuentra el bloque que genera un conjunto de datos necesarios para el siguiente bloque. El segundo bloque, es el que mantiene la lógica de negocio para la generación de los datos.

En el primer bloque se generan cinco conjuntos de datos que resultan en cinco ficheros *csv*. Estos ficheros, posteriormente, serán utilizados por el segundo bloque. Con este bloque se consigue generar aleatoriamente dos conjuntos de NIFs de personas físicas y jurídicas respectivamente, un conjunto de referencias catastrales (direcciones) y dos conjuntos para los números de teléfonos fijos y móviles. La única restricción utilizada en este bloque es que los elementos creados no se repitan. Además, se puede especificar la cantidad de elementos a crear por cada conjunto.

La funcionalidad de este segundo bloque es más compleja, tanto por la extensión del mismo, como por las diversas restricciones e implementaciones. Inicialmente se cargan todos los conjuntos de datos creados anteriormente el primer bloque, y se almacena en diferentes listas de *Python*.

Existen diferentes módulos en este bloque: generación de relaciones entre personas físicas (familiares, cónyuges y asistidos), generación de relaciones de socios y administradores, asignación de direcciones, teléfonos fijos y móviles a personas físicas, generación de modelos e imputaciones a los dos tipos de personas, generación de declaraciones y las diferentes relaciones de éstas con las personas y, finalmente, la generación de deudas y relaciones.

Para perseverar una coherencia en todo el conjunto de datos, se ha considerado dividir la lista de personas físicas en diferentes generaciones. De esta manera se puede conseguir un resultado más realista, y se puede asignar en diferentes módulos diferentes probabilidades de concurrencia según la generación a la que pertenece una persona.

Para este caso, se han conjeturado cuatro generaciones: primera generación (edad superior a 60 años), segunda generación (edad entre 30-60 años), tercera generación (edad entre 18-30 años) y la cuarta generación (menores de edad).

En cuanto a las relaciones de parejas, una persona física solo puede ser pareja de otra persona física de su misma generación. En las relaciones *padre-hijo*, una persona física solo puede tener hijos de una generación inferior. Y en las relaciones de asistidos, se ha considerado la restricción de que un padre no pueda asistir a su hijo, pero sí al revés. Además, dependiendo de la generación, la probabilidad de que sea asistido difiere.

En las personas jurídicas se ha seguido un proceso similar. Se ha divido las personas jurídicas en dos conjuntos. Por una parte, se encuentran las empresas matrices que no tienen ninguna empresa dominante, pero si pueden tener empresas dominadas. Y por el otro lado están las empresas que pueden ser dominantes y dominadas. La relación de socios/administradores entre las personas físicas y jurídicas es simple, como en todos los casos, dependiendo de la generación, la probabilidad de ser socio o administrador varía. Además, se ha añadido que, si una persona dispone de una pareja, y esta pareja es socio o administrador de una empresa en concreto, la persona que se está estudiando tiene mayor probabilidad de ser socio o administrador de la empresa de la pareja. Es

importante recalcar, que las personas físicas de última generación no pueden ser socios, ni administradores, puesto que son menores de edad.

Para asignar las direcciones se ha tenido en cuenta dos hechos: las direcciones de los padres y las direcciones de la pareja. La asignación se verá influenciada por la probabilidad de que se le asigne la misma dirección de la pareja o que se le asigne la misma dirección que la de sus padres. Como es obvio, puede darse el caso de que no comparta dirección o que la comparta con una persona sin ninguna relación directa con ella. Si se tiene pareja, la probabilidad de compartir dirección con la pareja es mayor que la de compartir con los padres. Aún y todo, como bien se ha comentado anteriormente, estas probabilidades pueden ser modificadas en cualquier momento. Las personas de última generación se le asignará la misma dirección que las de sus padres.

El teléfono fijo se encuentra vinculado a la dirección. Puede darse el caso de que una dirección no tenga teléfono fijo. Finalmente, la asignación de los móviles es totalmente aleatoria, por cada generación se disponen de una probabilidad de tener móvil o no.

Para finalizar con la explicación, tanto para las deudas, declaraciones y modelos se ha seguido un proceso similar. La asignación es dependiente de las diferentes probabilidades según las generaciones y se puede asignar un máximo de deudas, declaraciones o modelos para cada persona. En las declaraciones, como existen tres tipos de relaciones, por cada relación y por cada generación se tienen diferentes probabilidades.

Todo este proceso explicado concluye en diferentes ficheros *csv* que se utilizarán en capítulos posteriores.

4.3.1.1 - *Contratiempos y mejoras del sistema*

La principal complicación sufrida durante el desenlace de este sistema generador de datos ha sido el tiempo de respuesta de los diversos módulos explicados.

En una primera implementación del sistema, se consideró la utilización de listas (*list*) y conjuntos (*set*) estándares de *Python* para el procesamiento de datos. Tanto los datos que se iban generando como los diferentes elementos necesarios, se iban almacenando en listas y conjuntos que eran recorridos por diferentes estructuras iterativas (*for*, *while*, etc.).

Esta solución resultó provechosa para un primer contacto en el que se deseaba crear un conjunto de datos reducido. Sin embargo, a medida que se iba avanzando en la prospección de la base de datos y en el proyecto, era necesario crear un conjunto de datos mayor. Esto provocaba un aumento exponencial del tiempo de espera del sistema. Para la generación de conjuntos de datos considerables (unidades de millón) calculamos tiempos de espera en unidades de días, por lo que se decidió realizar una reestructuración del sistema implementado.

En esta segunda implementación se reemplazaron las estructuras estándares de listas y conjuntos por matrices de la librería *NumPy.* Esta librería ofrece diferentes métodos y técnicas más óptimas para el filtrado y recorrido de las matrices.

Además, la mayoría de las búsquedas y los filtrados que se realizaban, se aplicaban sobre diferentes columnas de una matriz. *Numpy* por defecto sigue la técnica *row-major order* en las matrices, es decir, los elementos de una misma línea se encuentran en posiciones adyacentes en memoria. Sin embargo, en nuestro caso se realizaban las búsquedas sobre las columnas de las matrices, por lo que era más oportuno que la técnica fuese *column-major order*, es decir, los elementos de una misma columna están en posiciones adyacentes en memoria. De esta manera, los saltos en memoria que se deben dar para recorrer una columna o filtrarla son pequeños, ya que, se encuentran en posiciones vecinas. Para establecer esta filosofía en las matrices *Numpy* se utilizó la función *asfortranarray()* que convertía una matriz *Numpy* con filosofía *row-major order* a una matriz *Numpy* con una filosofía *column-major order* (concretamente *Fortran*).

Asimismo, para optimizar aún más todo el sistema generador de datos, se utilizó el compilador *JIT* de *Numba* para la optimización de diferentes funciones implementadas.

Con todas estas modificaciones se logró reducir el tiempo computacional considerablemente, siendo aproximadamente 14 veces más rápido. Por ejemplo, para un mismo conjunto de millones de datos, la primera solución lo realizaba en dos días, mientras que está última implementación lo realizaba en tres o cuatro horas.

Capítulo 5 – Importación de datos

En el capítulo anterior, se ha realizado una explicación de cómo se ha conseguido el modelo orientado a grafos que se va a implementar y se ha descrito todo el proceso de obtención y generación de los datos. Ahora bien, se debe importar estos mismos datos en la base de datos de *Neo4j*. Para ello, en este capítulo se va a explicar cómo se ha llevado a cabo esta importación de datos y se detallarán los aspectos necesarios para la misma.

5.1 – Configuración y entornos de Neo4j

Antes de comenzar directamente con la importación de los datos, es preciso determinar diferentes aspectos en cuanto a la configuración y los entornos de *Neo4j*. Como bien se ha comentado en capítulos anteriores, *Neo4j* dispone de dos tipos de suscripciones, lo que concluye en dos tipos de entornos diferentes.

Por una parte, se ha creado un entorno para la edición *Community* de *Neo4j* (versión gratuita), y por otra parte, se ha creado otro entorno diferente para la versión de pago *Enterprise*. No obstante, en mi caso se ha conseguido una licencia de desarrollador que permite utilizar la versión *Enterprise* sin ningún coste, siempre y cuando no se utilice para fines comercialices.

Estos dos entornos mencionados se encuentran sobre un mismo servidor en diferentes contenedores. Es importante que se encuentren sobre el mismo servidor (hardware) para poder comparar en las mismas condiciones los dos tipos de suscripciones de *Neo4j*.

Neo4j ofrece la oportunidad de poder desplegar la base de datos sobre contenedores *Docker*. En *Docker Hub* se pueden obtener las imágenes oficiales de los contenedores. Una vez se tiene desplegadas las dos instancias se pueden acceder a ellas por el puerto que se haya establecido.

Por otra parte, *Neo4j Enterprise* te da la oportunidad de obtener una aplicación de escritorio en donde poder crear diferentes proyectos (bases de datos) en local. Desde la misma, también se puede acceder remotamente a diferentes instancias lanzadas.

Los dos puertos que habilita *Neo4j* por defecto son el 7474 y el 77687. El primero para desplegar un navegador, desde el cual se pueden ejecutar consultas y visualizar los resultados obtenidos, y el segundo sirve para acceder a la base de datos remotamente. Por ejemplo, vía la API de Python, con la cual hay que conectarse a este último puerto. Estos puertos son configurables y pueden ser modificados en cualquier momento.

En resumen, el entorno de trabajo en el que se ha desarrollado el proyecto es el siguiente. Por una parte, se dispone de un servidor en el que se tiene dos contenedores con instancias de *Neo4j* (*Community* y *Enterprise*). Y, por otro parte, en un ordenador de sobremesa se dispone de la aplicación de escritorio, del cual se puede conectar a las dos

instancias del servidor o a las instancias que se tengan en local. Estas instancias locales se utilizan para lanzar pruebas.

En cuanto a la configuración de *Neo4j*, existen varios aspectos que se deben tener en cuenta antes de lanzar cualquier consulta a la base de datos.

Como se puede ver en la **Figura 27**, *Neo4j* distribuye la memoria RAM en cuatro subáreas diferentes: *OS*, *Page cache*, *Heap space* y *Transaction state* [19].

La primera subárea es para el sistema operativo. Necesita reservar memoria para los procesos del sistema operativo. Una de las cosas a tener en cuenta es que éste no es configurable, luego no se puede establecer la cantidad de memoria a reservar. La memoria que se le asigna es la que queda disponible después de configurar las demás partes. Así que, si se abusa en las demás subáreas, ésta se quedará sin memoria y *Neo4j* no funcionará correctamente.

La subárea de *page cache* sirve para almacenar en caché los datos de *Neo4j* y los índices nativos. Con esto se consigue evitar el acceso a disco y obtener un mejor rendimiento.

La tercera subárea *heap space* es una con las que más problemas se han tenido durante el desarrollo del trabajo Fin de Grado. Ésta sirve para la ejecución de las consultas, la gestión del grafo, etc. El tamaño de reserva de la misma depende del tipo de consultas que se vaya a ejecutar. Si las consultas son muy complejas o tienen una larga duración, esta subárea necesitará más recursos (más cantidad de memoria). Durante el proyecto, se ha trabajado con consultas con una complejidad y duración elevada. Tanto para la importación de los datos, como para la explotación de la base de datos, en los dos casos, se ha tenido que ir incrementando la memoria reservada sin sobrepasarse. No obstante, en muchas ocasiones no se podía incrementar más y por lo tanto se ha tenido que duplicar la memoria RAM del servidor.

Finalmente se encuentra la subárea *Transaction state* que se utiliza para almacenar los datos y resultados intermedios en las transacciones que actualizan los registros de la base de datos.

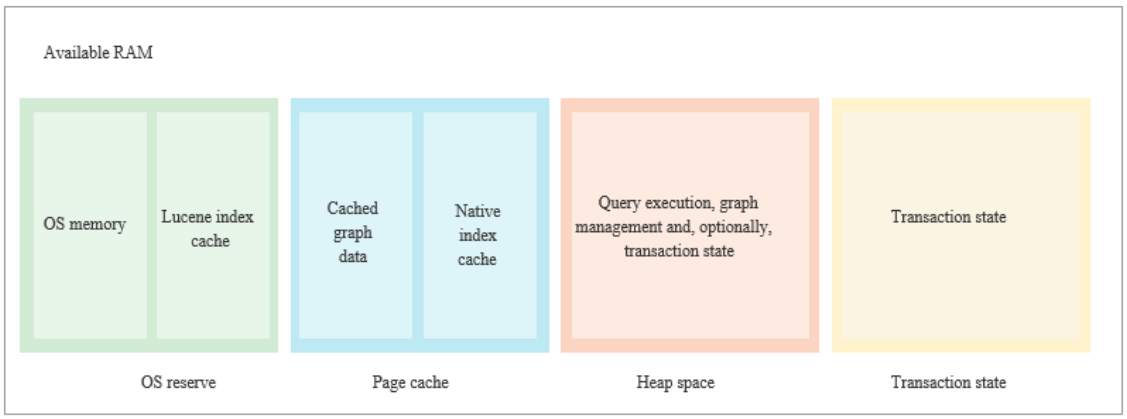


Figura 27 *- Representación de la distribución de la memoria RAM de Neo4j.*

5.2 – Creación de *constraints*

Es importante realizar la creación de los *constraints* antes de importar todos los datos por diversas razones:

- Optimiza el tiempo de importación ya que se crean índices en aquellas propiedades que tienen una restricción/constraints.
- Asegura que todos los datos que se insertan cumplen con esas restricciones. Si se añadiese después de la importación, podría ocurrir que alguno de los datos existentes entrase en conflicto con las restricciones a crear, y por lo tanto no se crearían, concluyendo en un error.
- Generar índices o restricciones después de la importación, supone recorrer todos los datos de nuevo para crear las restricciones e índices en cada uno de ellos. Sin embargo, si se efectúa esta tarea antes, primero se especifica el esquema de restricciones e índices y luego, a la hora de la importación, se generan los datos y las restricciones/índices simultáneamente.

En nuestro caso se va a crear una restricción de unicidad en las propiedades de los nodos que deben ser únicas:

- En la propiedad *NIF* del nodo con etiqueta *:Persona_fisica*

```
CREATE CONSTRAINT ON (n:Persona_fisica) ASSERT (n.NIF) IS UNIQUE
```

- En la propiedad *NIF* del nodo con etiqueta *:Persona_juridica*

```
CREATE CONSTRAINT ON (n:Persona_juridica) ASSERT (n.NIF) IS UNIQUE
```

- En la propiedad *dirección* del nodo con etiqueta *:Direccion*

```
CREATE CONSTRAINT ON (n:Direccion) ASSERT (n.direccion) IS UNIQUE
```

- En la propiedad *numero* del nodo con etiqueta *:Telefono_fijo*

```
CREATE CONSTRAINT ON (n:Telefono_fijo) ASSERT (n.numero) IS UNIQUE
```

- En la propiedad *numero* del nodo con etiqueta *:Telefono_movil*

```
CREATE CONSTRAINT ON (n:Telefono_movil) ASSERT (n.numero) IS UNIQUE
```

- En la propiedad *id* del nodo con etiqueta *:Modelo*

```
CREATE CONSTRAINT ON (n:Modelo) ASSERT (n.id) IS UNIQUE
```

- En la propiedad *id* del nodo con etiqueta *:Declaración*

```
CREATE CONSTRAINT ON (n:Declaracion) ASSERT (n.id) IS UNIQUE
```

- En la propiedad *id* del nodo con etiqueta *:Deuda*

```
CREATE CONSTRAINT ON (n:Deuda) ASSERT (n.id) IS UNIQUE
```

Como en capítulos anteriores se ha comentado, por cada restricción creada, *Neo4j* simultáneamente crea un índice para esa propiedad. Luego también se dispondrá de índices para las propiedades mencionadas. Esto hará que las búsquedas sobre esas propiedades sean más rápidas.

5.3 – Importación de datos

Una vez se tienen creadas las *constraints* que se desean, se procede a realizar el proceso de importación de datos.

La importación de datos se ha llevado a cabo mediante la función **LOAD CSV** que se ha estudiado en el *Capítulo 3.3.2 – Profundizando en Neo4j y Cypher.* Esta cláusula te permite importar los datos desde un fichero *csv.* En concreto se ha utilizado **LOAD CSV WITH HEADER**, que se usa cuando los ficheros *csv* disponen de una cabecera con los nombres de los campos.

Los campos del fichero *csv* pueden encontrarse separados por cualquier tipo de delimitador, ya que, en la cláusula **LOAD CSV** se permite especificar el delimitador que se quiere usar. Por defecto se utiliza la coma.

Importación de las personas físicas y jurídicas

La estructura del fichero csv se puede apreciar en la **Figura 28**. Se trata de un fichero csv con dos campos separados por comas y una cabecera para determinar el nombre de cada campo.

