



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Curso Académico 2017/2018

Trabajo Fin de Grado

**ESTUDIO COMPARATIVO DE BASES DE DATOS NoSQL:
MONGODB VS NEO4J**

Autor: Ester Cortés García

Directores: Belén Vela Sánchez

RESUMEN

El objetivo principal de este Trabajo de Fin de Grado de Ingeniería Informática es realizar un análisis comparativo del rendimiento de distintas consultas aplicadas a varios tipos de productos de Bases de Datos NoSQL (MongoDB y Neo4j). Con esto lo que se pretende es conseguir un mayor entendimiento del funcionamiento de los productos escogidos y ver en qué casos es mejor aplicar un tipo u otro.

Para ello, es necesario realizar un caso de estudio donde ejecutarán distintas consultas en ambos productos para medir los tiempos de respuesta y poder compararlos. Previamente, se realizarán un análisis histórico y un estudio de las Bases de Datos NoSQL para poder entender el por qué de su existencia y saber qué productos hay que escoger.

En primer lugar, para el caso de estudio, se ha usado *DBLP Computer Science Bibliography*, un dataset de gran tamaño y con numerosos atributos, que contiene artículos científicos de diferentes tipos.

Tras analizar la información almacenada en el *dataset* empleando EmEditor y ver cómo se encuentra estructurado el fichero, se han diseñado las consultas sobre las que se van a medir los tiempos de respuesta. En base a esas consultas, los productos NoSQL escogidos han sido MongoDB (Base de Datos Orientada a Documentos) y Neo4j (Base de Datos Orientada a Grafos), ya que el primer producto es adecuado para el almacenamiento de documentos, y el segundo es recomendable usarlo cuando son más importantes las relaciones entre los elementos que su contenido. Para este conjunto de datos, se puede obtener bastante información con la relación que existe entre publicaciones y autores.

A continuación, se han diseñado varios tipos de consultas, en concreto ocho, y se han analizado las distintas opciones de diseño de las bases de datos, escogiendo las que se han visto más adecuadas para obtener la información que nos interesa con las consultas definidas.

Seguidamente, se ha realizado la limpieza del dataset para adaptarlo a las necesidades de las consultas y de los productos escogidos.

Finalmente, se ha llevado a cabo un estudio comparativo para comprobar el rendimiento de los dos productos NoSQL escogidos mediante la medición de los tiempos de respuesta. Además, se ha realizado un análisis de la consecución de los objetivos propuestos para este trabajo.

ÍNDICE

1. INTRODUCCIÓN	5
1.1. Presentación	5
1.2. Objetivos	6
1.3. Método de trabajo	6
1.4. Medios hardware y software utilizados	7
1.5. Estructura de la memoria	8
2. ESTADO DEL ARTE	11
2.1. EVOLUCIÓN HISTÓRICA DE LAS BASES DE DATOS	11
2.1.1. Sistemas de Gestión de Datos Basados en Ficheros	11
2.1.2. Bases de Datos	12
2.1.2.1. Bases de Datos Jerárquicas	13
2.1.2.2. Bases de Datos en Red	13
2.1.2.3. Bases de Datos Relacionales	14
2.1.2.4. Bases de Datos Orientadas a Objetos	15
2.1.2.5. Bases de Datos Multidimensionales	15
2.1.2.6. Bases de Datos NoSQL	15
2.1.2.7. Bases de Datos NewSQL	16
2.2. PROPIEDADES DE LAS BASES DE DATOS	17
2.2.1. Propiedades ACID	17
2.2.2. Propiedades BASE	17
3. BASES DE DATOS NoSQL	18
3.1. INTRODUCCIÓN	18
3.2. CATEGORIZACIÓN DE LAS BASES DE DATOS NoSQL	21
3.2.1. Bases de Datos Clave-Valor	21
3.2.2. Bases de Datos Orientadas a Documentos	23
3.2.3. Base de Datos Orientadas a Grafos	24
3.2.4. Bases de Datos Familia de Columnas	25
3.3. COMPARATIVA DE LOS TIPOS DE BASES DE DATOS NoSQL	27
4. CASO DE ESTUDIO	29
4.1. CAPTURA DE LOS DATOS	29
4.2. PRODUCTOS NoSQL ESCOGIDOS	30
4.2.1. Instalación de MongoDB	31
4.2.2. Instalación de Neo4j	31
4.3. ASPECTO A COMPARAR	31
4.4. DISEÑO DE LAS CONSULTAS	32
4.5. DISEÑO DE LAS BASES DE DATOS	33

4.5.1. Diseño de la base de datos para MongoDB	33
4.5.2. Diseño de la base de datos para Neo4j	35
4.6. LIMPIEZA DEL DATASET	39
4.6.1. Limpieza de los datos para MongoDB	39
4.6.2. Limpieza de los datos para Neo4j	40
4.7. EJECUCIÓN DE LAS CONSULTAS	41
4.7.1. Ejecución de las consultas en MongoDB	41
4.7.2. Ejecución de las consultas en Neo4j	47
4.8. COMPARATIVA DE LOS RESULTADOS	55
5. CONCLUSIONES Y FUTUROS TRABAJOS	59
5.1. CONCLUSIONES	59
5.2. FUTUROS TRABAJOS	59
6. REFERENCIAS	61
7. ANEXOS	65
7.1. ANEXO A: CÓDIGO FUENTE XML_TO_JSON.PY	65
7.2. ANEXO B: CÓDIGO FUENTE XML_TO_CSV.PY	67
7.3. ANEXO C: CÓDIGO FUENTE IMPORT_DB_MONGODB.PY	70
7.4. ANEXO D: CÓDIGO FUENTE MONGODB_QUERY.PY	71