El comando dispone de tres elementos claves: la dirección del fichero *csv*, el delimitador que se quiere usar y el patrón que se quiere crear. En este último, se puede ver que mediante la variable *csvLine* se accede a los campos del fichero que se quiere. El patrón se creará por cada línea del fichero *csv*. Luego si el fichero consta de 1000 lineas, se crearán 1000 nodos de personas físicas o jurídicas.

```
1  ,nif
2  0,72291243N
3  1,94977480S
4  2,71239198S
5  3,80373439T
6  4,22609942M
7  5,68544894F
```

```
LOAD CSV WITH HEADERS FROM 'file:///personas_fisicas.csv' AS csvLine FIELDTERMINATOR ','
CREATE (:Persona_fisica {NIF:csvLine.nif});

LOAD CSV WITH HEADERS FROM 'file:///personas_juridicas.csv' AS csvLine FIELDTERMINATOR ','
CREATE (:Persona_juridica {NIF:csvLine.nif});
```

***Figura 28** - Estructura del fichero csv y código para la importación de personas.*

Importación de las relaciones de parejas, de hijos y de personas asistidas

En la **Figura 29** se puede apreciar la estructura de los ficheros junto con los comandos utilizados para crear las relaciones entre las personas físicas. En estos se almacenan los *NIF* de las diferentes personas físicas relacionadas.

Para crear las relaciones, por cada línea del fichero se extraen los dos NIFs, se obtienen los dos nodos que se quieren relacionar (cláusula **MATCH**) y se almacenan en dos variables. Aquí es donde se puede ver la ventaja de haber asignado una *constraint unique* a las propiedades *NIF*, ya que, al tener un índice, hace que la búsqueda de estos dos nodos sea más rápida.

Una vez se obtienen los dos nodos, se crea la relación con la etiqueta correspondiente.

relaciones_familiares_hijos.csv

```
,padreMadre,hijoHija
0,32559281G,50041540N
1,49653941K,25275574N
2,10273168W,95628124Z
3,15621424H,62911860P
4,32563052Q,95686836W
```

relaciones_familiares_parejas.csv

```
,pareja1,pareja2
0,72291243N,32226401G
1,94977480S,28637552T
2,80373439T,22716016F
3,22609942M,75264972K
4,90170542J,67420126B
```

relaciones_familiares_asistidos.cs

```
,asistido,asistente
0,94977480S,41396024X
1,22609942M,30958223Y
2,60698551P,66999505C
3,65222587R,27450689G
4,76424784A,25889101P
```

```
LOAD CSV WITH HEADERS FROM 'file:///relaciones_familiares_hijos.csv' AS csvLine FIELDTERMINATOR ','
MATCH (prog:Persona_fisica {NIF:csvLine.padreMadre}), (des:Persona_fisica {NIF:csvLine.hijoHija})
CREATE (prog)-[:ES_ASCENDENTE]->(des);

LOAD CSV WITH HEADERS FROM 'file:///relaciones_familiares_parejas.csv' AS csvLine FIELDTERMINATOR ','
MATCH (p1:Persona_fisica {NIF:csvLine.pareja1}), (p2:Persona_fisica {NIF:csvLine.pareja2})
CREATE (p1)-[:ES_CONYUGE]->(p2);

LOAD CSV WITH HEADERS FROM 'file:///relaciones_familiares_asistidos.csv' AS csvLine FIELDTERMINATOR ','
MATCH (p1:Persona_fisica {NIF:csvLine.asistido}), (p2:Persona_fisica {NIF:csvLine.asistente})
CREATE (p1)-[:ES_ASISTIDO_POR]->(p2);
```

***Figura 29** - Estructura de los ficheros csv y código para la creación de relaciones familiares.*

Importación de las relaciones de socios, y administradores

La primera consulta que se muestra en la **Figura 30**, se utiliza para crear las relaciones de participación entre las personas jurídicas. Sin embrago, las dos últimas consultas crean las relaciones de socios y administradores entre las personas físicas y las personas jurídicas.

Siguen la misma estructura que las consultas anteriores. Se obtienen los dos nodos a relacionar y se crean entre ellas dos la relación pertinente. Nos obstante, en las relaciones de socios entre personas físicas y jurídicas (última consulta), a la relación se le añade una propiedad del porcentaje de participación. Este porcentaje de participación se encuentra en el fichero de donde se obtiene los nodos a relacionar.

relaciones_socios_PJ.csv

```
,empresa,subEmpresa
0,2152919C,9789601G
1,11705042,9298315N
2,5115126Q,9984145K
3,9694988K,6602587V
4,9693172Y,4266139S
```

relaciones_administradores.csv

```
,PF,PJ
0,32226401G,6888178C
1,28637552T,4292334M
2,71239198S,6612589C
3,90170542J,8406932B
4,76843694P,8090922F
```

relaciones_socios_PF.csv

```
,PF,PJ,pesos
0,94977480S,7011522N,0.833590
1,89574625R,1215611D,0.140226
2,20378301R,1652728J,0.277163
3,71809676K,7980415J,0.111018
4,628627337,2732940X,0.242234
```

```
LOAD CSV WITH HEADERS FROM 'file:///relaciones_socios_PJ.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pj1:Persona_juridica {NIF:csvLine.empresa}), (pj2:Persona_juridica {NIF:csvLine.subEmpresa})
CREATE (pj1)-[:DOMINA]->(pj2);

LOAD CSV WITH HEADERS FROM 'file:///relaciones_administradores.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pj:Persona_juridica {NIF:csvLine.PJ}), (pf:Persona_fisica {NIF:csvLine.PF})
CREATE (pj)-[:TIENE_ADMINISTRADOR]->(pf);

LOAD CSV WITH HEADERS FROM 'file:///relaciones_socios_PF.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pf:Persona_fisica {NIF:csvLine.PF}), (pj:Persona_juridica {NIF:csvLine.PJ})
CREATE (pf)<-[:TIENE_SOCIO {participacion:csvLine.pesos}]-(pj);
```

Figura 30 *- Estructuras de los ficheros csv y código para la creación de las relaciones empresariales.*

Importación de las direcciones, teléfonos fijos y móviles

La importación de las direcciones, teléfonos y móviles se ha hecho de la misma forma que para las personas.

telefonos.csv

```
,tlf
0,941716424
1,984394463
2,996637801
3,971390623
4,984762458
```

moviles.csv

```
,movil
0,611260066
1,673921546
2,601547857
3,668465771
4,600210593
```

direcciones.csv

```
,direccion
0,5670670JB0003E0002YV
1,2863122RA0002J0003HX
2,2671589PS0001F0004JQ
3,3338613YG0001V0000BQ
4,2611140GW0004Q0001NB
```

```
LOAD CSV WITH HEADERS FROM 'file:///telefonos.csv' AS csvLine FIELDTERMINATOR ','
CREATE (:Telefono_fijo {numero:csvLine.tlf});

LOAD CSV WITH HEADERS FROM 'file:///moviles.csv' AS csvLine FIELDTERMINATOR ','
CREATE (:Telefono_movil {numero:csvLine.movil});

LOAD CSV WITH HEADERS FROM 'file:///direcciones.csv' AS csvLine FIELDTERMINATOR ','
CREATE (:Direccion {direccion:csvLine.direccion});
```

Figura 31 *- Estructuras de ficheros y código para la creación de las direcciones, fijos y móviles.*

Creación de relaciones entre personas físicas y direcciones, móviles y fijos

Para la creación de vínculos entre personas físicas y direcciones, móviles y fijos se sigue un procedimiento diferente. Esto se debe a que puede haber campos (*tlf*, *movil*, *direccionDerecho*) que tienen un valor que no corresponde con ninguna de los nodos existentes.

Cuando se quiere establecer que una persona física no dispone de un fijo o de un movil o de una dirección con derecho, se le asigna el valor -1 en el fichero *csv*. Con esto nos aseguramos de que a la hora de hacer **OPTIONAL MATCH** no encuentre ninguna coincidencia, ya que, ningún elemento tiene como identificador un número negativo. Como se tiene la cláusula **OPTINAL MATCH**, si no encuentra coincidencia la variable correspondiente se establecerá a NULL.

Por otra parte, como en *Cypher* no se dispone de estructuras *if*, mediante **CASE WHEN** y **FOREACH** se recrea la estructura *if*. Primero con **CASE WHEN** se crea una lista con un elemento si la variable no es NULL , y una lista vacía en caso contrario.

Así, si la lista es vacía (no hay coincidencias), no se entra en el **FOREACH**, por lo que no se realiza la acción que se encuentra dentro. Sin embargo, si la lista no está vacía (ha encontrado coincidencias), sí entra en el **FOREACH**. De esta forma se están separando las acciones dependiendo de la condición de si la variable es NULL o no. Como bien se ha comentado antes, la variable será NULL cuando no haya ninguna coincidencia para los fijos, móviles o direcciones de derecho.

teléfonos_direcciones.csv

```
,PF,tlf,direccion,movil,direccionDerecho
0,72291243N,953370172,1502739FR0002X0001AB,633312808,-1
1,94977480S,975248909,0711118NQ0004W0001BM,-1,7920062AB0002T0003BB
2,71239198S,902254631,8567887TB0001B0001NA,607766562,1035179VX0000T0001VS
3,80373439T,902037989,4413777EA0003T0000GY,644746750,7696692CF0003J0001XX
4,22609942M,-1,1857962ZJ0000L0003BD,612304040,8219711FY0003D0000AN
```

```
LOAD CSV WITH HEADERS FROM 'file:///telefonos_direcciones.csv' AS csvLine FIELDTERMINATOR ','
OPTIONAL MATCH (pf:Persona_fisica {NIF:csvLine.PF})
OPTIONAL MATCH(direccion:Direccion {direccion:csvLine.direccion})
OPTIONAL MATCH(direccionDerecho:Direccion {direccion:csvLine.direccionDerecho})
OPTIONAL MATCH(tlf:Telefono_fijo {numero:csvLine.tlf})
OPTIONAL MATCH(movil:Telefono_movil {numero:csvLine.movil})
WITH *, CASE WHEN movil IS NOT NULL THEN [1] ELSE [] END  as movil_no_nulo,
        CASE WHEN tlf IS NOT NULL THEN [1] ELSE [] END  as tlf_no_nulo,
        CASE WHEN direccionDerecho IS NOT NULL THEN [1] ELSE [] END  as direccionDerecho_no_nulo
CREATE (pf)-[:HA_INDICADO]->(direccion)
FOREACH(x in direccionDerecho_no_nulo| CREATE(direccionDerecho)<-[:TIENE_DERECHO_SOBRE]-(pf))
FOREACH(x in movil_no_nulo| CREATE(pf)-[:TIENE_MOVIL]->(movil))
FOREACH(x in tlf_no_nulo | CREATE(pf)-[:TIENE_FIJO]->(tlf));
```

Figura 32 *- Estructura del fichero csv y código para las relaciones de direcciones, fijos y móviles.*

Importación de modelos y las relaciones

Las dos primeras consultas que se pueden apreciar en la **Figura 33** sirven para crear un modelo y asignarle la persona física a la que se le imputa y la persona jurídica que lo imputa.

En la primera consulta, se obtiene el nodo (segunda línea de la consulta) que será a la cual se le impute el modelo, luego se crea el modelo (tercera línea) y finalmente se crea la relación entre estos dos (cuarta línea).

En la segunda consulta se quiere crear la relación entre el modelo anterior y la empresa que lo imputa. Para ello se obtiene el modelo que se ha creado arriba, se obtiene la empresa y se crea la relación.

La última consulta se ha hecho de diferente forma, para proporcionar diferentes puntos de vista. En primer lugar, se obtiene la persona jurídica a la que se le va a imputar el modelo y la persona jurídica que lo va a imputar. Posteriormente, mediante **MERGE** se crea el modelo correspondiente. Como se ha visto en anteriores capítulos, si la cláusula **MERGE** no encuentra coincidencias del patrón, lo crea y le asigna las propiedades que se establecen en **ON CREATE SET**. Por lo que en nuestro caso siempre lo creará ya que no existirá ningún nodo modelo con ese identificador.

relaciones_modelos_PF.cs

```
,PF,modelo,PJ,id,importe
0,72291243N,180,9674601Y,0,1000
1,72291243N,159,9707307E,1,1000
2,32226401G,180,1653890H,2,1000
3,32226401G,159,6354900W,3,1000
4,94977480S,180,1514594D,4,1000
```

relaciones_modelos_PJ.csv

```
,PJ1,modelo,PJ2,id,importe
0,2152919C,159,2108432M,2564019,5000
1,5790370D,159,7563924Z,2564020,1000
2,7858366P,159,7415437K,2564021,1000
3,3152451C,159,5008364R,2564022,1000
4,7906795J,159,1571584R,2564023,5000
```

```
LOAD CSV WITH HEADERS FROM 'file:///relaciones_modelos_PF.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pf:Persona_fisica {NIF:csvLine.PF})
CREATE (m:Modelo {id:csvLine.id, modelo:csvLine.modelo, importe:csvLine.importe})
CREATE (pf)-[:ES_IMPUTADO]->(m) ;

LOAD CSV WITH HEADERS FROM 'file:///relaciones_modelos_PF.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pj:Persona_juridica {NIF:csvLine.PJ}),(m:Modelo {id:csvLine.id})
CREATE (m)<-[:IMPUTA]-(pj);

LOAD CSV WITH HEADERS FROM 'file:///relaciones_modelos_PJ.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pj1:Persona_juridica {NIF:csvLine.PJ1}), (pj2:Persona_juridica {NIF:csvLine.PJ2})
MERGE (m:Modelo {id:csvLine.id})
ON CREATE SET m.modelo = csvLine.modelo, m.importe = csvLine.importe
CREATE (pj1)-[:ES_IMPUTADO]->(m)<-[:IMPUTA]-(pj2);
```

Figura 33 *- Estructuras de ficheros csv y código para crear modelos y relaciones de modelos.*

Importación de declaraciones y sus relaciones

Para la importación de estas entidades, se han utilizado técnicas explicadas en las anteriores importaciones. Por una parte, dependiendo del campo *dRel* del fichero *csv*, se quiere crear una relación con una etiqueta o con otra. Esto sirve para determinar qué tipo de relación hay entre una persona y una declaración (sujeto pasivo, causante, transmitente). Para ello se han utilizado las estructuras **CASE WHEN - FOREACH** que simulan la estructura *if*. Por otra parte, se ha utilizado la cláusula **MERGE** para crear las declaraciones.

La primera consulta es para los modelos de las personas físicas y la segunda consulta para los de las personas jurídicas.

declaraciones.csv

```
,PF,dID,dTipo,dRel,importe
0,72291243N,1,mueble,1,20000
1,72291243N,2,inmueble,1,50000
2,94977480S,3,mueble,1,20000
3,94977480S,4,vehiculo,1,50000
4,94977480S,5,inmueble,1,20000
```

declaraciones_pj.csv

```
,PJ,dID,dTipo,dRel,importe
0,2152919C,2898138,cuenta,1,20000
1,5790370D,2898139,cuenta,2,1000
2,5790370D,2898140,vehiculo,2,5000
3,7858366P,2898141,cuenta,1,5000
4,3152451C,2898142,cuenta,1,50000
```

```
LOAD CSV WITH HEADERS FROM 'file:///declaraciones.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pf:Persona_fisica {NIF:csvLine.PF})
MERGE (d:Declaracion {id:csvLine.dID})
ON CREATE SET d.tipo = csvLine.dTipo, d.importe = csvLine.importe
WITH *,CASE WHEN csvLine.dRel = '0' THEN [1] ELSE [] END  as causante,
       CASE WHEN csvLine.dRel = '1' THEN [1] ELSE [] END  as sp,
       CASE WHEN csvLine.dRel = '2' THEN [1] ELSE [] END  as trasmitente
FOREACH(x in causante | CREATE (pf)-[:ES_CAUSANTE]->(d))
FOREACH(x in trasmitente | CREATE (pf)-[:ES_TRASMITENTE]->(d))
FOREACH(x in sp | CREATE(pf)-[:ES_SP]->(d));

LOAD CSV WITH HEADERS FROM 'file:///declaraciones_pj.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pj:Persona_juridica {NIF:csvLine.PJ})
MERGE (d:Declaracion {id:csvLine.dID})
ON CREATE SET d.tipo = csvLine.dTipo, d.importe = csvLine.importe
WITH *,CASE WHEN csvLine.dRel = '0' THEN [1] ELSE [] END  as causante,
       CASE WHEN csvLine.dRel = '1' THEN [1] ELSE [] END  as sp,
       CASE WHEN csvLine.dRel = '2' THEN [1] ELSE [] END  as trasmitente
FOREACH(x in causante | CREATE (pj)-[:ES_CAUSANTE]->(d))
FOREACH(x in trasmitente | CREATE (pj)-[:ES_TRASMITENTE]->(d))
FOREACH(x in sp | CREATE(pj)-[:ES_SP]->(d));
```

Figura 34 *- Estructuras de ficheros csv y código para importación de las declaraciones y sus relaciones.*

Importación de las deudas y sus relaciones

Finalmente se encuentra la importación de las deudas que siguen la misma estructura que las demás consultas. Primero se obtiene la persona que contrae la deuda con **MATCH**, luego se crea la deuda con el funcionamiento que supone **MERGE**, y finalmente, se crea la relación con **CREATE**.

deudas.csv

```
,PF,dID,importe
0,80373439T,1,200000
1,22609942M,2,200000
2,23808030E,3,160000
3,56425419D,4,160000
4,42360784G,5,200000
```

deudas_pj.csv

```
,PJ,dID,importe
0,5790370D,340955,160000
1,7858366P,340956,160000
2,2214521D,340957,500000
3,2442162P,340958,50000
4,4534511H,340959,160000
```

```
LOAD CSV WITH HEADERS FROM 'file:///deudas.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pf:Persona_fisica {NIF:csvLine.PF})
MERGE (d:Deuda {id:csvLine.dID})
ON CREATE SET d.importe = csvLine.importe
CREATE (pf)-[:TIENE_CONTRAIDA]->(d);

LOAD CSV WITH HEADERS FROM 'file:///deudas_pj.csv' AS csvLine FIELDTERMINATOR ','
MATCH (pj:Persona_juridica {NIF:csvLine.PJ})
MERGE (d:Deuda {id:csvLine.dID})
ON CREATE SET d.importe = csvLine.importe
CREATE (pj)-[:TIENE_CONTRAIDA]->(d);
```

Figura 35 *- Estructuras de fichero csv y código para importar deudas y sus relaciones.*

5.3.1 – Proceso de importación

El proceso de importación se ha llevado a cabo mediante la API de *Neo4j* para *Python*, creando así un notebook de *Jupyter* que interactúa con las diferentes bases de datos de *Neo4j*.