ÍNDICE DE FIGURAS

Figura 1. Método de trabajo	7
Figura 2. Estructura de un Sistema de Gestión de Ficheros	11
Figura 3. Esquema de una Base de Datos Clave-Valor	21
Figura 4. Esquema de una Base de Datos Clave-Valor con varios espacios de nombre	22
Figura 5. Ejemplos de documentos con distinta estructura	23
Figura 6. Base de Datos Orientada a Documentos con distintas colecciones	23
Figura 7. Esquema de una Base de Datos Orientada a Grafos	24
Figura 8. Esquema de una Base de Datos de Familia de Columnas	26
Figura 9. Fragmento de datos del fichero <i>dblp.xml</i>	30
Figura 10. Código para calcular los tiempos de respuesta en MongoDB	32
Figura 11. Esquema de la base de datos con una colección	34
Figura 12. Esquema de la base de datos con dos colecciones	34
Figura 13. Posibles nodos para la base de datos de Neo4j	35
Figura 14. Esquema de la base de datos Neo4j con un tipo de nodo	37
Figura 15. Esquema de la base de datos Neo4j con dos tipos de nodos	37
Figura 16. Esquema de la base de datos de Neo4j	39
Figura 17. Esquema resultante de la limpieza para la base de datos de MongoDB	40
Figura 18. Mandatos para iniciar el servidor de MongoDB en una terminal de Windows	41
Figura 19. Proceso de creación de índices para MongoDB	42
Figura 20. Carga de los ficheros CSV a la base de datos de Neo4j	48
Figura 21. Mandatos para arrancar el servidor de Neo4j	49
Figura 22. Interfaz de Neo4j en el navegador	49
Figura 23. Creación de índices para Neo4j	50
Figura 24. Consultas ejecutadas en las bases de datos	55

ÍNDICE DE TABLAS

Tabla 1: Comparativa entre Bases de Datos NoSQL y Bases de Datos Relacionales_____	20
Tabla 2. Comparativa de los tipos de Bases de Datos NoSQL_____	27
Tabla 3. Comparativa de los tiempos de respuesta_____	55

1. INTRODUCCIÓN

1.1 PRESENTACIÓN

Las Bases de Datos Convencionales de tipo relacional son estructuras tabulares que almacenan datos que se relacionan entre sí de la manera más eficiente posible. Son el modelo más popular de bases de datos. Sin embargo, en los últimos años, con el crecimiento de Internet y con la forma en la que las aplicaciones gestionan los datos, la escalabilidad y el rendimiento de las Bases de Datos Relacionales se ha visto mermado en algunos casos. Han surgido así nuevos modelos que permiten gestionar grandes cantidades de información, que pueden ser introducidos y extraídos rápidamente. En estos sistemas importa más la flexibilidad, la velocidad y la capacidad de escalado horizontal que otras cuestiones tradicionalmente cruciales como la consistencia o disponer de una estructura perfectamente definida para los datos. Éstas son las Bases de Datos NoSQL.

Con este Trabajo de Fin de Grado se va a realizar un análisis de los distintos tipos de bases de datos que han ido apareciendo a lo largo de los años, haciendo hincapié en las Bases de Datos NoSQL. Para comprobar el rendimiento de algunos de los productos más representativos de este tipo de bases de datos, se realizará un caso de estudio donde se analizará el rendimiento de dichos productos mediante la ejecución de diversas consultas para obtener una comparativa en cuanto a los tiempos de respuesta.

Para ello, en primer lugar, se estudia la evolución histórica de los sistemas de almacenamiento, que comienzan con los sistemas basados en ficheros y continúan con numerosos tipos de bases de datos que van surgiendo para solucionar los problemas de los modelos anteriores, hasta llegar a las Bases de Datos NoSQL, que conviven con las Bases de Datos Relacionales.

A continuación, se centrará el trabajo en las Bases de Datos No SQL. Se analizarán sus diferencias respecto a las Relacionales, se estudiarán los distintos tipos que podemos encontrar, siendo cuatro los principales: Bases de Datos Clave-Valor, Bases de Datos Orientadas a Documentos, Bases de Datos Orientadas a Grafos y Bases de Datos de Familia de Columnas. Sobre estos modelos se realizará una comparación para saber qué características comparten y cuáles los diferencian.

Tras el análisis, se realizará un caso de estudio cuyo objetivo es estudiar el rendimiento de diferentes productos NoSQL mediante el estudio de los tiempos de respuesta ante diferentes consultas. Para ello, será necesario escoger un *dataset* sobre el que se realizarán las consultas, decidir los productos de los que se quiere analizar el rendimiento (MongoDB como Base de

Datos Orientada a Documentos y Neo4j como Base de Datos Orientada a Grafos), diseñar las consultas que se van a ejecutar, así como las bases de datos sobre las que se van a realizar las consultas. También se limpiará el dataset en función de los aspectos anteriores y se ejecutarán las consultas en las bases de datos. Es imprescindible comparar los resultados de las pruebas realizadas para poder tomar una decisión en cuanto al rendimiento.

Por último, finalizado el caso de estudio, es necesario analizar si se han cumplido los objetivos establecidos y decidir los próximos pasos que se podrían dar para hacer un caso de estudio más exhaustivo.

1.2 OBJETIVOS

El objetivo principal de este trabajo es realizar un análisis comparativo del rendimiento de distintas consultas aplicadas a varios tipos de productos de Bases de Datos NoSQL. Para analizar dicho rendimiento se compararán los tiempos de respuesta de los productos.