La facilidad de automatizar la recogida de los tiempos y de los resultados es uno de los principales motivos que han hecho decantarse por la API para *Python* y no por el navegador de *Neo4j*. En el navegador se muestran los tiempos, pero no se almacenan en ningún fichero, por lo que se debe ir recolectando manualmente de uno en uno los tiempos obtenidos en un documento aparte. Sin embargo, mediante la API de *Python* se pueden ir almacenando en un fichero conforme se van obteniendo los resultados.

A continuación, se van a mostrar diferentes partes de código utilizadas para conectarse con el driver de *Neo4j* y lanzar las consultas necesarias.

Lo primero que se debe hacer para poder trabajar con *Neo4j* desde *Python*, es instalarse las librerías del driver de *Neo4j* para el entorno de *Python* que se esté utilizando. Una vez esté instalado se debe importar en el notebook correspondiente.

Una vez importado, se debe especificar la localización de la base de datos junto con el puerto del servicio. También se debe especificar el usuario y contraseña de la instancia de *Neo4j*, véase **Código 26**.

```
uri = 'bolt://localhost:7687'
driver = GraphDatabase.driver( uri , auth = ("nombreUsuario", "contraseña") )
```

Código 26 *- Establecimiento del driver de Neo4j.*

Para la creación de las *constraints* se ha utilizado el siguiente código junto con las consultas explicadas en el *apartado 5.2*. En este código, por cada consulta, se ha abierto una sesión nueva con el driver anteriormente establecido y se ha creado una transacción para lanzar la misma. Una vez terminada se almacenan los tiempos, se completa la transacción y se cierra la sesión.

```
etiquetas = ['Persona_fisica','Persona_juridica','Direccion','Telefono_movil','Telefono_fijo','Declaracion','Deuda','Modelo']
atributosIndexados = ['NIF','NIF','direccion','numero','numero','id','id','id']
tiempos = []
consultas = []


for i in range(0,len(etiquetas)):

    consulta = "CREATE CONSTRAINT ON (nodo:{{etiqueta}}) ASSERT (nodo.{{atributo}}) IS UNIQUE;"

    with driver.session() as session:

        with session.begin_transaction() as tx:

            consulta = consulta.replace("{{etiqueta}}", etiquetas[i])
            consulta = consulta.replace("{{atributo}}", atributosIndexados[i])
            consultas.append(consulta)

            s = time.time()*1000
            result = tx.run(consulta)
            e = time.time()*1000

            print(e-s)
            tiempos.append(e-s)


df = pd.DataFrame({'indexNodo':etiquetas, 'indexAtributo':atributosIndexados, 'tiempo':tiempos, 'consulta':consultas},
                  columns = ['consulta','indexNodo', 'indexAtributo', 'tiempo'])

df.to_excel(carpeta+'tiempo_indices.xlsx',sheet_name='Constraints',encoding='utf-8')
```

Código 25 *- Creación de constraints mediante API de Neo4j para Python.*

Para el proceso de importación de los datos se ha utilizado el **Código 27**. Este código sigue la misma estructura que el anterior. Sin embargo, las consultas mencionadas anteriormente están almacenadas en un fichero.

```
tiempos = []
consultas = []

with open('importacionDatos.cypher', 'r') as f:

    for consulta in f:
        consultas.append(consulta)
        s = timer()

        with driver.session() as session:

            with session.begin_transaction() as tx:
                consultas.append(consulta)
                result = tx.run(consulta)


        e = timer()
        print(e-s)
        print("_____")
        tiempos.append(e-s)


df =  pd.DataFrame({'tiempoTransaccion':tiempos, 'consulta':consultas}, columns = ['consulta', 'tiempoTransaccion'])

df.to_excel(carpeta+'tiempos.xlsx',sheet_name='Carga de datos',encoding='utf-8')
```

Código 27 *- Importación de datos mediante API de Neo4j para Python.*

5.4 – Comparaciones y resultados

Durante toda la realización del proyecto, se han utilizado un total de cuatro muestras que difieren en la cantidad de nodos y relaciones a importar. No obstante, todas siguen el mismo modelo y estructura de datos, véase **Figura 26**.

Las cuatro muestras son las siguiente:

- **Muestra 1**
  - Cantidad de NODOS = 77.194
  - Cantidad de RELACIONES = 158.491
- **Muestra 2**
  - Cantidad de NODOS = 755.194
  - Cantidad de RELACIONES = 1.449.330
- **Muestra 3**
  - Cantidad de NODOS = 2.573.935
  - Cantidad de RELACIONES = 4.199.542
- **Muestra 4**
  - Cantidad de NODOS = 11.003.177
  - Cantidad de RELACIONES = 13.762.667

5.4.1 - Primer estudio. Beneficios de las propiedades indexadas.

En este primer estudio se ha realizado una comparativa entre una importación con indexaciones y una importación sin indexaciones. Para ello, en los dos supuestos, se ha utilizado la misma muestra *(Muestra 1)* y se ha lanzado en una misma instancia de *Neo4j Enterprise* local.

En el primer supuesto no se crean ni índices ni constraint en ningún momento. Sin embargo, en el según supuesto se crea primero las claves primarias que conlleva consigo una creación automática de índices. Esto se realiza antes de importar los datos por los diversos motivos que se han visto anteriormente.

Como se puede apreciar en la **Tabla 2**, los resultados obtenidos son significativos, ya que el tiempo de importación de los datos sin indexación se dispara. Como bien se ha explicado en apartados anteriores, el hecho de establecer unas restricciones y/o índices repercute de una forma considerable, ya que hace que los accesos a los datos sean más rápidos.

Existen varios aspectos a considerar tras analizar la tabla de los resultados:

- En las consultas que solo se insertan nodos (**CREATE**) no se utiliza ninguna cláusula **MATCH** (véase *apartado 5.3*), lo que supone que no debe hacer ninguna búsqueda de ningún nodo en la base de datos. Es por ello, que el disponer de un índice no supone ningún beneficio en esta consulta. Es más, el tiempo de creación de nodos en el supuesto con índices es mayor, ya que el establecimiento de los índices en las propiedades que se hayan especificado conlleva un tiempo extra. No obstante, no es una diferencia muy grande.
- En las consultas en las que se crean relaciones, sin embargo, sí que se hace uso de la cláusula **MATCH**. Primero se extraen los nodos que se quieren relacionar y luego se crea la relación (**CREATE**). En este caso, el uso de los índices sí que es beneficioso, ya que la extracción de nodos se hace mediante las propiedades únicas que en un supuesto están indexadas y en el otro no. Como es de esperar, en este primero la realización del **MATCH** se hace mucho más rápida que en el segundo.

En conclusión, utilizar índices de una forma correcta es beneficioso. En nuestro caso se ha obtenido una mejora del $99{,}74\%$ en cuanto a tiempo de espera se refiere. El aspecto más importante y el que mayor tiempo necesita en estas importaciones son las búsquedas de los nodos que se quieren relacionar, es por ello que con los índices este tiempo se reduce considerablemente. La creación de nodos o relaciones en sí es parecida en los dos supuestos. Todas estas conclusiones se ven reflejadas en la **Tabla 2**

Finalmente, en cuanto a memoria consumida, el supuesto con indexación consume un 72.15% más de memoria puesto que los índices ocupan un espacio extra, véase **Tabla 3**.

			CON ÍNDICES	SIN ÍNDICES
RENDIMIENTO TIEMPO	NODOS	RELACIONES	TIEMPO (ms)	TIEMPO (ms)
Personas físicas	745	-	339	218
Personas jurídicas	450	-	69	28
Relaciones familiares (ascendientes)	-	1048	485	1281
Relaciones familiares (conyuges)	-	179	66	159
Relaciones familiares (asistidos)	-	66	40	65
Relaciones socios (PJ-PJ)	-	13982	764	6975
Relaciones administradores	-	129	29	107
Relaciones socios (PF-PJ)	-	113	35	121
Teléfonos	497	-	25	20
Móviles	695	-	24	17
Direcciones	497	-	36	14
Relaciones PF-direcciones-tlf-mvl	-	1728	225	1350
Relaciones modelos PF (1)	60164	60164	2292	30295
Relaciones modelos PF (2)	-	60164	1036	2210622
Relaciones modelos PJ	6772	13544	468	274491
Declaraciones PF	2699	2699	248	3786
Declaraciones PJ	3888	3888	184	12145
Deudas PF	209	209	29	130
Deudas PJ	578	578	42	380
TOTAL	77194	158491	6436 ms 6,436 seg	2542204 ms 42,3700666 min

Tabla 2 *- Comparativa del tiempo de respuesta.*

	CON ÍNDICES	SIN ÍNDICES
RENDIMIENTO MEMORIA	MEMORIA	MEMORIA
Count store	3,00 KiB	3,34 KiB
Label store	16,02 KiB	16,02 KiB
Index Store	2,73 MiB	80,00 KiB
Schema store	16,01 KiB	8,01 KiB
Array Store	8,01 KiB	8,01 KiB
Logical Log	145,71 MiB	35,90 MiB
Node store	1,12 MiB	935,79 KiB
Property store	3,04 MiB	2,49 MiB
Relationship store	5,29 MiB	4,56 MiB
String store	8,01 KiB	8,01 KiB
TOTAL store	157.95 MiB 165,622579 MB	43,99 MiB 46,126858 MB

Tabla 3 *- Comparativa de la memoria consumida.*

5.4.2 - Segundo estudio. Comparativa entre la importación de datos en Enterprise y Community.

El objetivo de este apartado es comparar el proceso de importación de los datos entre las dos ediciones de *Neo4j*. Para realizar esta comparativa, se han utilizado los dos contenedores *Docker* anteriormente definidos: contenedor *Neo4j Community* y contenedor *Neo4j Enterprise*. Para cada uno de ellos se ha realizado la misma importación de datos.

Además de analizar el comportamiento en las diferentes ediciones, también se quiere ver cómo progresa el rendimiento de éstas según aumenta la cantidad de datos a importar. Es por ello que se han utilizado las muestras definidas en el *apartado 5.4* (*Muestras 1, Muestra 2, Muestra 3 y Muestra 4*).

Los resultados obtenidos se encuentran recogidos en la siguiente **Tabla 4**.

	Community	Enterprise
Muestra 1 *Nodos = 77.194* *Relaciones = 158.491*	24,6308 seg	19,9062 seg
Muestra 2 *Nodos = 755.194* *Relaciones = 1.449.330*	172,9738 seg	128,1440 seg
Muestra 3 *Nodos = 2.573.935* *Relaciones = 4.199.542*	545,0077 seg ≈ 9 min	393,2219 seg ≈ 6 min
Muestra 4 *Nodos = 11.003.177* *Relaciones = 13.762.667*	2906,0149 seg ≈ 48 min	1564,7874 seg ≈ 26 min

Tabla 4 *- Tiempos de respuesta en la importación de los datos.*

Por una parte, se puede apreciar que los tiempos obtenidos con la instancia de *Neo4j Enterprise* son mejores que con la instancia *Community*. En las muestras con una cantidad de datos pequeña, la diferencia es pequeña y poco representativa. Sin embargo, a medida que el tamaño aumenta, la diferencia es notable, véase **Figura 36**.

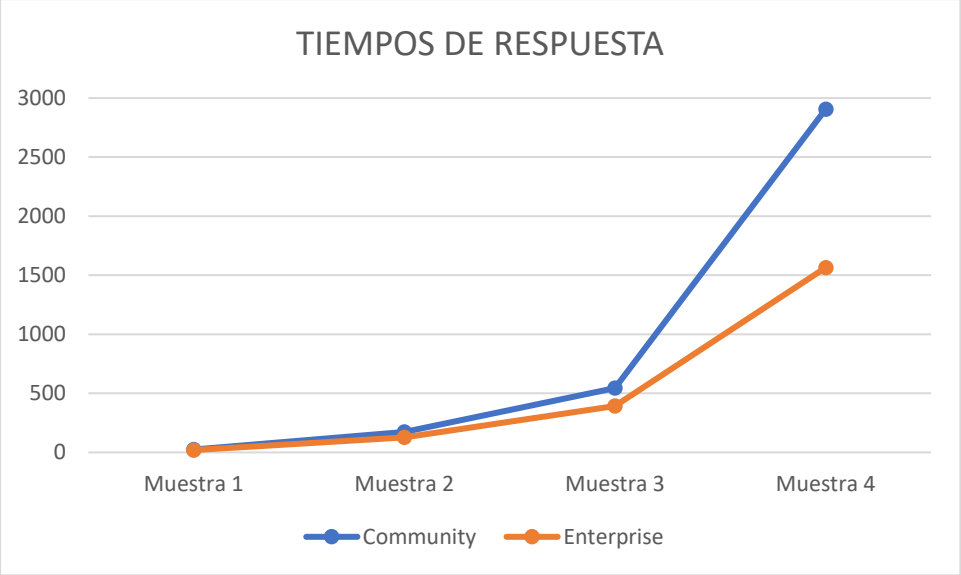


Figura 36 *- Gráfica de los tiempos en segundos.*

Para llevar a cabo la comparación se ha hecho uso del rendimiento relativo de los dos contenedores (*Enterprise* y *Community*) por cada muestra:

$$\frac{Tiempo\ ejecución\ Community}{Tiempo\ ejecución\ Enterprise} = 1 + \frac{n}{100}$$

$$n = \frac{Tiempo\ ejecución\ Community - Tiempo\ ejecución\ Enterprise}{Tiempo\ ejecución\ Enterprise} \times 100$$

- Para la muestra 1: el contenedor con *Neo4j Enterprise* es un $\underline{\mathbf{23,73\%}}$ más rápido que el contenedor con *Neo4j Community*.

$$n = \frac{24{,}6308\ -\ 19{,}9062}{19{,}9062} \times 100 = 23{,}7343\%$$

- Para la muestra 2: el contenedor con *Neo4j Enterprise* es un $\underline{\mathbf{34,98\%}}$ más rápido que el contenedor con *Neo4j Community*.

$$n = \frac{172{,}9738 - 128{,}1440}{128{,}1440} \times 100 = 34{,}9839\%$$

- Para la muestra 3: el contenedor con *Neo4j Enterprise* es un **38,60%** más rápido que el contenedor con *Neo4j Community*.

$$n = \frac{545{,}0077 - 393{,}2219}{393{,}2219} \times 100 = 38{,}6005\%$$

- Para la muestra 4: el contenedor con *Neo4j Enterprise* es un **85,71%** más rápido que el contenedor con *Neo4j Community*.

$$n = \frac{2906{,}0149 - 1564{,}7874}{1564{,}7874} \times 100 = 85{,}7131\%$$

En las siguientes tablas se puede apreciar un desglose del tiempo de respuesta por cada consulta lanzada.

MUESTRA 1	NODOS	RELACIONES	COMMUNITY	ENTERPRISE
Personas físicas	745	-	1,0671	0,9623
Personas jurídicas	450	-	0,2519	0,2223
Relaciones familiares (ascendientes)	-	1048	0,8980	0,7802
Relaciones familiares (conyuges)	-	179	0,1586	0,1456
Relaciones familiares (asistidos)	-	66	0,1045	0,0950
Relaciones socios (PJ-PJ)	-	13982	2,3742	1,9774
Relaciones administradores	-	129	0,1090	0,1046
Relaciones socios (PF-PJ)	-	113	0,1274	0,1109
Teléfonos	497	-	0,1224	0,1030
Móviles	695	-	0,1446	0,1024
Direcciones	497	-	0,1628	0,0916
Relaciones PF-direcciones-tlf-mvl	-	1728	0,6447	0,6635
Relaciones modelos PF (1)	60164	60164	9,1374	7,2205
Relaciones modelos PF (2)	-	60164	5,8670	4,6493
Relaciones modelos PJ	6772	13544	1,8619	1,4533
Declaraciones PF	2699	2699	0,6301	0,4669
Declaraciones PJ	3888	3888	0,7232	0,5583
Deudas PF	209	209	0,0947	0,0788
Deudas PJ	578	578	0,1513	0,1202
TOTAL	77194	158491	**24,6308**	**19,9062**

Tabla 5 *- Desglose de tiempos (segundos) para la muestra 1.*

MUESTRA 2	NODOS	RELACIONES	COMMUNITY	ENTERPRISE
Personas físicas	8500	-	2,6526	2,0803
Personas jurídicas	550	-	0,1381	0,1210
Relaciones familiares (ascendientes)	-	12140	2,6764	2,5716
Relaciones familiares (conyuges)	-	1880	0,3909	0,4988
Relaciones familiares (asistidos)	-	1030	0,2250	0,2382
Relaciones socios (PJ-PJ)	-	384	0,1894	0,1202
Relaciones administradores	-	550	0,1637	0,1526
Relaciones socios (PF-PJ)	-	1308	0,3291	0,3066
Teléfonos	7000	-	0,7971	0,6217
Móviles	7500	-	0,6612	0,6072
Direcciones	7000	-	0,5607	0,5037
Relaciones PF-direcciones-tlf-mvl	-	19682	2,2502	1,9790
Relaciones modelos PF (1)	679827	679827	78,4694	57,1144
Relaciones modelos PF (2)	-	679827	75,5412	55,0821
Relaciones modelos PJ	8185	16370	2,2551	1,9387
Declaraciones PF	29127	29127	4,3531	3,1424
Declaraciones PJ	4803	4803	0,7251	0,6421
Deudas PF	2010	2010	0,4533	0,2757
Deudas PJ	692	392	0,1419	0,1476
TOTAL	755194	1449330	**172,9738**	**128,1440**

Tabla 6 *- Desglose de tiempos (segundos) para la muestra 2.*

MUESTRA 3	NODOS	RELACIONES	COMMUNITY	ENTERPRISE
Personas físicas	110000	-	9,1882	8,2640
Personas jurídicas	7500	-	0,6632	0,6973
Relaciones familiares (ascendientes)	-	174695	16,2395	14,3474
Relaciones familiares (conyuges)	-	37445	4,6389	3,6799
Relaciones familiares (asistidos)	-	20978	2,3441	1,9483
Relaciones socios (PJ-PJ)	-	7901	0,7521	0,6123
Relaciones administradores	-	7500	0,9491	0,8399
Relaciones socios (PF-PJ)	-	16454	2,2640	1,8554
Teléfonos	90000	-	4,9532	3,6594
Móviles	95000	-	4,8013	3,6004
Direcciones	90000	-	4,9716	3,4478
Relaciones PF-direcciones-tlf-mvl	-	271196	22,1967	16,5338
Relaciones modelos PF (1)	1369823	1369823	167,4395	123,0125
Relaciones modelos PF (2)	-	1369823	173,0856	116,3895
Relaciones modelos PJ	112115	224230	24,6225	21,0248
Declaraciones PF	582836	582836	85,6820	59,3299
Declaraciones PJ	65405	65405	11,7593	8,0847
Deudas PF	41855	41855	6,7618	4,7050
Deudas PJ	9401	9401	1,6950	1,1894
TOTAL	2573935	4199542	**545,0077**	**393,2219**

Tabla 7 *- Desglose de tiempos (segundos) para la muestra 3.*

MUESTRA 4	NODOS	RELACIONES	COMMUNITY	ENTERPRISE
Personas físicas	1250000	-	65,3804	54,1658
Personas jurídicas	50000	-	2,9680	2,5447
Relaciones familiares (ascendientes)	-	1505513	149,0646	116,1677
Relaciones familiares (conyuges)	-	471240	61,2517	44,1734
Relaciones familiares (asistidos)	-	134267	20,5157	13,3988
Relaciones socios (PJ-PJ)	-	27457	2,6702	2,1396
Relaciones administradores	-	50000	6,3150	4,7906
Relaciones socios (PF-PJ)	-	187590	24,9422	19,3184
Teléfonos	1245000	-	68,9180	50,4446
Móviles	1250000	-	68,8565	50,1010
Direcciones	1245000	-	70,2941	51,9898
Relaciones PF-direcciones-tlf-mvl	-	2809404	305,7547	204,6057
Relaciones modelos PF (1)	2564019	2564019	1034,1363	271,2467
Relaciones modelos PF (2)	-	2564019	309,0979	234,3380
Relaciones modelos PJ	50000	100000	12,8194	11,2605
Declaraciones PF	2898137	2898137	623,8507	373,3978
Declaraciones PJ	85116	85116	16,8460	13,9424
Deudas PF	340954	340954	57,1180	43,3279
Deudas PJ	24951	24951	5,2154	3,4339
TOTAL	11003177	13762667	**2906,0149**	**1564,7874**

Tabla 8 *- Desglose de tiempos (segundos) para la muestra 4.*

Capítulo 6 – Explotación de bases de datos

Tras la importación de todos los datos, en este capítulo, se realizará una exposición de los supuestos más interesantes solicitados por el departamento de Gestión de Deudas. Para cada supuesto, se describirá el escenario planteado, se explicará la consulta y las características que se deben tener en cuenta y finalmente se mostrará el resultado obtenido.

6.1 - Supuesto 1. Árbol genealógico.

Objetivo

El objetivo de este primer supuesto es la adquisición de una estructura jerárquica que muestre las relaciones familiares entre generaciones de las diferentes personas físicas. Es decir, se parte de una persona deudora o investigada y se quiere obtener toda persona física relacionada a ella, ya sea de una forma directa o no tan directa.

Las relaciones que interesan en este supuesto son las relaciones de familiares de hijos junto con las relaciones de pareja (*conyuges*). Además, se quiere obtener tanto las ascendencias[6] del deudor como las descendencias[7] del mismo.

Confección

Lo primero que se debe especificar es el tipo de salida que se quiere obtener. En este caso se habla de un grafo que muestra a la persona deudora junto con las personas físicas relacionadas.

Se pretende realizar los dos siguientes ejemplos:

- Obtener la estructura jerárquica del deudor con NIF *22609942M.*
- Obtener la estructura jerárquica del deudor con NIF *98361455V.*

El código utilizado para llevar a cabo esta consulta se presenta en el **Código 28**. Para los dos ejemplos mostrados se utiliza la misma consulta, lo único que se debe cambiar es el NIF de la persona que se quiere analizar.

La implementación comienza con una cláusula **MATCH** que es utilizada para extraer el deudor que se quiere estudiar. En este caso el deudor se guardará en la variable *pf.*

```
MATCH (pf:Persona_fisica{NIF:'22609942M'})
```

Seguidamente se utiliza **OPTIONAL MATCH** para obtener todas las ascendencias del deudor, que se almacenarán en la variable *_ascendientes.* Se puede apreciar que el deudor

[6] Con ascendencias se refiere al conjunto de antepasados (antecesores, ancestros, etc.) de una persona física.
[7] El concepto descendencia se define como el conjunto de sucesores de una persona física.

obtenido de la anterior cláusula es utilizado en éste como nodo destino. Además, junto con la especificación *:ES_ASCENDENTE** se indica que se obtengan todos las personas físicas que lleguen al nodo destino (el deudor, variable *pf*) dando la cantidad de saltos que haga falta, a través de las aristas con la etiqueta *ES_ASCENDENTE*.

```
OPTIONAL MATCH (_ascendientes:Persona_fisica)-[:ES_ASCENDENTE*]->(pf)
```

El mismo procedimiento se sigue para las descendencias. Sin embargo, en este caso el deudor no será el nodo destino, sino que será el nodo origen.

```
OPTIONAL MATCH (pf)-[:ES_ASCENDENTE*]->(_descendientes:Persona_fisica)
```

Llegados a este punto, se obtiene tanto las descendencias como las ascendencias. También se obtienen todas las personas descendentes de las descendencias. Sin embargo, no se está obteniendo todas las personas descendentes de las personas ascendientes. Este caso se puede ver representado en la **Figura 37**. Los nodos verdes son las personas que se obtienen, y los nodos rojos son las personas que no se obtienen.

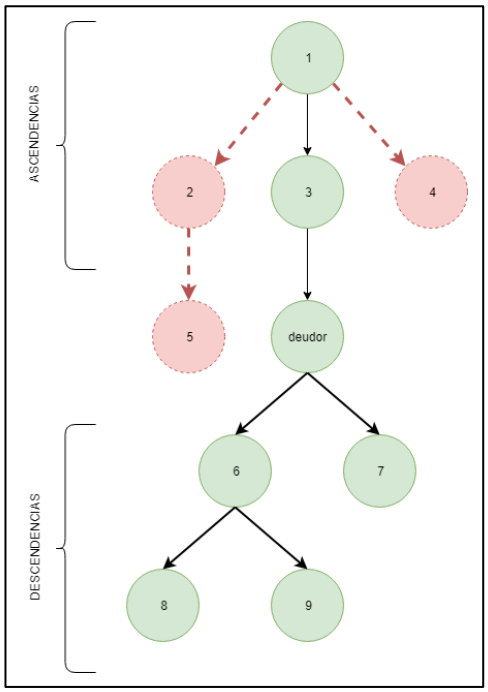


Figura 37 *- Nodos excluidos.*

Por ello, para obtener los nodos excluidos (nodos rojos) se ha añadido la quinta línea. Con ésta se consigue los descendientes de aquellas personas ascendentes del deudor. Estas personas que estaban excluidas se guardan en la variable *p*.

```
OPTIONAL MATCH (_ascendientes)-[:ES_ASCENDENTE*]->(p:Persona_fisica)
```

Con la cláusula **WITH** se recogen en una lista tanto los ascendientes como los descendientes y las personas excluidas, para posteriormente utilizarla en el siguiente bloque.

```
WITH collect(DISTINCT(_descendientes)) + collect(DISTINCT(_ascendientes)) + collect(DISTINCT(p)) + collect(DISTINCT(pf)) AS personas
```

En las siguientes tres lineas se logra obtener las parejas de todas las personas almacenadas en la lista *personas*. Con la cláusula **UNWIND** se consigue pasar de una lista a filas de individuos, para que posteriormente se procese la cláusula **OPTIONAL MATCH** por cada fila. Es importante recalcar que debe ser una búsqueda bidireccional, ya que, en este tipo de relación no tiene sentido ninguno las aristas dirigidas. Tanto las personas y sus parejas se almacenan en otra lista.

```
UNWIND personas AS p
OPTIONAL MATCH (p)-[:ES_CONYUGE]-(conyuges:Persona_fisica)
WITH personas + collect(conyuges) AS personas
```

Hasta este punto se tienen todas las personas que se quieren mostrar, pero se deben conseguir las relaciones entre ellas. Por ello se realizan las siguientes lineas. En las variables *p0* y *p1* se almacenarán tanto los nodos como las relaciones entre ellos.

```
UNWIND personas AS p
OPTIONAL MATCH p0 = (p)-[:ES_ASCENDENTE*]->(:Persona_fisica)
OPTIONAL MATCH p1 = (p)-[:ES_CONYUGE]-(:Persona_fisica)
RETURN p0, p1
```

Es importante recalcar que para este mismo supuesto existen diversas formas de plantear la consulta. No obstante, en nuestro caso se ha elegido ésta porque es con la que mejor resultado se ha obtenido en cuanto a tiempo de respuesta.

```
//MATCH (pf:Persona_fisica{NIF:'98361455V'})
  MATCH (pf:Persona_fisica{NIF:'22609942M'})
  OPTIONAL MATCH (_ascendientes:Persona_fisica)-[:ES_ASCENDENTE*]->(pf)
  OPTIONAL MATCH (pf)-[:ES_ASCENDENTE*]->(_descendientes:Persona_fisica)
  OPTIONAL MATCH (_ascendientes)-[:ES_ASCENDENTE*]->(p:Persona_fisica)
  WITH collect(DISTINCT(_descendientes)) + collect(DISTINCT(_ascendientes)) + collect(DISTINCT(p)) + collect(DISTINCT(pf)) AS personas

  UNWIND personas AS p
  OPTIONAL MATCH (p)-[:ES_CONYUGE]-(conyuges:Persona_fisica)
  WITH personas + collect(conyuges) AS personas

  UNWIND personas AS p
  OPTIONAL MATCH p0 = (p)-[:ES_ASCENDENTE*]->(:Persona_fisica)
  OPTIONAL MATCH p1 = (p)-[:ES_CONYUGE]-(:Persona_fisica)
  RETURN p0, p1
```

Código 28 *- Consulta para obtener un árbol genealógico desde una persona.*

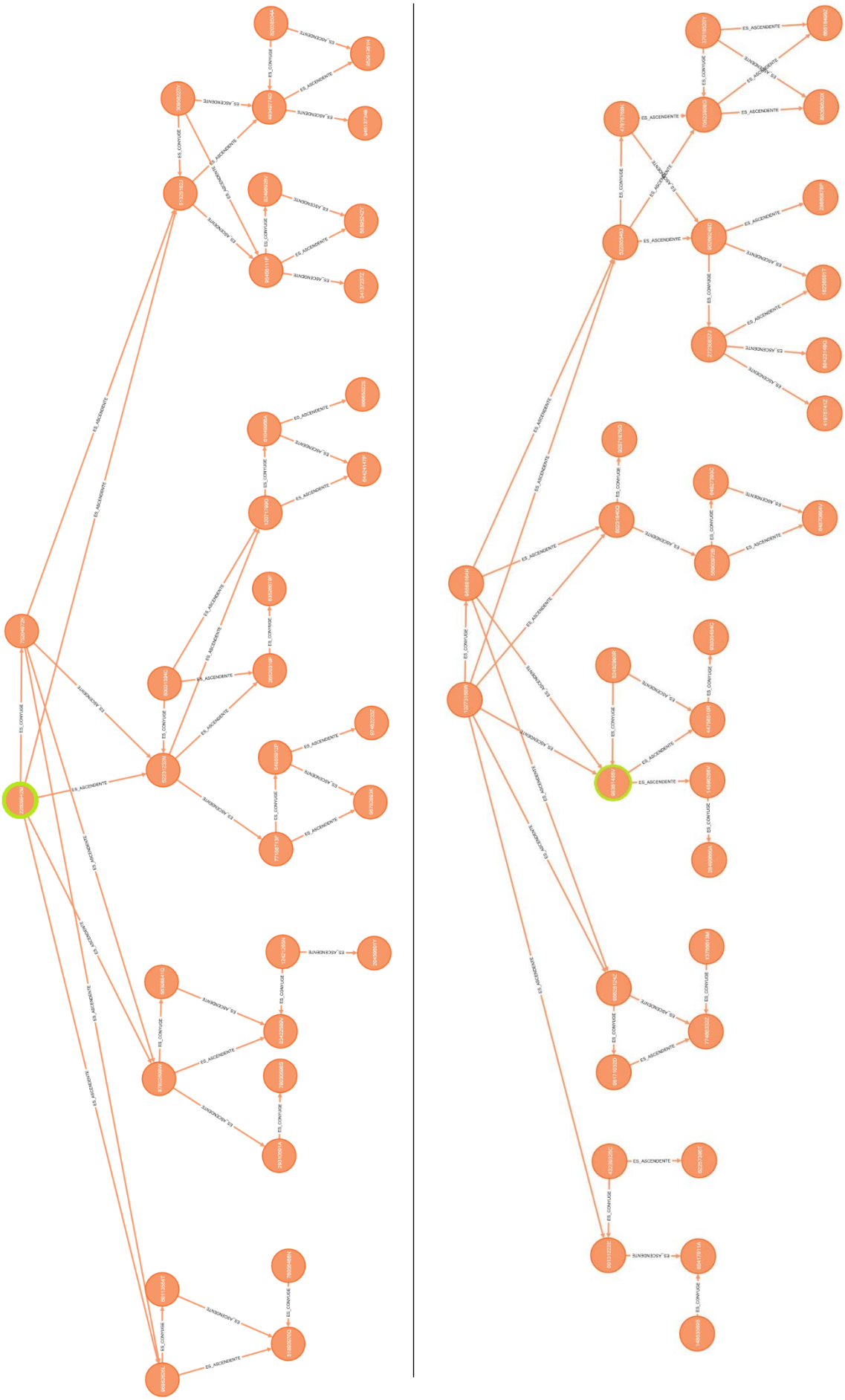
Resultado

En las siguientes figuras se pueden apreciar el grafo resultante tras la consulta explicada. En las figuras se han marcado en verde[8] los nodos deudores, es decir, los puntos de partida de cada consulta.