Con el fin de alcanzar el objetivo principal de este trabajo, se han definido los siguientes objetivos parciales:

- Análisis de la evolución histórica de los sistemas de almacenamiento.
- Estudio de las principales características y productos de las Bases de Datos NoSQL.
- Desarrollo de un caso de estudio para las Bases de Datos NoSQL. Este objetivo parcial se desglosa en varios más pequeños:
 - Elección de un *dataset* y su limpieza para los dos productos elegidos.
 - Diseño de las consultas y de las bases de datos para los dos productos elegidos.
 - Desarrollo del caso de estudio aplicando las consultas sobre el dataset almacenado en las bases de datos diseñadas.
 - Comparativa de los resultados.
- Creación de la memoria.

1.3 MÉTODO DE TRABAJO

Este trabajo está compuesto por tres fases distintas, como se puede ver en la Figura 1. Paralelamente se ha ido elaborando la presente memoria.

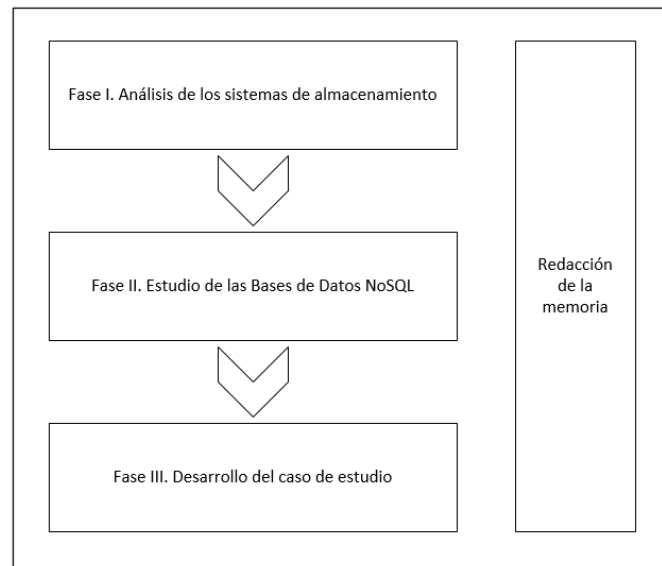


Figura 1. Método de trabajo.

La fase I es una fase de análisis, en la que se estudiará la evolución de los distintos sistemas de almacenamiento que han surgido a lo largo de los años, así como las características de los dos tipos principales que conviven en la actualidad.

La fase II es una fase de estudio centrada en las Bases de Datos NoSQL. Se determinarán las principales características, se compararán los cuatro tipos principales y se enumerarán los productos más populares de cada modelo.

La fase III se corresponde con el caso de estudio. En ella se decidirá el *dataset* sobre el que se va a realizar el análisis del rendimiento de los productos que se escogerán también en esta fase. También se diseñarán las bases de datos y las consultas a aplicar sobre ese *dataset* y se realizará una comparativa de los resultados obtenidos para poder decidir qué producto aporta mayor rendimiento para el *dataset* y el tipo de consultas escogido.

Por último, la elaboración de esta memoria se ha ido haciendo de manera paralela a todas las fases descritas anteriormente.

1.4 MEDIOS HARDWARE Y SOFTWARE UTILIZADOS

Para la realización de este trabajo han sido necesarios varios medios, tanto hardware como software, que se indican a continuación:

- Hardware:
 - Ordenador de sobremesa iMac Retina 5K de 27 pulgadas, con un procesador 4GHz Intel Core i7, memoria RAM de 8GB 1600 MHz DDR3, tarjeta gráfica AMD Radeon R9 M290X 2048 MB y sistema

operativo macOS High Sierra. Este ordenador se ha utilizado para realizar la documentación del trabajo.

- Ordenador portátil Asus GL552VW de 15 pulgadas, con un procesador 2.60GHz Intel Core i7-6700HQ, memoria RAM de 20GB, tarjeta gráfica NVIDIA GeForce GTX960M 4GB GDDR5 VRAM, disco duro 256GB SSD + 1TB 7200rpm SATA y un sistema operativo Windows 10 Home. En este equipo se realiza el caso de estudio, ya que será donde se encuentren instalados los productos MongoDB y Neo4j.
- Software:
 - PyCharm Community Edition 2018.1.4: para la realización de los scripts de código de limpieza y de introducción de consultas en la base de datos MongoDB.
 - MongoDB 3.6.5: base de datos que representa al tipo basado en documentos.
 - Neo4j Community Edition 3.4.0: base de datos que representa al tipo basado en grafos.
 - EmEditor 17.8.1: editor de texto que se emplea para el análisis superficial del fichero donde está contenido el *dataset* original.

1.5 ESTRUCTURA DE LA MEMORIA

La presente memoria se encuentra estructurada en cinco capítulos donde se explican detalladamente todas las acciones que se han llevado a cabo para la realización de este trabajo.

El **primer capítulo** se trata de una introducción sobre el contenido del trabajo, los objetivos propuestos, el método de trabajo seguido por fases de desarrollo y los medios hardware y software empleados.

El **segundo capítulo** contiene un análisis de la evolución histórica de los sistemas de almacenamiento que se han ido utilizando a lo largo de los años, así como una comparación de las características de los dos tipos de almacenamiento más usados en la actualidad, las Bases de Datos Relacionales y las Bases de Datos NoSQL.

El **tercer capítulo** aborda en profundidad las Bases de Datos NoSQL. Se explica la causa de su origen, en qué casos hay que utilizarlas, los distintos tipos que existen junto con su producto más popular y, finalmente, se comparan los tipos para conocer características comunes y las características que los diferencian a unos de otros y que afectan en la decisión a la hora de decidir el tipo de base de datos a usar.