Mediante estos grafos es muy fácil ver las relaciones existentes entre el deudor y el resto de personas físicas. A los empleados del departamento de Gestión de Deudas este grafo les facilita el trabajo de investigación, ya que, pueden ver de una forma directa y jerárquica todas las relaciones familiares. De esta forma se evita el tener que insertar manualmente los diferentes NIFs para obtener un resultado en forma de tabla. Actualmente, con los datos que consiguen mediante las tablas relacionales, se crean su propio grafo manualmente. Sin embargo, de esta forma lo pueden ver visualmente y con una sola consulta.

En el ejemplo del deudor con NIF *22609942M,* se puede apreciar que éste no dispone de ningún ascendiente. Sin embargo, en el segundo ejemplo (NIF *98361455V*), se puede ver que sí dispone de ascendientes. Luego, se puede avistar que una misma consulta sirve para diferentes casos.

[8] Esta distinción de nodo no lo realiza Neo4j. Es una característica extra que se ha añadido a la figura para una mejor representación.


ES_ASCENDENTE
ES_CONYUGE

6.2 – Supuesto 2. Grupos consolidados.

Objetivo

Según el artículo 30.2 de la Ley Foral 13/2000, de 14 de diciembre, General Tributaria:

> *<<En los supuestos de declaración consolidada, las sociedades del grupo responderán solidariamente del pago de la deuda tributaria, excluidas las sanciones.>>*

Asimismo, el objetivo principal de este supuesto es derivar la deuda de una declaración consolidada a las sociedades que forman parte del grupo consolidado. Los grupos de consolidación fiscal es un conjunto de personas jurídicas (sociedades dominadas y dominantes) residentes en territorio español que cumplan los requisitos establecidos en el Artículo 58 Ley 27/2014, de 27 de noviembre.

Luego, para facilitar el trabajo de la extracción de este grupo de consolidación fiscal, se plantea realizar una consulta que parta de una persona jurídica deudora y muestre todas las sociedades dominantes de ésta y todas las sociedades dominadas.

Es importante destacar que se quiere obtener todas las empresas dominantes del deudor, pero no se quieren obtener todas las empresas dominadas de estas empresas dominantes del deudor. Este supuesto jugaría el papel contrario al planteado en el supuesto anterior. En este caso, los nodos excluidos (nodos rojos) de la **Figura 37** no se quieren extraer.

Confección

Para este segundo supuesto se han realizado dos consultas que difieren en la cantidad de información que se muestra. Es decir, en primer lugar, se plantea una consulta en la que se muestra sólo las personas jurídicas que se encuentran relacionadas de una forma directa o indirecta con el deudor. Sin embargo, en la segunda consulta se muestran estas mismas personas junto con los socios y administradores.

Para la **primera consulta**, que se puede apreciar en el **Código 29**, se comienza extrayendo el nodo deudor que se almacena en la variable *pj*. En este caso, tanto la variable *p0* como la variable *pj* almacenan el mismo valor.

```
MATCH p0 = (pj:Persona_juridica {NIF:'3232480E'})
```

Seguidamente, se obtiene todas las empresas dominadas por este deudor. Para ello, se fija el nodo deudor como nodo origen, se fija un número indefinido de saltos por las aristas *DOMINA* y se determina que los nodos destino deben tener la etiqueta

Persona_juridica. Finalmente, los nodos destinos que cumplan con estas tres condiciones planteadas se almacenan en la variable *dominadas*.

```
OPTIONAL MATCH p1 = (pj)-[:DOMINA*]->(dominadas: Persona_juridica)
```

El mismo procedimiento se sigue para extraer las sociedades dominantes, no obstante, el nodo deudor, en este caso, será el nodo destino.

```
OPTIONAL MATCH p2 = (dominantes: Persona_juridica)-[:DOMINA*]->(pj)
```

En las variables *p1* y *p2* se almacenan los caminos que se entablan entre los diferentes nodos, es decir, se tiene tanto los nodos como relaciones de estas vinculaciones de sociedades.

En este caso concreto, las variables *dominantes* y *dominadas* no tienen utilidad ninguna. No obstante, en la siguiente consulta, el codigo inicial es el mismo y estas variables sí se utilizan, es por eso que aparecen en ésta.

```
MATCH p0 = (pj:Persona_juridica {NIF:'3232480E'})
OPTIONAL MATCH p1 = (pj)-[:DOMINA*]->(dominadas: Persona_juridica)
OPTIONAL MATCH p2 = (dominantes: Persona_juridica)-[:DOMINA*]->(pj)
RETURN p0, p1, p2
```

Código 29 *- Consulta para obtener las sociedades dominantes y dominadas.*

Para la **segunda consulta** mencionada (**Código 30**), el código inicial es el mismo. Se parte de las empresas dominadas y dominantes y se debe obtener los socios y administradores de estos dos conjuntos de sociedades.

El código extra que se añade a la consulta es:

```
WITH collect(pj) + collect(dominadas) + collect(dominantes) as candidatosC, p0, p1, p2
UNWIND candidatosC AS candidatos
OPTIONAL MATCH p3 = (candidatos)-[:TIENE_SOCIO|TIENE_ADMINISTRADOR]->(:Persona_fisica)
```

Con este código, en la primera línea, se almacenan todas las sociedades en una lista y además se pasa las variables *p0*, *p1*, *p2* para posteriormente utilizarlas en la cláusula **RETURN**.

Por cada persona jurídica de la lista creada se realiza la cláusula **OPTIONAL MATCH**, con la cual se obtiene todos los nodos relacionados mediante una única arista TIENE_SOCIO o una arista TIENE_ADMINISTRADOR.

En el **Código 30** se muestra cómo quedaría la consulta de este segundo ejemplo.

```
MATCH p0 = (pj:Persona_juridica {NIF:'3232480E'})
OPTIONAL MATCH p1 = (pj)-[:DOMINA*]->(dominadas: Persona_juridica)
OPTIONAL MATCH p2 = (dominantes: Persona_juridica)-[:DOMINA*]->(pj)
WITH collect(pj) + collect(dominadas) + collect(dominantes) as candidatosC, p0, p1, p2
UNWIND candidatosC AS candidatos
OPTIONAL MATCH p3 = (candidatos)-[:TIENE_SOCIO|TIENE_ADMINISTRADOR]->(:Persona_fisica)
RETURN p0, p1, p2, p3
```

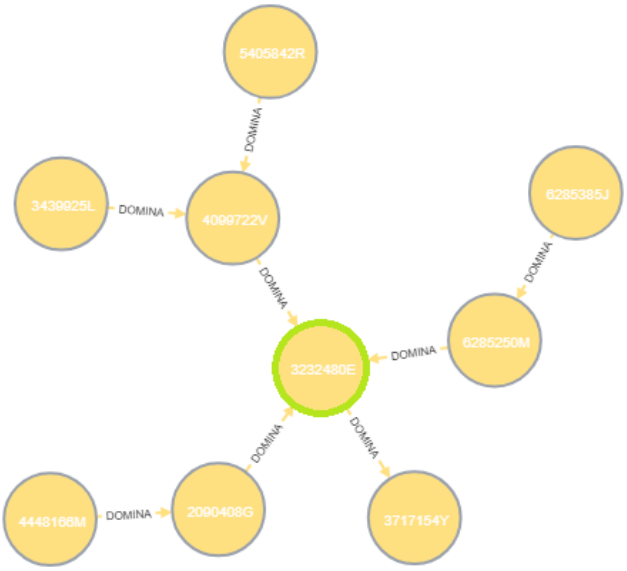
Código 30 *- Consulta para obtener dominantes, dominadas, socios y administradores.*

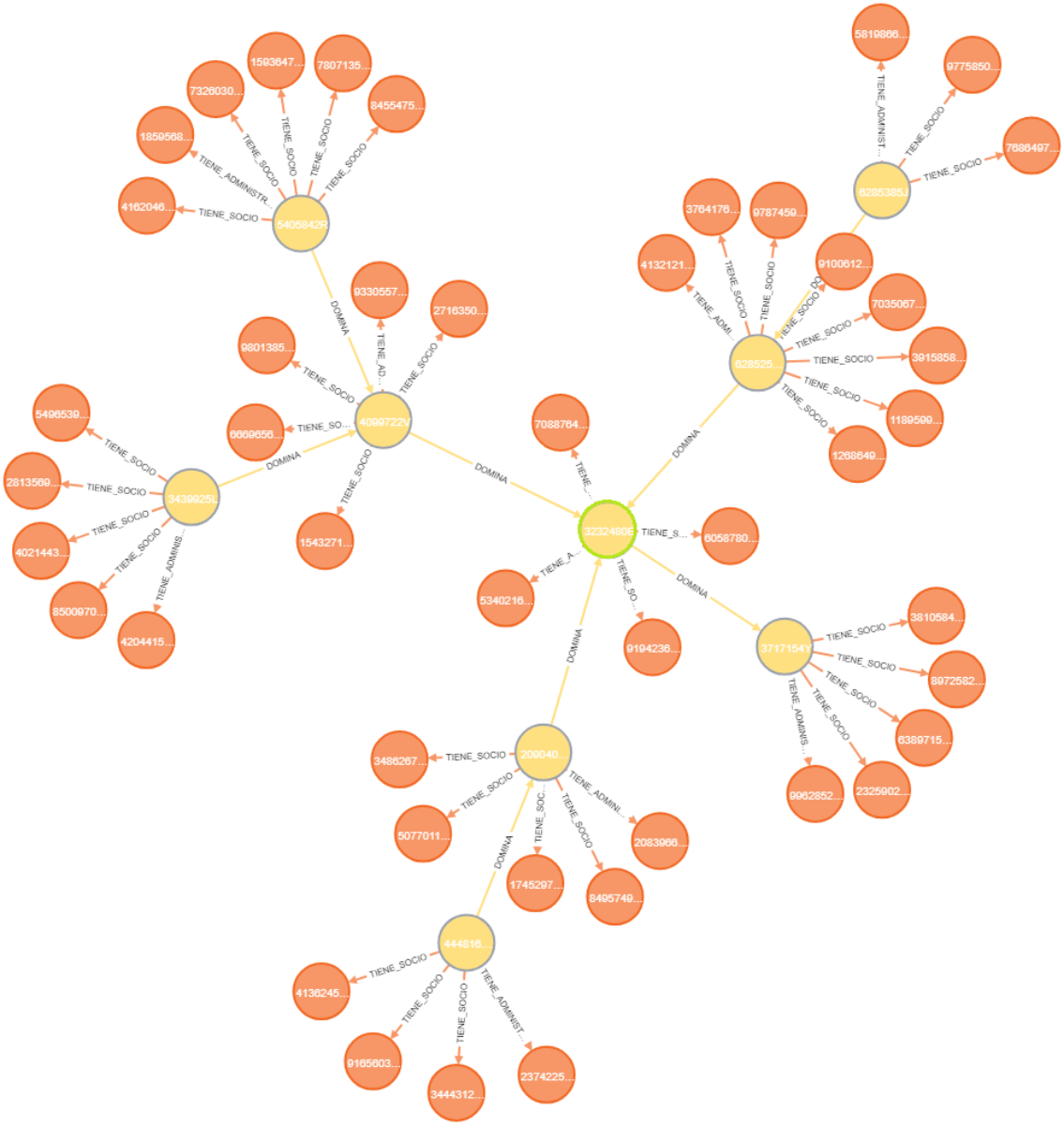
Resultado

El resultado de las dos consultas se muestra en las figuras de abajo, en las que se ha marcado en verde los nodos deudores. En estos dos grafos los nodos amarillos son las personas jurídicas mientras que los nodos naranjas son las personas físicas (socios o administradores).

En la primera figura se puede apreciar de una forma directa que el deudor dispone de siete empresas dominantes mientras que sólo dispone de una empresa dominada. Además, se puede ver, si estas dominaciones son directas o no.

En la segunda figura se aprecian las mismas personas jurídicas pero cada una con un conjunto extra de nodos relacionados que representan los socios y administradores.


5405842R
DOMINA
3439925L
DOMINA
4099722V
DOMINA
6285385J
DOMINA
DOMINA
6285250M
3232480E
DOMINA
DOMINA
DOMINA
4448168M
DOMINA
2090408G
3717154Y


5405842R
4099722V
3439925L
6285385J
628525...
3232480E
3717154Y
209040...
444816...
DOMINA
TIENE_SOCIO
TIENE_ADMINISTR...
TIENE_ADMINIS...
TIENE_ADMINI...
TIENE_SO...
TIENE_S...
TIENE_A...

6.3 - Supuesto 3. Solvencia del colaborador.

En este tercer supuesto se da un paso más y se quiere obtener todos los bienes de las personas físicas y de las personas jurídicas que tengan una relación con el deudor. Se van a presentar dos consultas que, de una manera u otra, engloban los dos supuestos anteriormente explicados.

Se conoce como solvencia a la capacidad económica que tiene una persona física o jurídica. Para esta investigación la solvencia será el conjunto de bienes que dispone el colaborador al que se le pretende derivar la responsabilidad. Para el modelo que se sigue, el importe de los bienes se calculará de los modelos informativos que tenga imputados y de las declaraciones asignadas (solo se tomarán aquellas de las cuales sea sujeto pasivo).

6.3.1 - Visualización de solvencia en grafo

Objetivo

El objetivo principal de esta consulta es obtener un grafo en la que se muestre la deuda que contrae el deudor, las empresas relacionadas a ese deudor, los socios y administradores de éstas y finalmente los bienes de cada persona (física o jurídica).

Asimismo, mediante una segunda versión de esta consulta, se ha añadido la posibilidad de mostrar también las relaciones familiares de los socios y administradores y los bienes de estas últimas personas.

Este supuesto se encuentra orientado para las sociedades deudoras. No obstante, también se puede aplicar para una persona física.

Confección

Tanto para la primera (**Código 31**) como para la segunda versión (**Código 32**) de esta consulta, las siete primeras lineas del codigo son iguales.

Se empieza extrayendo el deudor que se quiere analizar como en todos los supuestos. En este caso no se especifica la etiqueta del nodo ya que puede ser una personas física o jurídica.

```
MATCH (pj {NIF:'5790370D'})
```

Seguidamente se obtiene la deuda del deudor extraído.

```
OPTIONAL MATCH p0 = (pj)-[:TIENE_CONTRAIDA]->(:Deuda)
```

Después, se obtienen las empresas dominantes y dominadas y se almacenan en una lista (se utiliza la misma técnica que en los supuestos anteriores). En caso de que el deudor sea una persona física, los valores de las variables serán NULL, ya que, no se encontrará ninguna coincidencia.

```
OPTIONAL MATCH p1 = (pj)-[:DOMINA*]->(dominadas: Persona_juridica)
OPTIONAL MATCH p2 = (dominantes: Persona_juridica)-[:DOMINA*]->(pj)
WITH collect(pj)+ collect(dominadas) + collect(dominantes) AS candidatosC, p0, p1, p2
```

A continuación, se obtienen los socios y administradores de cada persona almacenada en la lista anterior. En caso de las personas físicas, las variables volverán a ser nulas.

```
UNWIND candidatosC AS candidatos
OPTIONAL MATCH p3 = (candidatos)-[:TIENE_SOCIO|TIENE_ADMINISTRADOR]->(personas:Persona_fisica)
```

Si se quieren añadir también las relaciones familiares de los socios y administradores (segunda versión de la consulta), se deberán incluir las siguientes lineas. En ellas se buscan las relaciones de parejas, de hijos y de las personas asistidas.

```
OPTIONAL MATCH p4 = (personas)-[:ES_CONYUGE|ES_ASCENDENTE|ES_ASISTIDO_POR*]->(descendientes:Persona_fisica)
OPTIONAL MATCH p5 = (personas)<-[:ES_CONYUGE|ES_ASCENDENTE|ES_ASISTIDO_POR*]-(ascendientes:Persona_fisica)
WITH collect(descendientes) + collect(ascendientes) + collect(candidatos) + collect(personas) AS personas, p0, p1, p2, p3, p4, p5
```

En caso de que no se quiera (primera versión de la consulta), sólo se deberá incluir la cláusula **WITH** que se proporciona a continuación. En ella lo único que se hace es recolectar las empresas y las personas físicas (socios y administradores) en una única lista:

```
WITH collect(candidatos) + collect(personas) AS personas, p0, p1, p2, p3
```

Finalmente, tanto para una versión como para la otra, se extraerán los bienes de cada persona. Para ello, como bien se ha comentado antes, habrá que centrarse en las declaraciones en las que la persona es sujeto pasivo (aquellas aristas con la etiqueta *ES_SP*) y en los modelos imputados.

```
UNWIND personas AS p
OPTIONAL MATCH p6 = (p)-[:ES_SP]->(:Declaracion)
OPTIONAL MATCH p7 = (p)-[:ES_IMPUTADO]->(:Modelo)
```