El **cuarto capítulo** hace referencia al caso de estudio que analiza el rendimiento de los productos NoSQL basándose en los tiempos de respuesta a determinadas consulta. Incluye la elección del dataset y los productos a comparar, el diseño de las bases de datos y de las consultas a ejecutar en ellas, la limpieza del dataset para adaptarlo a las bases de datos diseñadas y la ejecución de las consultas. También incluye una comparativa de los tiempos de respuesta obtenidos con la aplicación de las consultas y que nos permiten decidir en cuanto al mejor rendimiento.

El **quinto y último capítulo** se trata de las conclusiones, así como de una reflexión acerca del cumplimiento de los objetivos y de los posibles trabajos futuros para poder mejorar este análisis de rendimiento.

A continuación, figuran las **referencias** que han servido de apoyo para la realización del trabajo.

Por último, se encuentran varios **anexos** en los que figura el código que ha sido necesario para el desarrollo del caso de estudio.

2. ESTADO DEL ARTE

En primer lugar, es necesario ver la evolución que ha tenido el almacenamiento de datos en la historia para entender por qué ha sido necesaria la aparición de las Bases de Datos NoSQL. Además, veremos las propiedades que las diferencian de su principal competidor, las Bases de Datos Relacionales. [2]

2.1 EVOLUCIÓN HISTÓRICA DE LAS BASES DE DATOS

A lo largo de los años se han ido sucediendo distintos métodos para el almacenamiento y la recuperación de datos. En este capítulo vamos a ver desde los Sistemas de Gestión de Datos basados en Ficheros hasta las Bases de Datos NewSQL, pasando por todos los tipos de bases de datos que han aparecido entre ambos.

2.1.1 Sistemas de Gestión de Datos basados en Ficheros

Un fichero es un conjunto de datos organizados, almacenados en un medio de almacenamiento de larga duración. Al depender de un medio físico para el almacenamiento, se producen importantes restricciones debido a las capacidades de este.

Dichos ficheros estarán divididos en bloques dentro de los cuales se almacena la información deseada y a los que se tiene acceso de manera secuencial. En algunos casos se puede tener un acceso directo, pero es más costoso. Un ejemplo de la estructura de un fichero se puede ver en la Figura 2.

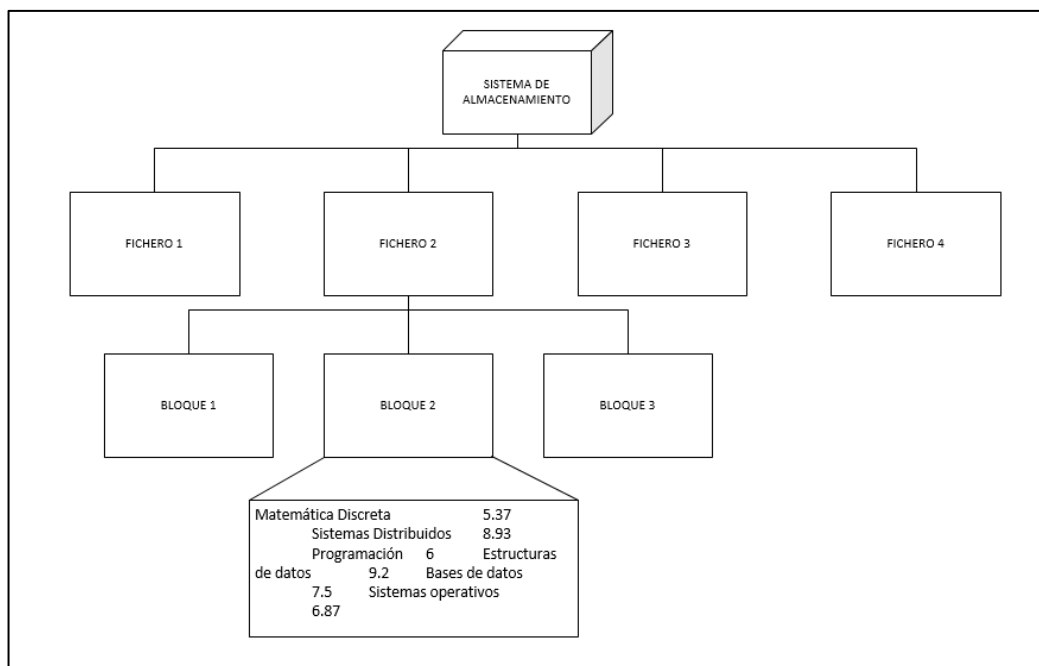


Figura 2. Estructura de un Sistema de Gestión de Ficheros

La organización de los datos dentro de los ficheros depende en gran medida del tipo de programa que se vaya a utilizar.

Este tipo de almacenamiento tiene varios problemas importantes:

- No se puede limitar el acceso a datos concretos del fichero, es decir, no se puede garantizar la confidencialidad de información concreta.
- Pueden existir duplicados de los datos en distintos ficheros originándose una inconsistencia del sistema.
- Es un sistema ineficiente en cuanto a la recuperación de la información.

Estos problemas se abordan con las bases de datos, tal y como veremos en el siguiente apartado.

2.1.2 Bases de Datos

Las bases de datos son una “colección o depósito de datos integrados, almacenados en soporte secundario (no volátil) y con redundancia controlada. Los datos, que han de ser compartidos por diferentes usuarios y aplicaciones, deben mantenerse independientes de ellos, y su definición (estructura de la base de datos) única y almacenada junto con los datos, se ha de apoyar en un modelo de datos, el cual ha de permitir captar las interrelaciones y restricciones existentes en el mundo real. Los procedimientos de actualización y recuperación, comunes y bien determinados, facilitarán la seguridad del conjunto de los datos” [4].