Primera versión de la consulta – Sin familiares de los socios y administradores.

```
MATCH (pj {NIF:'5790370D'})
OPTIONAL MATCH p0 = (pj)-[:TIENE_CONTRAIDA]->(:Deuda)
OPTIONAL MATCH p1 = (pj)-[:DOMINA*]->(dominadas: Persona_juridica)
OPTIONAL MATCH p2 = (dominantes: Persona_juridica)-[:DOMINA*]->(pj)
WITH collect(pj) + collect(dominadas) + collect(dominantes) AS candidatosC, p0, p1, p2

UNWIND candidatosC AS candidatos
OPTIONAL MATCH p3 = (candidatos)-[:TIENE_SOCIO|TIENE_ADMINISTRADOR]->(personas:Persona_fisica)
WITH collect(candidatos) + collect(personas) AS personas, p0, p1, p2, p3

UNWIND personas AS p
OPTIONAL MATCH p6 = (p)-[:ES_SP]->(:Declaracion)
OPTIONAL MATCH p7 = (p)-[:ES_IMPUTADO]->(:Modelo)

RETURN p0, p1, p2, p3, p6, p7
```

Código 31 *- Consulta para mostrar deuda, deudor, empresas relacionadas, socios, administradores y bienes.*

Segunda versión de la consulta – Con familiares de los socios y administradores

```
MATCH (pj {NIF:'5790370D'})
OPTIONAL MATCH p0 = (pj)-[:TIENE_CONTRAIDA]->(:Deuda)
OPTIONAL MATCH p1 = (pj)-[:DOMINA*]->(dominadas: Persona_juridica)
OPTIONAL MATCH p2 = (dominantes: Persona_juridica)-[:DOMINA*]->(pj)
WITH collect(pj) + collect(dominadas) + collect(dominantes) AS candidatosC, p0, p1, p2

UNWIND candidatosC AS candidatos
OPTIONAL MATCH p3 = (candidatos)-[:TIENE_SOCIO|TIENE_ADMINISTRADOR]->(personas:Persona_fisica)
OPTIONAL MATCH p4 = (personas)-[:ES_CONYUGE|ES_ASCENDENTE|ES_ASISTIDO_POR*]->(descendientes:Persona_fisica)
OPTIONAL MATCH p5 = (personas)<-[:ES_CONYUGE|ES_ASCENDENTE|ES_ASISTIDO_POR*]-(ascendientes:Persona_fisica)
WITH collect(descendientes) + collect(ascendientes) + collect(candidatos) + collect(personas) AS personas, p0, p1, p2, p3, p4, p5

UNWIND personas AS p
OPTIONAL MATCH p6 = (p)-[:ES_SP]->(:Declaracion)
OPTIONAL MATCH p7 = (p)-[:ES_IMPUTADO]->(:Modelo)

RETURN p0, p1, p2, p3, p4, p5, p6, p7
```

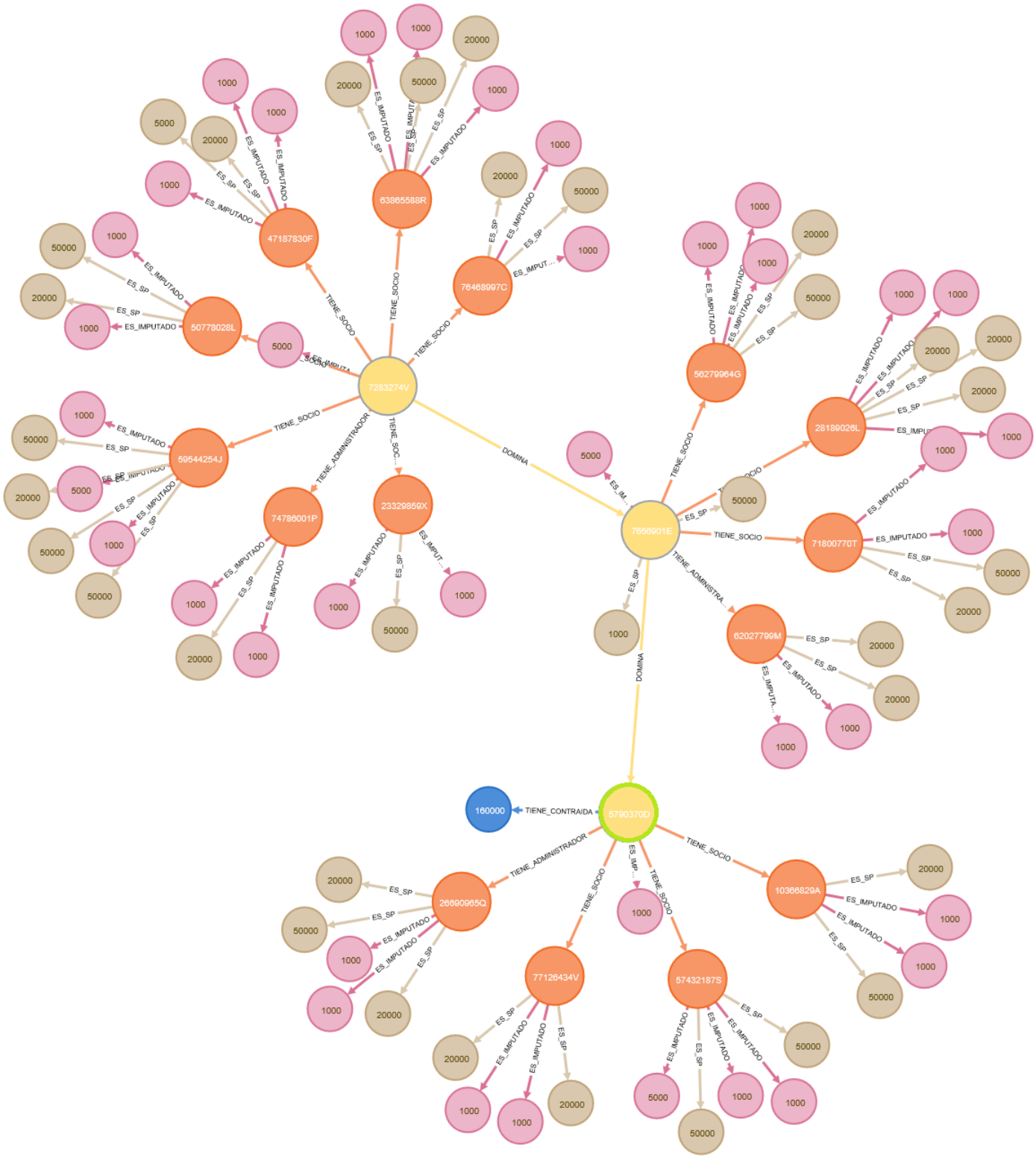
Código 32 *- Consulta añadiendo la parte de la familia.*

Resultado

En los resultados obtenidos el nodo azul representa la deuda, los nodos amarillos representan las personas jurídicas, los nodos naranjas representan las personas físicas (socios y administradores), los nodos marrones representan las declaraciones y los nodos rosas los modelos. Para las deudas, declaraciones y modelos se muestran el importe de cada una de ella. Para las personas se muestran el NIF.

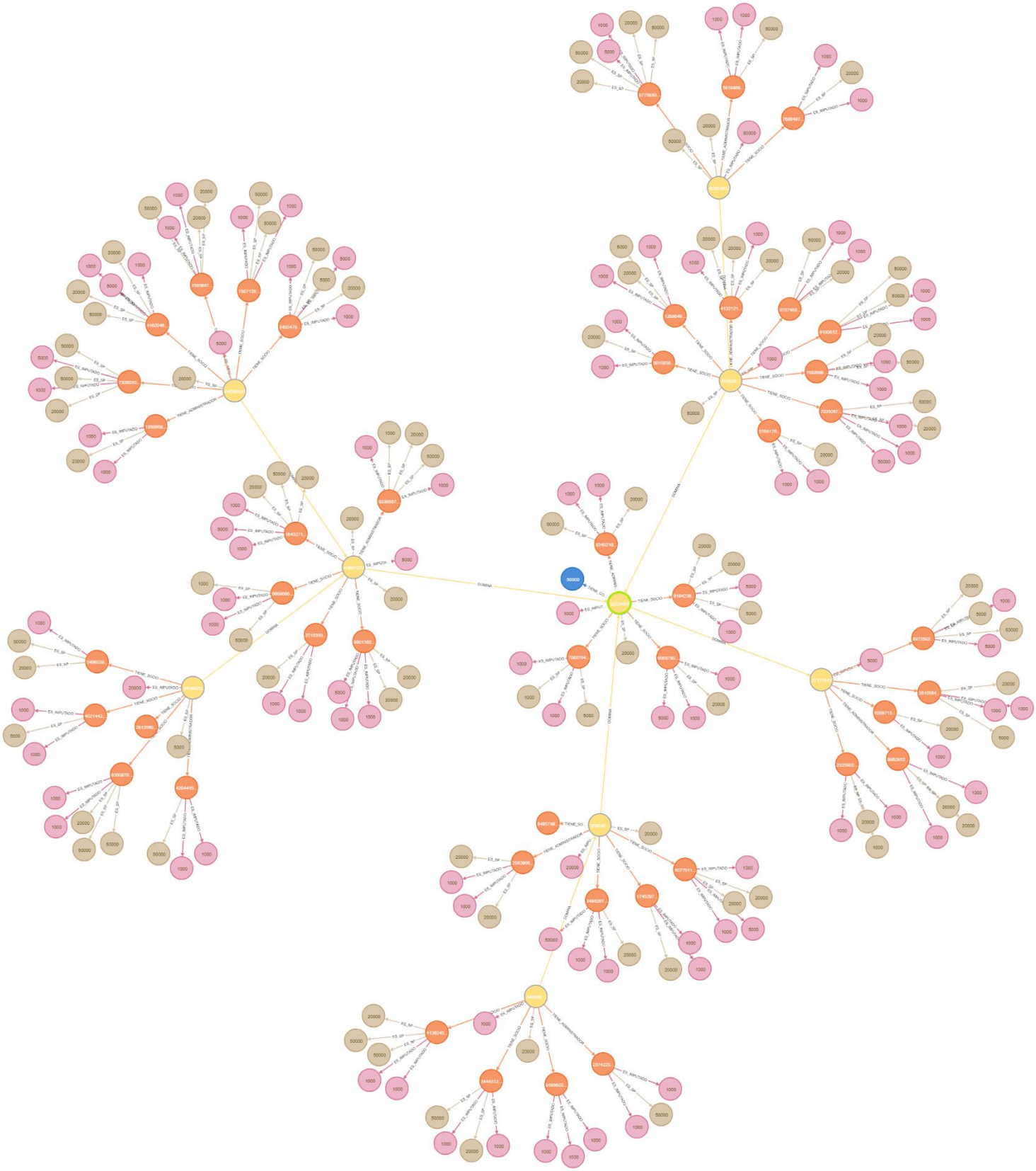
Primera versión de la consulta – No se muestran familiares.

Deudor con NIF 5790370D



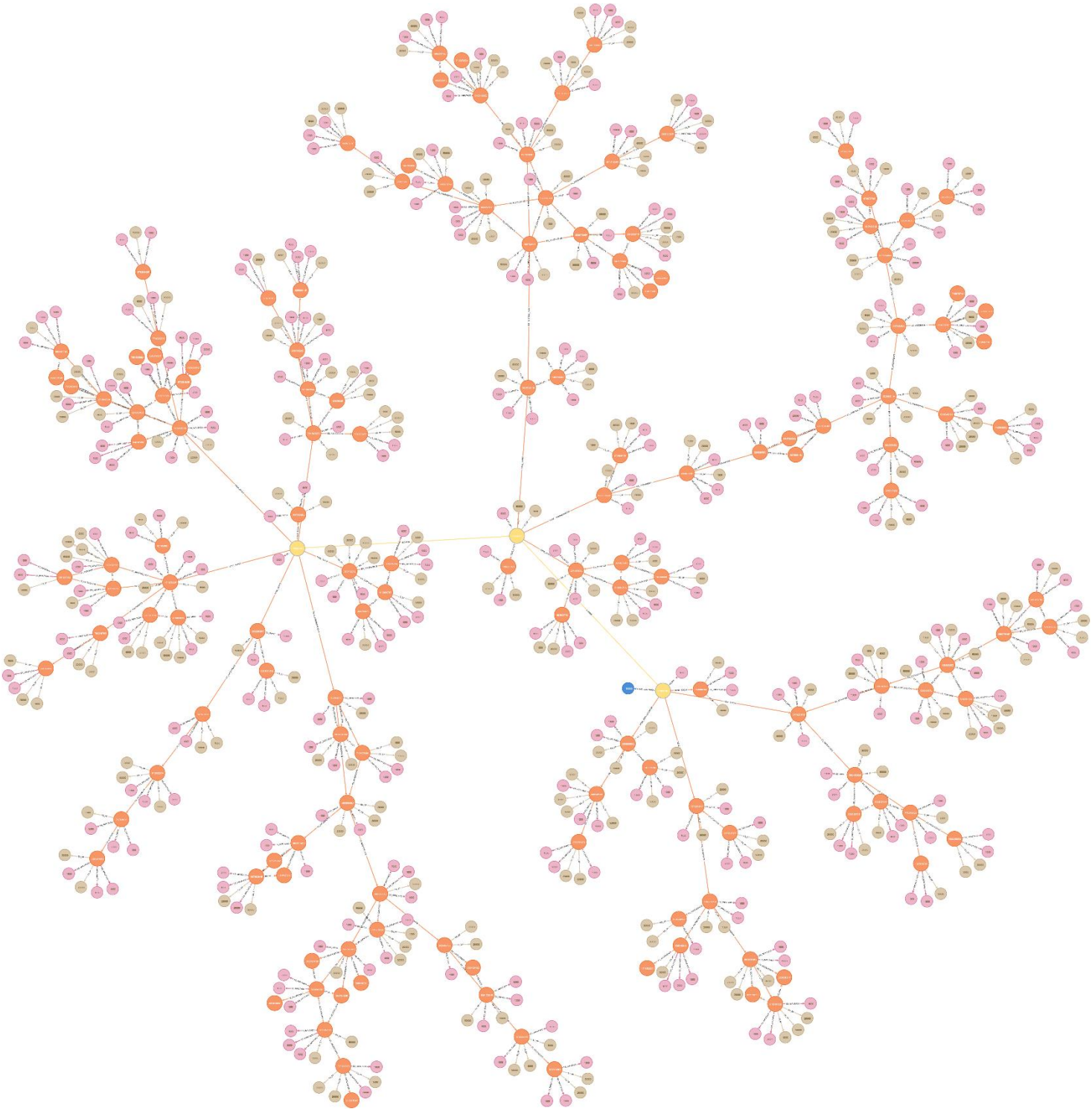
Primera versión de la consulta – No se muestran familiares.

La misma consulta pero con un deudor con más datos relacionados - NIF 3232480E



Segunda versión de la consulta – Con familiares (personas asistidas, conyuges, hijos).

Deudor con NIF *5790370D*



6.3.2 - Solvencia para un conjunto de deudores (formato tabla).

Objetivo

En este apartado se quiere analizar un conjunto de deudores en una misma consulta. La entrada de esta consulta será una lista de NIFs de cada deudor a analizar. La salida será una tabla con la siguiente estructura:

Deudor	Persona relacionada	Caminos	Bienes	Importe	Importe total persona	Importe total deudor
Deudor1	Persona1	[, ,]	Bien1	Importe1	TotalP1	TotalD1
Deudor1	Persona1	[, ,]	Bien2	Importe2	TotalP1	TotalD1
Deudor1	Persona2	[, ,]	Bien1	Importe1	TotalP2	TotalD1
. . .	. . .	. . .	. . .	. . .	. . .	. . .
Deudor2						TotalD2
. . .	. . .	. . .	. . .	. . .	. . .	. . .

En la primera columna se mostrará el NIF del deudor, en la segunda el NIF de la persona con la que se relaciona (puede ser física o jurídica), en la tercera se mostrarán todos los caminos posibles desde el deudor hasta la persona, en la cuarta se mostrarán los bienes de cada persona (modelos o declaraciones), en la quinta se mostrará el importe del bien, en la sexta se mostrará el importe total de los bienes de la persona y en la última se muestra el importe total de los bienes del deudor.

En definitiva, para cada deudor habrá tantas filas como bienes relacionados tenga.

Confección

Para realizar este supuesto se ha utilizado el **Código 33**. En éste se comienza extrayendo los deudores cuyos NIFs vienen dados por los elementos de la lista. Por cada deudor se obtiene el conjunto de empresas a las que domina. Este agrupamiento se realiza en la cláusula **WITH**. Para realizar una función de agrupación respecto a un elemento, antes de la función se debe especificar el elemento por el que se quiere agrupar. Es decir, en este caso se pone la variable del deudor junto con el **COLLECT**, para que el agrupamiento de las empresas dominadas se haga por cada deudor.