Surgen para solventar los problemas de los Sistemas de Gestión basados en Ficheros y deben garantizar tres aspectos:

- Almacenamiento de datos persistente. Los datos se deben almacenar en memoria persistente, de tal forma que no se pierdan si cae la base de datos. El almacenamiento tendrá lugar en diferentes dispositivos y, normalmente, se utilizarán índices para recuperar la información.
- Consistencia de datos. Es importante asegurar que se escriben y se recuperan los datos correctos. Podría existir un error debido al fallo del *hardware* o por accesos simultáneos a la información, por lo que será necesario escoger un tipo de base de datos que garanticen ese tipo de accesos.
- Disponibilidad de los datos. Los datos deben estar siempre disponibles. Una posible solución para evitar problemas es tener varios servidores, uno primario para actualizaciones y consultas y otro de respaldo o *backup* en caso de fallo (los cambios que se realicen en el primer servidor serán reflejados en el segundo para

que ambos almacenen la misma información en todo momento). Esto provoca que las operaciones de escritura ralenticen el funcionamiento.

A continuación, se van a ver los distintos tipos de bases de datos que han ido apareciendo desde los años 60 hasta la actualidad.

2.1.2.1 Bases de Datos Jerárquicas

Surgen en 1968 de la mano de IBM y su proyecto IMS (*Information Management System*), liderado por Vernon Watts. [2]

En este modelo, los datos se organizan usando una estructura de árbol. Un nodo padre de información puede tener varios nodos hijos que aporten información adicional acerca del nodo padre. El nodo que no tiene padre se conoce como raíz y los nodos que no tienen hijos se conocen como hojas.

Es un sistema adecuado para los datos que se pueden ajustar a este tipo de estructura (relaciones 1:N) pero en el caso de que un nodo hijo pueda tener varios padres (N:M), por ejemplo que un libro tengo más de un autor, se produce una ruptura del esquema causando varios problemas:

- Uso ineficiente del espacio de almacenamiento.
- Posibilidad de datos inconsistentes si no se gestiona bien la redundancia.
- Posibilidad de errores al agregar datos (número total de libros).

Actualmente no se encuentran en uso.

2.1.2.2 Bases de Datos en Red

Este modelo fue estandarizado en 1969 por CODASYL [2], teniendo como producto más relevante IDMS. Al igual que la base de datos anterior, emplea enlaces entre los nodos de información, pero no se limita a un único nodo padre por nodo hijo, por lo que pasamos de una estructura en árbol a una estructura en red, habitualmente denominada grafo.

En este grafo, los arcos son dirigidos para poder representar claramente la relación padre-hijo existente.

Con este tipo de bases de datos encontramos los siguientes problemas:

- Son difíciles de diseñar y mantener.
- Acceder a determinados datos puede ser complejo.
- No es un modelo adecuado para modelos de datos complejos.
- Modificar el esquema puede ser difícil e ineficiente.

2.1.2.3 Bases de Datos Relacionales

Diseñadas por Edgar F. Codd en 1970 [2], aparecen debido a la necesidad de solventar varios problemas:

- Duplicación de datos.
- Falta de seguridad.
- Búsquedas ineficientes.
- Falta de independencia entre la representación lógica y la física.

Estas bases de datos fueron revolucionarias porque permitían acceder a la información sin conocer cómo o dónde se almacenaba permitiendo la independencia de los datos. Han dominado el mundo de las bases de datos durante décadas, adaptando los productos para almacenar datos en diferentes formatos y en 1974 surgió el lenguaje que las acompaña, SQL. Posteriormente, en 1977, aparecería la primera base de datos comercial de tipo relacional, Oracle. Actualmente son las bases de datos más utilizadas.

En muchos casos, este tipo de bases de datos son *Open Source* [10], lo que significa que son productos con licencia libre. Esto hace que sean bases de datos muy económicas, normalmente de reconocido prestigio, que aportan fiabilidad, velocidad, y rendimiento. Son de fácil administración y tienen conexión con otros productos. Son bases de datos bien documentadas, con buenas herramientas y para los que es posible recibir cursos de formación siendo, además, cada vez más usadas en entornos productivos, siendo una alternativa real para las empresas. Los principales proyectos de software libre son MySQL (1995) y PostgreSQL (1994).

A pesar de todo, es posible encontrar en los productos asociados a las Bases de Datos Relacionales algunas de las siguientes limitaciones:

- Fragmentación de los conceptos del mundo real debido a la normalización.
- Pobreza de conceptos para captar la semántica.
- Poco soporte para recoger restricciones (sobre todo en los productos).
- Excesiva homogeneidad horizontal (mismos atributos) y vertical (mismos dominios), exigiendo valores atómicos.
- Limitada cantidad de operaciones disponibles.
- Dificultad para gestionar consultas recursivas.
- Dificultad de cambios en el esquema.

La Orientación a Objetos pretendía ser la solución.

2.1.2.4 Bases de Datos Orientadas a Objetos

Una Base de Datos Orientada a Objetos es aquella en la cual se representa la información en forma de objetos que son utilizados en programación orientada a objetos, por lo que se trabaja con lenguajes de programación de este tipo. Además, estas bases de datos incluyen los conceptos clave del modelo de objetos, que son las siguientes propiedades [9]:

- Encapsulación. Oculta información al resto de objetos para impedir conflictos o un acceso incorrecto.
- Herencia. Jerarquía de clases a partir de la que los objetos heredan comportamientos.
- Polimorfismo. Propiedad de una operación que permite aplicarse a objetos de distinta tipología.

Se dan dos tendencias distintas, una más revolucionaria en el año 1989 que mezclaba características de la orientación a objetos y de los sistemas de gestión de bases de datos teniendo como resultado las Bases de Datos Objeto-Relacionales, y una tendencia más conservadora en el año 1990 con las Bases de Datos Orientadas a Objetos.

Entre los productos con más popularidad podemos encontrar Caché y Db4o.

2.1.2.5 Bases de Datos Multidimensionales

Son bases de datos que se utilizan para aplicaciones de procesamiento analítico en línea (OLAP) y para *data warehouse* [11]. Con frecuencia se crean usando entradas de las bases de datos relacionales existentes. Fueron especialmente populares en los años 1990s y 2000s.

Al tratarse de sistemas orientados al análisis, proporcionan a las compañías un sistema fiable que les permita mejorar las operaciones productivas, tomar decisiones inteligentes y optimizar la competitividad en el mercado.

Un ejemplo de un producto que incluya una Base de Datos Multidimensional es Oracle 18c, aunque la mayoría de los Sistemas de Gestión de Bases de Datos Relacionales importantes tienen una parte de soporte para el desarrollo y la explotación de *data warehouses*.

2.1.2.6 Bases de Datos NoSQL

El acrónimo NoSQL corresponde a la frase “*Not only SQL*” (aunque en los inicios fue “*Not SQL*”). Son bases de datos que surgen debido a las necesidades crecientes causadas por la expansión de aplicaciones de internet.

Comenzaron a originarse a finales de los años noventa y fueron concebidas por distintas empresas y grupos independientes que buscaban soluciones específicas para sus problemas, asumiendo una consistencia eventual a cambio de poder escalar horizontalmente.

Algunas de las razones por las que estas compañías optaron por desarrollar sus propios Sistemas de Gestión de Bases de Datos Distribuidos son las siguientes:

- Se quería conseguir una mayor disponibilidad y rendimiento que hiciesen posibles muchas operaciones concurrentes.
- Se consideró que una gran parte de las características de las Bases de Datos Relacionales no eran necesarias y que el modelo relacional no era el más apropiado para representar datos, ni el lenguaje SQL el más adecuado para consultas simples.

En el capítulo 3 veremos con mayor profundidad este tipo de bases de datos.

2.1.2.7 Bases de Datos NewSQL

El término surge en 2011 debido a la necesidad de escalar en algunas aplicaciones y a la incapacidad de usar las soluciones NoSQL en determinadas ocasiones, porque no pueden renunciar al soporte de transacciones y a la consistencia. [15]

Estas bases de datos se encuentran divididas en nodos, cada uno de los cuales almacena un subconjunto de los datos. A la hora de hacer consultas, éstas se dividen en partes y se envían a los nodos que contienen y procesan los datos, lo que permite escalar linealmente a medida que se agregan más nodos.

Este tipo de bases de datos están pensadas principalmente para aplicaciones que requieren gestionar una gran cantidad de registros, pero con la consistencia que ofrecen las Bases de Datos Relacionales y mediante el lenguaje SQL.

Sus principales características son las siguientes:

- Las transacciones de lectura-escritura de corta duración.
- Afectan a un pequeño subconjunto de datos.

Algunos productos populares de este tipo de base de datos son Google Spanner, Clustrix, VoltDB y SQLFire.

2.2 PROPIEDADES DE LAS BASES DE DATOS

De todos los tipos de bases de datos de los que hemos hablado anteriormente hay dos de gran importancia en la actualidad, las Bases de Datos Relacionales y las Bases de Datos NoSQL, cada una con propiedades distintas.

Las Bases de Datos Relacionales implementan las propiedades ACID, mientras que las Bases de Datos NoSQL implementan las propiedades BASE, tal y como veremos en los siguientes subapartados. [12]

2.2.1 Propiedades ACID

Las bases de datos que poseen las propiedades ACID son las Relacionales. ACID es el acrónimo que se crea con las siguientes características:

- *Atomicity*. Todas las transacciones deben completarse, no puede quedar ninguna inacabada.
- *Consistency*. Las transacciones deben garantizar la integridad de los datos, es decir, que se mantengan exactos y consistentes. De esta manera se garantiza que la información que se presenta al usuario será siempre la misma.
- *Isolation*. Esta propiedad asegura que una operación no puede afectar a otras, ya que las transacciones sobre la misma información se mantienen independientes y aisladas hasta que la transacción se ha completado.
- *Durability*. Los resultados de las transacciones son persistentes y no se pueden deshacer.

A estas propiedades hay que añadir:

- Escalabilidad vertical. Se suele escalar mediante un hardware más potente, lo que puede implicar fuertes inversiones.
- Ante un número muy elevado de peticiones pueden generar cuellos de botella, debido a la consistencia y el aislamiento, ralentizando todo el sistema.

2.2.2 Propiedades BASE

Las bases de datos que poseen las propiedades BASE son las NoSQL. BASE es el acrónimo que se crea con las siguientes características:

- *Basically Available*. Es posible que ocurra un fallo parcial del sistema y que el resto del sistema pueda seguir funcionando.

- *Soft state*. Los datos que llevan mucho tiempo en el mismo estado pueden reescribirse con datos más recientes, lo que provoca la inconsistencia del sistema.
- *Eventually consistent*. La base de datos puede encontrarse en un estado inconsistente temporalmente. Hay varios tipos: [2]
 - *Casual consistency*. Asegura que todas las copias de la base de datos reflejan las operaciones en el mismo orden.
 - *Read-your-writes consistency*. Si un usuario actualiza un dato, todas sus lecturas recuperarán el dato correctamente actualizado.
 - *Session consistency*. Asegura consistencia “*Read-your-writes*” durante una sesión. No se garantiza en otra sesión en el mismo servidor.
 - *Monotonic read consistency*. Una vez recuperado un resultado, nunca podremos recuperar el dato de una versión anterior.
 - *Monotonic write consistency*. Si se realizan varias actualizaciones, se ejecutan en el orden correcto.