Al final de estas sentencias se obtiene tres filas, una por cada deudor, con la lista de empresas dominadas de cada deudor.

```
UNWIND ['7980415J','6612589C','5790370D'] AS NIF
MATCH (deudor {NIF:NIF})
OPTIONAL MATCH (deudor)-[:DOMINA*]->(dominadas: Persona_juridica)
WITH  distinct(deudor) AS deudor, collect(distinct(dominadas.NIF)) + collect(deudor.NIF) AS empresas
```

A continuación, se obtienen las empresas dominantes por cada deudor y se agregan estas empresas dominantes a las listas de empresas creadas anteriormente de cada deudor. Es decir, las empresas dominantes del primer deudor se agregarán a la lista del primer deudor, las del segundo deudor a la segunda lista, etc.

```
OPTIONAL MATCH (dominantes: Persona_juridica)-[:DOMINA*]->(deudor)
WITH  distinct(deudor) AS deudor, empresas + collect(distinct(dominantes.NIF)) AS empresas
```

Como puede haber ciclos en este tipo de relaciones (véase **Figura 38**), puede que una misma empresa se encuentre dos veces en la lista (una por ser dominada y otra por ser dominante). En el caso de la figura el nodo rojo se encontraría dos veces. Por ello se debe recorrer la lista y borrar duplicados.

```
UNWIND empresas AS em
WITH deudor, collect(distinct(em)) AS empresas
```

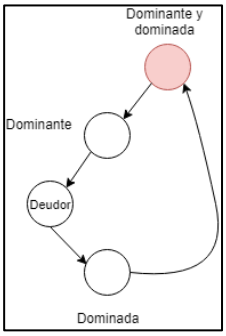


Figura 38 *- Ciclos en relaciones de dominancias.*

En las siguientes sentencias se obtiene los socios y administradores de las personas almacenadas en la lista *empresas*. Las personas resultantes de esta búsqueda (variable *personas*) se agregan con la lista de *empresas* y se almacena en una nueva variable *listPersonas*. En la lista *listPersonas* se tendrá tanto empresas como socios y administradores.

```
UNWIND empresas AS empresa
OPTIONAL MATCH (:Persona_juridica {NIF:empresa})-[:TIENE_SOCIO|TIENE_ADMINISTRADOR]->(personas:Persona_fisica)
WITH distinct(deudor) AS deudor, collect(distinct(personas.NIF))+empresas AS listPersonas
```

En el siguiente bloque se obtienen las descendencias de los socios y administradores (se incluyen las parejas y las personas asistidas o asistentes) y se agregan a la lista *listPersonas*.

```
UNWIND  listPersonas AS persona
OPTIONAL MATCH (:Persona_fisica {NIF:persona})-[:ES_ASCENDENTE*]->(descendientes:Persona_fisica)
OPTIONAL MATCH (descendientes)-[:ES_CONYUGE]-(conyuges:Persona_fisica)
OPTIONAL MATCH (descendientes)-[:ES_ASISTIDO_POR]-(asistidos:Persona_fisica)
WITH distinct(deudor) AS deudor, collect(distinct(descendientes.NIF)) + collect(distinct(conyuges.NIF)) + collect(distinct(asistidos.NIF)) + listPersonas AS listPersonas
```

Se vuelve a eliminar posibles duplicados.

```
UNWIND listPersonas AS p
WITH deudor, collect(distinct(p)) AS listPersonas
```

Para obtener las ascendencias, se realiza el mismo proceso que con las descendencias.

```
UNWIND  listPersonas AS persona
OPTIONAL MATCH (:Persona_fisica {NIF:persona})<-[:ES_ASCENDENTE*]-(ascendientes:Persona_fisica)
OPTIONAL MATCH (ascendientes)-[:ES_CONYUGE]-(conyuges:Persona_fisica)
OPTIONAL MATCH (ascendientes)-[:ES_ASISTIDO_POR]-(asistidos:Persona_fisica)
WITH distinct(deudor) AS deudor, collect(distinct(ascendientes.NIF)) + collect(distinct(conyuges.NIF)) + collect(distinct(asistidos.NIF)) + listPersonas AS listPersonas

UNWIND listPersonas AS p
WITH distinct(deudor) AS deudor, collect(distinct(p)) AS listPersonas
```

Los siguientes dos bloques tienen como objetivo calcular el importe total de cada deudor. En el primer bloque se extraen todas las personas y se almacenan en una nueva lista. En las listas anteriores se guardaban los NIF, no los nodos en sí, por ello se hace este paso.

Y en el segundo, se obtienen los bienes de cada persona y se hace un sumatorio del importe de todos los bienes de las personas por cada deudor.

```
UNWIND  listPersonas AS p
OPTIONAL MATCH (p1:Persona_fisica {NIF:p})
OPTIONAL MATCH (p2:Persona_juridica {NIF:p})
WITH deudor, collect(p1) + collect(p2) AS lista, listPersonas

UNWIND lista AS p
OPTIONAL MATCH (p)-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH distinct(deudor) AS deudor, sum(toInteger(bienes.importe)) AS totalDeudor, listPersonas
```

Estos dos bloques se podrían realizar en un sólo bloque, como se muestra a continuación. No obstante, no se ha realizado de esta forma, ya que, se obtiene un peor tiempo de respuesta. Esto se debe a que en la extracción de la persona "*OPTIONAL MATCH (persona {NIF:p})*" no se le especifica la etiqueta del nodo por el cual filtrar, por lo que hace una búsqueda por todos los tipos de nodos, concluyendo así en un peor rendimiento.

```
UNWIND listPersonas AS p
OPTIONAL MATCH (persona {NIF:p})
OPTIONAL MATCH (persona)-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH distinct(deudor) AS deudor, sum(toInteger(bienes.importe)) AS totalDeudor, listPersonas
```

Para el cálculo total de importe de bienes de cada persona, se han utilizado los siguiente dos bloques de código. Tiene la misma estructura que los de arriba. No obstante, la única diferencia es que a la hora de realizar el sumatorio, se agrupa por deudor y persona.

```
UNWIND listPersonas AS p
OPTIONAL MATCH (p1:Persona_fisica {NIF:p})
OPTIONAL MATCH (p2:Persona_juridica {NIF:p})
WITH deudor, totalDeudor, collect(p1) + collect(p2) AS lista, listPersonas

UNWIND lista AS p
OPTIONAL MATCH (p)-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH distinct(deudor) AS deudor, p, sum(toInteger(bienes.importe)) AS total, totalDeudor
WITH distinct(deudor) AS deudor, collect([deudor,totalDeudor, p.NIF, total]) AS informacion
```

Seguidamente se obtiene la información de los bienes de las personas físicas y personas jurídicas. Cabe destacar, que denuevo se podría resumir estos dos bloques en uno sólo, pero con peor rendimiento.

```
UNWIND informacion AS candidatos
MATCH (p:Persona_fisica {NIF:candidatos[2]})-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH  distinct(deudor) AS deudor, p, collect([deudor, candidatos[1], p, null, candidatos[3], bienes, bienes.importe ]) AS informacion1, informacion

UNWIND informacion AS candidatos
MATCH (p2:Persona_juridica {NIF:candidatos[2]})-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH  distinct(deudor) AS deudor, p2, informacion1 + collect([candidatos[0],candidatos[1], p2, null, candidatos[3], bienes, bienes.importe ]) AS informacion
```

Posteriormente, se obtiene todos los caminos entre el deudor y la persona relacionada. Para ello se utiliza una función de la librería de *APOC* que dispone *Neo4j*. Los parámetros de esta función son: nodo de partida, nodo destino, los tipos de relaciones por las que se puede navegar y, finalmente, el número máximo de saltos permitidos. Mientras mayor número de saltos, mayor será el tiempo de respuesta de la consulta.

```
UNWIND informacion AS x
MATCH (e:Persona_fisica {NIF:x[2].NIF})
call apoc.algo.allSimplePaths(x[0], e, 'DOMINA|TIENE_SOCIO|TIENE_ADMINISTRADOR|ES_CONYUGE|ES_ASCENDENTE|ES_ASISTIDO_POR', 10) YIELD path
```

Finalmente, se muestra todos los datos que se han ido obteniendo durante todos los pasos anteriores, ordenados por el deudor y la persona relacionada.

```
RETURN  x[0] AS deudor, x[2] AS persona_relacionada,
        collect([re IN relationships(path) | [ STARTNODE(re).NIF, type(re), ENDNODE(re).NIF]]) AS caminos,
        x[5] AS bienes, x[6] AS importe, x[4] AS totalPersona, x[1] AS totalDeudor
ORDER BY deudor, persona_relacionada
```

La consulta íntegra se puede ver en el **Código 33**.

```
UNWIND ['7980415J','6612589C','5790370D'] AS NIF
MATCH (deudor {NIF:NIF})
OPTIONAL MATCH (deudor)-[:DOMINA*]->(dominadas: Persona_juridica)
WITH  distinct(deudor) AS deudor, collect(distinct(dominadas.NIF)) + collect(deudor.NIF) AS empresas

OPTIONAL MATCH (dominantes: Persona_juridica)-[:DOMINA*]->(deudor)
WITH  distinct(deudor) AS deudor, empresas + collect(distinct(dominantes.NIF)) AS empresas

UNWIND empresas AS em
WITH deudor, collect(distinct(em)) AS empresas

UNWIND empresas AS empresa
OPTIONAL MATCH (:Persona_juridica {NIF:empresa})-[:TIENE_SOCIO|TIENE_ADMINISTRADOR]->(personas:Persona_fisica)
WITH distinct(deudor) AS deudor, collect(distinct(personas.NIF))+empresas AS listPersonas

UNWIND  listPersonas AS persona
OPTIONAL MATCH (:Persona_fisica {NIF:persona})-[:ES_ASCENDENTE*]->(descendientes:Persona_fisica)
OPTIONAL MATCH (descendientes)-[:ES_CONYUGE]-(conyuges:Persona_fisica)
OPTIONAL MATCH (descendientes)-[:ES_ASISTIDO_POR]-(asistidos:Persona_fisica)
WITH distinct(deudor) AS deudor,
     collect(distinct(descendientes.NIF)) + collect(distinct(conyuges.NIF)) +
     collect(distinct(asistidos.NIF)) + listPersonas AS listPersonas

UNWIND listPersonas AS p
WITH deudor, collect(distinct(p)) AS listPersonas

UNWIND  listPersonas AS persona
OPTIONAL MATCH (:Persona_fisica {NIF:persona})<-[:ES_ASCENDENTE*]-(ascendientes:Persona_fisica)
OPTIONAL MATCH (ascendientes)-[:ES_CONYUGE]-(conyuges:Persona_fisica)
OPTIONAL MATCH (ascendientes)-[:ES_ASISTIDO_POR]-(asistidos:Persona_fisica)
WITH distinct(deudor) AS deudor,
     collect(distinct(ascendientes.NIF)) + collect(distinct(conyuges.NIF)) +
     collect(distinct(asistidos.NIF)) + listPersonas AS listPersonas

UNWIND listPersonas AS p
WITH distinct(deudor) AS deudor, collect(distinct(p)) AS listPersonas

UNWIND  listPersonas AS p
OPTIONAL MATCH (p1:Persona_fisica {NIF:p})
OPTIONAL MATCH (p2:Persona_juridica {NIF:p})
WITH deudor, collect(p1) + collect(p2) AS lista, listPersonas

UNWIND lista AS p
OPTIONAL MATCH (p)-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH distinct(deudor) AS deudor, sum(toInteger(bienes.importe)) AS totalDeudor, listPersonas

UNWIND listPersonas AS p
OPTIONAL MATCH (p1:Persona_fisica {NIF:p})
OPTIONAL MATCH (p2:Persona_juridica {NIF:p})
WITH deudor, totalDeudor, collect(p1) + collect(p2) AS lista, listPersonas

UNWIND lista AS p
OPTIONAL MATCH (p)-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH distinct(deudor) AS deudor, p, sum(toInteger(bienes.importe)) AS total, totalDeudor
WITH distinct(deudor) AS deudor, collect([deudor,totalDeudor, p.NIF, total]) AS informacion

UNWIND informacion AS candidatos
MATCH (p:Persona_fisica {NIF:candidatos[2]})-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH  distinct(deudor) AS deudor, p,
      collect([deudor, candidatos[1], p, null, candidatos[3], bienes, bienes.importe ]) AS informacion1, informacion

UNWIND informacion AS candidatos
MATCH (p2:Persona_juridica {NIF:candidatos[2]})-[:ES_IMPUTADO|ES_SP]->(bienes)
WITH  distinct(deudor) AS deudor, p2,
     informacion1 + collect([candidatos[0],candidatos[1], p2, null, candidatos[3], bienes, bienes.importe ]) AS informacion

UNWIND informacion AS x
MATCH (e:Persona_fisica {NIF:x[2].NIF})
call apoc.algo.allSimplePaths(x[0], e, 'DOMINA|TIENE_SOCIO|TIENE_ADMINISTRADOR|ES_CONYUGE|ES_ASCENDENTE|ES_ASISTIDO_POR', 3) YIELD path

RETURN  x[0] AS deudor, x[2] AS persona_relacionada,
       collect([re IN relationships(path) | [ STARTNODE(re).NIF, type(re), ENDNODE(re).NIF]]) AS caminos,
       x[5] AS bienes, x[6] AS importe, x[4] AS totalPersona, x[1] AS totalDeudor
ORDER BY deudor, persona_relacionada
```

Código 33 - *Consulta para mostrar la tabla explicada.*

Resultado

Con la consulta expuesta se consigue una tabla con aproximadamente 677 lineas. Es por ello por lo que en la figura de la parte inferior no se mostrarán todas ellas. No obstante, se puede apreciar cómo se estructura el esquema anteriormente explicado.

Utilizando el visualizador de *Neo4j* puede que la visualización y el análisis de estos datos sea engorrosa, ya que, no te permite ningún tipo de interacción con la tabla mostrada. No obstante, *Neo4j* ofrece la posibilidad de exportar la tabla a un fichero *csv* y así poder abrir el fichero con otra herramienta más gráfica y cómoda como podría ser *Excel*.

"deudor"	"persona_relacionada"	"caminos"	"bienes"	"importe"	"totalPersona"	"totalDeudor"
{"NIF":"5790370D"}	{"NIF":"76468997C"}	[[[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]]]	{"importe":"50000","tipo":"inmueble","id":"34794"}	"50000"	72000	12083000
{"NIF":"5790370D"}	{"NIF":"76468997C"}	[[[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]]]	{"importe":"20000","tipo":"vehiculo","id":"34793"}	"20000"	72000	12083000
{"NIF":"5790370D"}	{"NIF":"76468997C"}	[[[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]]]	{"modelo":"159","importe":"1000","id":"53494"}	"1000"	72000	12083000
{"NIF":"5790370D"}	{"NIF":"76468997C"}	[[[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_SOCIO","76468997C"]]]	{"modelo":"180","importe":"1000","id":"53493"}	"1000"	72000	12083000
{"NIF":"5790370D"}	{"NIF":"19870311Y"}	[[[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]]]	{"importe":"50000","tipo":"inmueble","id":"86653"}	"50000"	53000	12083000
{"NIF":"5790370D"}	{"NIF":"19870311Y"}	[[[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]]]	{"importe":"1000","tipo":"cuenta","id":"86652"}	"1000"	53000	12083000
{"NIF":"5790370D"}	{"NIF":"19870311Y"}	[[[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]]]	{"modelo":"159","importe":"1000","id":"128380"}	"1000"	53000	12083000
{"NIF":"5790370D"}	{"NIF":"19870311Y"}	[[[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]],[["7656901E","DOMINA","5790370D"],["7656901E","TIENE_SOCIO","56279964G"],["56279964G","ES_ASISTIDO_POR","19870311Y"]]]	{"modelo":"180","importe":"1000","id":"128379"}	"1000"	53000	12083000
{"NIF":"5790370D"}	{"NIF":"74786001P"}	[[[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]]]	{"importe":"20000","tipo":"inmueble","id":"138061"}	"20000"	22000	12083000
{"NIF":"5790370D"}	{"NIF":"74786001P"}	[[[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]]]	{"modelo":"180","importe":"1000","id":"197827"}	"1000"	22000	12083000
{"NIF":"5790370D"}	{"NIF":"74786001P"}	[[[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]],[["7656901E","DOMINA","5790370D"],["7283274V","DOMINA","7656901E"],["7283274V","TIENE_ADMINISTRADOR","74786001P"]]]	{"modelo":"159","importe":"1000","id":"197828"}	"1000"	22000	12083000
{"NIF":"5790370D"}	{"NIF":"76231662L"}	[[[["5790370D","TIENE_SOCIO","77126434V"],["77126434V","ES_ASCENDIENTE","58472621W"],["76231662L","ES_ASISTIDO_POR	{"importe":"20000","tipo":"mueble","id":"171666"}	"20000"	97000	12083000

6.4 - Tiempos de respuesta obtenidos

La comparación y medición de los tiempos de respuesta o de latencia de las consultas en *Neo4j* no resulta una tarea fácil. Esto se debe a que es muy difícil lanzar dos veces una misma consulta en las mismas condiciones.

Por una parte, *Neo4j* optimiza el tiempo de respuesta de las consultas utilizando la memoria caché. Es habitual que la primera ejecución de una consulta sea mucho más lenta que la segunda ejecución de la misma. Esto se debe a que en un arranque frío, el servidor no tiene ningún elemento almacenado en caché y por lo tanto necesita acudir al disco para obtener los registros. Una vez almacenados todos estos registros en memoria, en la segunda ejecución no debe acudir al disco, por lo que el tiempo disminuye considerablemente, véase **Figura 39**.