A estas propiedades hay que añadir:

- Escalabilidad horizontal. Las Bases de Datos NoSQL están diseñadas para ser escalables a través de múltiples servidores de coste reducido.
- Son capaces de almacenar y manejar grandes volúmenes de datos con alto rendimiento y disponibilidad a costa de pérdida de consistencia y aislamiento.

3. BASES DE DATOS NoSQL

Dentro de las Bases de Datos NoSQL tenemos varios tipos distintos: clave-valor, documentales, de grafos y de familia de columnas. Será necesario ver primero las características generales que se asocian a todos los tipos y luego, de manera más específica, las características particulares de cada una para poder entender en qué situación es conveniente aplicar cada modelo.

3.1 INTRODUCCIÓN

Las Bases de Datos Relacionales han sido las bases de datos dominantes de las últimas décadas, pero, debido a la gran cantidad de datos que manejan las aplicaciones web, se necesitan unos requisitos que son difíciles de cumplir y más costosos usando este tipo de bases de datos:

- Será necesario tener disponible más CPU, más memoria, usar dispositivos más rápidos... pero por mucho que ampliemos, existe un límite.
- Para obtener un rendimiento mayor, hay que desnormalizar las bases de datos, lo que implica más riesgo de anomalías de datos.
- Usar varios servidores, lo cual es complejo en una Base de Datos Relacional.

En caso de no tener tantos recursos, la mejor solución es usar Bases de Datos NoSQL, ya que la gestión de grandes volúmenes de datos requiere varias características que este tipo de bases de datos son capaces de ofrecer:

- Escalabilidad. Es posible aumentar o disminuir el número de servidores sin comprometer el funcionamiento y la calidad de la base de datos. Las Bases de Datos NoSQL están diseñadas para utilizar los servidores disponibles con poca intervención de los administradores.
- Coste. La mayoría de las Bases de Datos NoSQL están disponibles como *Open Source*, existiendo empresas que proporcionan soporte para este tipo de sistemas, lo que implica un abaratamiento de este tipo de bases de datos propiciando su expansión.
- Flexibilidad. Las Bases de Datos NoSQL no requieren una estructura fija, por lo que no es necesario conocer de antemano la forma que va a tener nuestro problema, lo que aporta una gran flexibilidad y capacidad de adaptación en caso de que vayan surgiendo nuevas necesidades.
- Disponibilidad. Las Bases de Datos NoSQL están diseñadas para poder funcionar en múltiples servidores baratos.

Los diseñadores de bases de datos migran a sistemas NoSQL cuando las soluciones relacionales en aspectos como escalabilidad, coste, flexibilidad y disponibilidad no son suficientemente eficientes, dando prioridad a alguna de ellas en función de la aplicación que se vaya a implementar.

Aunque las Bases de Datos NoSQL no tienen por qué estar implementadas en sistemas distribuidos, algunas de sus características más interesantes destacan en una implementación distribuida. Además, tiene sentido implementarlas en múltiples servidores cuando la disponibilidad y la escalabilidad son aspectos básicos porque es necesario un equilibrio entre disponibilidad, consistencia, *partition-protection* y durabilidad, aspectos fundamentales en este tipo de bases de datos.

En la Tabla 1 se puede ver una tabla comparativa de las propiedades en función del tipo de base de datos escogido.

Característica	Bases de Datos NoSQL	Bases de Datos Relacionales
Rendimiento	Alto	Bajo
Fiabilidad	Baja	Alta
Disponibilidad	Alta	Alta
Consistencia	Baja	Alta
Almacenamiento de datos	Optimizadas para una gran cantidad de datos	Adecuadas para una cantidad de datos mediana
Escalabilidad	Alta	Alta (pero más cara)

Tabla 1: Comparativa entre Bases de Datos NoSQL y Bases de Datos Relacionales [14]

Otros aspectos a tener en cuenta a la hora de elegir una Base de Datos NoSQL:

- Tipos de consultas que se realizan.
- Volumen de lecturas y escrituras.
- Tolerancia ante datos inconsistentes en las réplicas.
- La naturaleza de las relaciones entre entidades.
- Disponibilidad y requisitos de recuperación ante pérdidas.
- Flexibilidad en los modelos de datos.
- Requisitos de latencia.

Al contrario que ocurrió con las Bases de Datos Relacionales, que desplazaron a sus antecesores, las Bases de Datos NoSQL coexisten entre sí y con las Bases de Datos Relacionales por las diferentes necesidades a las que hay que dar respuesta.

3.2 CATEGORIZACIÓN DE LAS BASES DE DATOS NOSQL

Dentro de las Bases de Datos NoSQL podemos encontrar cuatro tipos distintos en función del tipo de esquema que emplean para guardar los datos:

- Bases de Datos Clave-Valor.
- Bases de Datos Orientadas a Documentos
- Bases de Datos Orientadas a Grafos
- Bases de Datos Familia de Columnas

A continuación, se va a proceder a explicar cada una de ellas, así como el producto más popular de cada una. [1]

3.2.1 Bases de Datos Clave-Valor

Las Bases de Datos Clave-Valor son el tipo más sencillo dentro de las Bases de Datos NoSQL. Permiten obtener y actualizar datos en base a una clave única que lleva un valor asociado, como se puede ver en la Figura 3.