Por otra parte, Neo4j también almacena los planes de ejecución de las consultas que se lanzan. Cuando se manda a ejecutar una consulta, lo primero que se hace es determinar si la consulta se encuentra en la caché o no. En caso de encontrarla, *Neo4j* no deberá realizar un nuevo plan de ejecución, por lo que el tiempo de respuesta mejora.

Estas técnicas, a pesar de resultar ventajosas a la hora de lanzar consultas y de obtener los resultados en un tiempo asequible, dificultan las posibles comparaciones de los tiempos, puesto que no se puede saber qué es lo que se encuentra almacenado en cache cuando se lanza la consulta.

En la siguiente tabla se puede apreciar cual es el comportamiento del tiempo de respuesta cuando se ejecuta una consulta diez veces consecutivamente. Además, se considera que en la primera ejecución la caché se encuentra vacía.

	1	2	3	4	5	6	7	8	9	10
Sup 1 - Primera consulta	298,0	14,0	12,0	9,0	8,0	8,0	7,0	7,0	7,0	5,0
Sup 2 - Primera consulta	14,0	3,0	1,0	1,0	1,0	1,0	2,0	1,0	1,0	1,0
Sup 2 - Segunda consulta	188,0	5,0	2,0	2,0	2,0	2,0	2,0	1,0	1,0	2,0
Sup 3 - Consulta grafo v1	210,0	23,0	6,0	6,0	6,0	5,0	4,0	6,0	4,0	4,0
Sup 3 - Consulta grafo v2	1510,0	685,0	81,0	59,0	52,0	38,0	31,0	30,0	31,0	26,0

***Tabla 9** - Tiempos de respuesta en ms de diez consultas consecutivas[9].*

[9] Estos tiempos son obtenidos de una instancia de Neo4j Community.

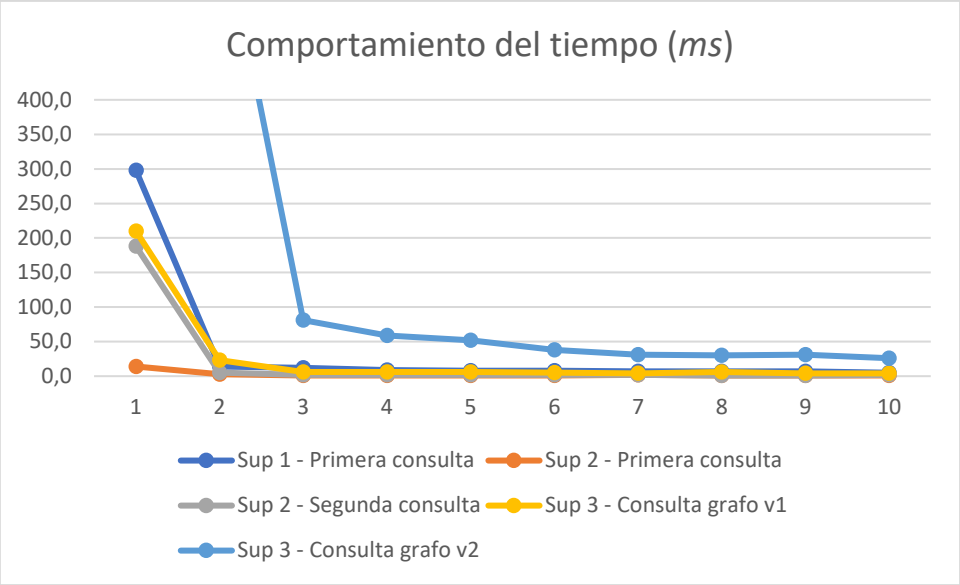


Figura 39 *- Gráfico que muestra los tiempos de las consultas tras diez ejecuciones consecutivas.*

Centrándonos en los supuestos explicados, ha resultado difícil comparar los resultados obtenidos en la edición Enterprise y en la edición Community. Para comparar estos tiempos, lo ideal sería que o bien en la caché se encontrase los mismo registros y planes de ejecución tanto en la instancia de Enterprise como en la de Community, o bien que las dos memorias caché estuviesen vacías. No obstante, limpiar la cache y saber el estado de la misma es una tarea complicada.

Como bien se ha explicado en capítulos anteriores, se han utilizado dos contenedores para desplegar las dos ediciones de Neo4j. Se quiere lanzar una misma consulta en las dos instancias, pero para ello se debe lanzar bajo las mismas condiciones. Para conseguirlo, antes de lanzar la consulta se ha parado el contenedor y se ha vuelto a lanzar el mismo. Con esto parece que la memoria cache se borra, aunque no se pueda asegurar al cien por cien.

Esto se ha hecho para cada consulta y los tiempos de respuestas se pueden apreciar en la **Tabla 10**. En esta se puede ver que para una misma consulta y un mismo entorno se obtienen tiempos bastantes diferentes. Esto recalca el hecho de que es difícil obtener un tiempo de respuesta uniforme y fiable, este depende de muchos factores y recursos.

		COMMUNITY (*ms*)				ENTERPRISE (*ms*)			
		1	**2**	**3**	**MEDIA**	**1**	**2**	**3**	**MEDIA**
SUPUESTO 1	**Primera consulta**	298	201	669	389,33	179	214	585	326,00
SUPUESTO 2	**Primera consulta**	12	14	21	15,67	10	12	16	12,67
	Segunda consulta	188	197	156	180,33	134	141	126	133,67
SUPUESTO 3	**Consulta grafo v1**	210	213	1160	527,67	241	882	188	437,00
	Consulta grafo v2	1510	1200	1023	1244,33	866	749	1103	906,00
	Consulta tabla	20479	15506	15903	17296,00	17959	13973	13534	15155,33

Tabla 10 *- Tiempo de respuesta de diferentes primeras ejecuciones.*

De todo esto se pueden obtener varias conclusiones. Por un parte, el rendimiento de este tipo de base de datos es muy bueno ya que se obtiene tiempos muy pequeños para la realización de las consultas. En las cuatro primeras consultas, los tiempos de respuesta no superan el segundo. Sin embargo, las dos últimas sí, pero comparándolo con la cantidad de datos con los que trabajan, este tiempo es bastante reducido.

Por otra parte, si bien es verdad que, en la importación de los datos, la instancia Enterprise obtenía un rendimiento mucho mejor que la instancia de *Community*, en estos tipos de consultas la diferencia es mínima (cogiendo la media un 20-30% mejor).

Finalmente, como conclusión de todo esto, se puede afirmar que, aunque hacer una comparativa de los tiempos sea una tarea difícil, al final todos los tiempos concluyen en un mismo hecho: son tiempos pequeños para la cantidad de datos y relaciones que se maneja.

Capítulo 7 - Conclusiones y lineas futuras

El objetivo principal de este Trabajo Fin de Grado se ha basado en el despliegue de una base de datos orientada a grafos para facilitar todo el proceso de investigación de Derivaciones de Responsabilidad. Para ello, se ha tenido que realizar una prospección de este tipo de bases de datos para posteriormente seleccionar aquella que mejor se adecuase. Seguidamente se ha diseñado el nuevo modelo orientado a grafos y se ha desarrollado un sistema para llevar a cabo la generación de datos sintéticos aleatorios. Finalmente, se han importado estos datos y se han realizado los diferentes supuestos propuestos.

Asimismo, con el desarrollo de esta nueva herramienta, se ha conseguido cubrir las diferentes necesidades que se habían presentado.

Por una parte, se ha dotado al departamento de Gestión de Deudas de una herramienta la cual ofrece una visualización gráfica de las relaciones existentes entre las diferentes entidades. Es más, no sólo es una visualización, sino que también presenta la posibilidad de interactuar con el grafo. De esta manera, lo que anteriormente se realizaba mediante tablas relaciones poco confortables, ahora se puede realizar mediante la estructura de un grafo.

Por otra parte, se ha reducido considerablemente el número de operaciones y consultas a realizar. Con una misma consulta se pueden obtener tantos niveles de información como se quiera. Es decir, para un mismo NIF y con una única consulta se pueden obtener las personas relacionadas a éste, los bienes, información personal de éste y de sus personas relacionadas. Esto no podía realizarse anteriormente, ya que para obtener todo ese conjunto de datos debían de ir NIF a NIF individualmente.

De igual manera, también se ha conseguido combatir toda esa dispersión de la información y conseguir así un único punto de partida. Es decir, toda la información necesaria para estas investigaciones se encuentra recogida en un mismo grafo.

Finalmente, con esta tecnología se ha mejorado el rendimiento para aquellas búsquedas transversales, es decir, aquellos casos en los que se necesita dar diferentes saltos entre personas.

De cara a lineas futuras, en primer lugar, se proyecta la necesidad de nuevos supuestos que suponen una modificación del modelo y una nueva generación de datos. Concretamente, se pretende añadir temporalidad al grafo, haciendo así, que ciertos nodos y relaciones dispongan de una nueva propiedad que refleje la fecha de estos elementos.

Por otro lado, una vez cubierta esta temporalidad del grafo, se plantea desplegar toda esta herramienta, pero con datos reales ofrecidos por la Hacienda Tributaria de Navarra.

Finalmente, se pretende llevar todo este proyecto al entorno de trabajo de los empleados del departamento de Gestión de Deudas.

Referencias y bibliografía

[1] Wikipedia, «Wikipedia - Historia de los grafos.,» [En línea]. Available: https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos#cite_note-3. [Último acceso: abril 2019].

[2] Wikipedia, «Wikipedia - Graph Database - History,» [En línea]. Available: https://en.wikipedia.org/wiki/Graph_database#History. [Último acceso: 01 mayo 2019].

[3] B. G. Inc, «History of Databases and Graph Database,» 09 noviembre 2016. [En línea]. Available: https://bitnine.net/blog-graph-database/history-of-databases-and-graph-database/. [Último acceso: 01 mayo 2019].

[4] Wikipedia, «Wikipedia - NoSQL,» [En línea]. Available: https://es.wikipedia.org/wiki/NoSQL. [Último acceso: 01 mayo 2019].

[5] J. Webber, «A Note on Native Graph Databases,» 21 septiembre 2017. [En línea]. Available: https://neo4j.com/blog/note-native-graph-databases/.

[6] D. McCreary, «How to Explain Index-free Adjacency to Your Manager,» 15 marzo 2018. [En línea]. Available: https://medium.com/@dmccreary/how-to-explain-index-free-adjacency-to-your-manager-1a8e68ec664a.

[7] J. Barrasa, «DZone - RDF Triple Stores vs. Labeled Property Graphs: What's the Difference?,» 26 enero 2018. [En línea]. Available: https://dzone.com/articles/rdf-triple-stores-vs-labeled-property-graphs-whats.

[8] DB-ENGINES, «DB-ENGINES - System Properties Comparison ArangoDB vs. Neo4j vs. OrientDB,» [En línea]. Available: https://db-engines.com/en/system/ArangoDB%3bNeo4j%3bOrientDB. [Último acceso: 13 abril 2019].

[9] DB-ENGINES, «DB-ENGINES - System Properties Comparison Neo4j vs. TigerGraph,» [En línea]. Available: https://db-engines.com/en/system/Neo4j%3BTigerGraph. [Último acceso: 13 abril 2019].

[10] M. Bachman, «GraphAware - Modelling Data in Neo4j: Bidirectional Relationships,» 11 octubre 2013. [En línea]. Available: https://graphaware.com/neo4j/2013/10/11/neo4j-bidirectional-relationships.html.

[11] I. Robinson, J. Webber y E. Eifrem, «Graph Databases - NEW OPPORTUNITIES FOR CONNECTED DATA,» de *Graph Databases - NEW OPPORTUNITIES FOR CONNECTED DATA*, 2 ed., O'Reilly, 2015.

[12] Neo4j, «Neo4j - Suscriptions,» [En línea]. Available: https://neo4j.com/subscriptions/. [Último acceso: 01 mayo 2019].

[13] O. Panzarino, Learning Cypher - Write powerful and efficient queries for Neo4j with Cypher, its offical query language., Packt Publishing Ltd., 2014.

[14] Neo4j, «Neo4j - Constraints,» [En línea]. Available: https://neo4j.com/docs/cypher-manual/3.5/schema/constraints/. [Último acceso: 01 mayo 2019].

[15] C. G. d. P. J. Real Academia Española, «DEJ (RAE) - Persona física,» [En línea]. Available: https://dej.rae.es/lema/persona-física. [Último acceso: 09 mayo 2019].

[16] C. G. d. P. J. Real Academia Española, «DEJ (RAE) - Persona jurídica,» [En línea]. Available: https://dej.rae.es/lema/persona-jurídica. [Último acceso: 09 mayo 2019].

[17] C. G. d. P. J. Real Academia Española, «DEJ (RAE) - Declaración,» [En línea]. Available: https://dej.rae.es/lema/declaración.

[18] C. G. d. P. J. Real Academia Española, «DEJ (RAE) - Deuda tributaria,» [En línea]. Available: https://dej.rae.es/lema/deuda-tributaria. [Último acceso: 09 mayo 2019].

[19] Neo4j, «Neo4j - Memory configuration,» [En línea]. Available: https://neo4j.com/docs/operations-manual/current/performance/memory-configuration/. [Último acceso: 7 mayo 2019].

[20] G. Anadiotis, «Linked Data Orchestration,» [En línea]. Available: http://linkeddataorchestration.com/the-year-of-the-graph/.

[21] ArangoDB, «ArangoDB manual - ArangoDB Web UI,» [En línea]. Available: https://www.arangodb.com/arangodb-training-center/arangodb-web-ui/. [Último acceso: 12 abril 2019].

[22] OrientDB, «OrientDB manual - Graph editor,» [En línea]. Available: https://orientdb.com/docs/2.0/orientdb-studio.wiki/Graph-Editor.html. [Último acceso: 12 abril 2019].

[23] TigerGraph, «TigerGraph manual - Explore Graph,» [En línea]. Available: https://docs.tigergraph.com/ui/graphstudio/explore-graph. [Último acceso: 12 abril 2019].

[24] DB-ENGINES, «DB-ENGINES - Ranking of Graph DBMS,» [En línea]. Available: https://db-engines.com/en/ranking/graph+dbms. [Último acceso: 2012 abril 2019].

[25] DB-ENGINES, «DB-ENGINES - Ranking of Graph DBMS,» [En línea]. Available: https://db-engines.com/en/ranking/graph+dbms. [Último acceso: 12 abril 2019].

[26] DB-ENGINES, «DB-ENGINES - Trend of Graph DBMS Popularity,» [En línea]. Available: https://db-engines.com/en/ranking_trend/graph+dbms. [Último acceso: 12 abril 2019].

[27] TigerGraph, «TigerGraph - Introducing TigerGraph,» [En línea]. Available: https://www.tigergraph.com/product/. [Último acceso: 12 abril 2019].

[28] A. Deutsch, Y. Xu, M. Wu y V. Lee, «TigerGraph: A Native MPP Graph Database,» 24 enero 2019. [En línea]. Available: https://arxiv.org/pdf/1901.08248.pdf.

[29] TigerGraph, «TigerGraph - Common Applications,» [En línea]. Available: https://docs.tigergraph.com/dev/gsql-examples/common-applications. [Último acceso: 12 abril 2019].

[30] ArangoDB, «ArangoDB - Graph traversals in AQL,» [En línea]. Available: https://docs.arangodb.com/3.4/AQL/Graphs/Traversals.html. [Último acceso: 12 abril 2019].

[31] ArangoDB, «ArangoDB manual - Concepts,» [En línea]. Available: https://docs.arangodb.com/3.4/Manual/DataModeling/Concepts.html. [Último acceso: 12 abril 2019].

[32] ArangoDB, «ArangoDB manual - Index basics,» [En línea]. Available: https://docs.arangodb.com/3.4/Manual/Indexing/IndexBasics.html. [Último acceso: 12 abril 2019].

[33] «Wikipedia - OrientDB,» [En línea]. Available: https://en.wikipedia.org/wiki/OrientDB. [Último acceso: 12 abril 2019].

[34] OrientDB, «OrientDB manual - PLocal Storage,» [En línea]. Available: http://orientdb.com/docs/2.2.x/Paginated-Local-Storage.html. [Último acceso: 12 abril 2019].

[35] OrientDB, «OrientDB manual - SQL Reference,» [En línea]. Available: http://orientdb.com/docs/2.2.x/SQL.html. [Último acceso: 12 abril 2019].

[36] ArangoDB, «ArangoDB manual - Native vs Layered,» [En línea]. Available: https://www.arangodb.com/why-arangodb/native-multi-model-database-advantages/. [Último acceso: 12 abril 2019].

[37] Wikipedia, «Wikipedia - Relational database,» [En línea]. Available: https://en.wikipedia.org/wiki/Relational_database.

Ilustraciones

Códigos

Tablas