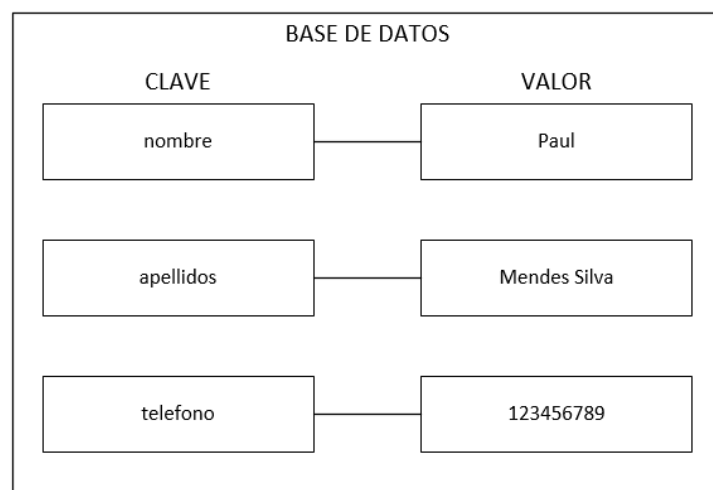


Figura 3. Esquema de una Base de Datos Clave-Valor

Estas bases de datos son adecuadas cuando la facilidad de almacenamiento y recuperación son más importantes que la organización de los datos en estructuras más complejas, como tablas o redes.

La clave es el elemento principal de este tipo de base de datos. Debe ser única en el espacio de nombres y, en caso de tener un único espacio de nombres, la clave será única en ella. Sin embargo, en caso de tener una base de datos con varios espacios de nombre, como se puede

ver en la Figura 4, esta clave podrá aparecer en varios, siempre que no haya repeticiones dentro del mismo.

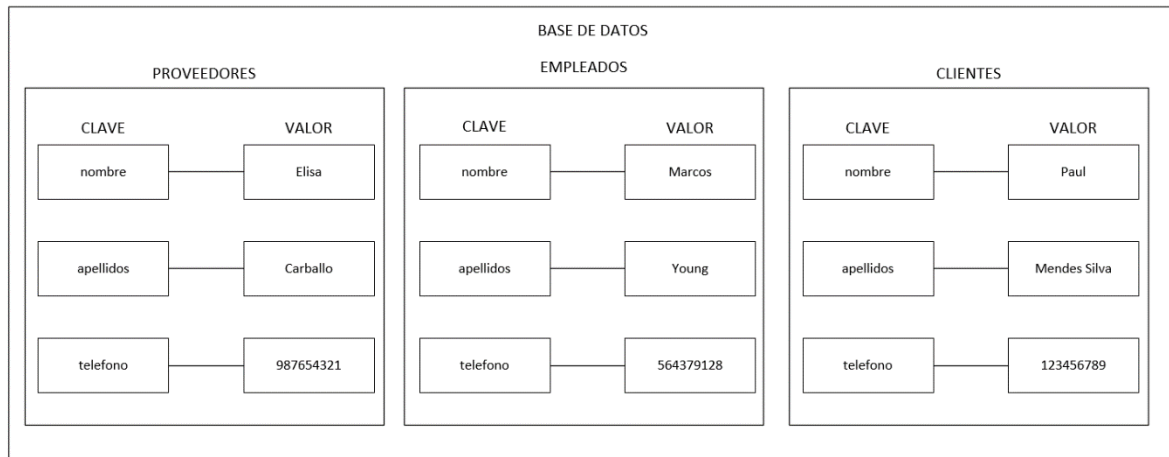


Figura 4. Esquema de una Base de Datos Clave-Valor con varios espacios de nombre.

En caso de querer repetir la clave para varias entidades dentro de un mismo espacio de nombres, es posible añadirle un prefijo que referencie cada entidad.

El valor asociado a la clave es el dato que queremos almacenar y que puede ser de cualquier tipo: texto, número, código de marcado, código de programación, una imagen... y de cualquier tamaño. Debido a la falta de comprobación del tipo, será necesario que los desarrolladores implementen las comprobaciones en sus aplicaciones.

Según DB-ENGINES [3] y su ranking de base de datos, la Base de Datos Clave-Valor con mayor popularidad es Redis [4,21,22,23].

El lanzamiento de Redis se realizó en 2009 y tiene las siguientes características:

- Almacena todos los datos en memoria, por lo que las operaciones de lectura y escritura son muy rápidas, pero esto implica que las estructuras tienen como límite el tamaño de la memoria.
- Es “*single threaded*”, es decir, siempre ejecuta únicamente un comando, siendo éstos atómicos.
- Permite trabajar con múltiples bases de datos identificadas por un número secuencial.
- Está diseñada para ser escalable y proporciona soporte para tipos de datos avanzados.
- Cuenta con una espectacular velocidad, sencillez de uso y flexibilidad.

3.2.2 Bases de Datos Orientadas a Documentos

Estas bases de datos emplean un sistema clave-valor distinto al anterior ya que el valor no será directamente la información que se quiere almacenar, sino un elemento denominado documento dentro del cual estará dicha información. Seguirá existiendo, al igual que con el tipo anterior, una clave única para acceder a cada documento contenido en la base de datos.

El documento almacena la información utilizando una estructura simple, normalmente JSON o XML y se utiliza una clave única por cada registro. Permite hacer consultas no sólo a nivel general, sino también internas al documento.

Estos documentos no tienen por qué tener todos las mismas secciones, atributos o claves, como se puede ver en la Figura 5. Además, pueden existir documentos embebidos. Todo esto aporta una gran flexibilidad a este tipo de bases de datos, pero también posibles problemas a la hora de hacer consultas, por lo que será necesaria una buena gestión.



Figura 5. Ejemplos de documentos con distinta estructura.

La base de datos puede estar formada por varias colecciones, que están a su vez compuestas por un conjunto de documentos, como se puede ver en la Figura 6. Dentro de cada colección los documentos deberían compartir alguna estructura común y ser del mismo tipo de entidad.

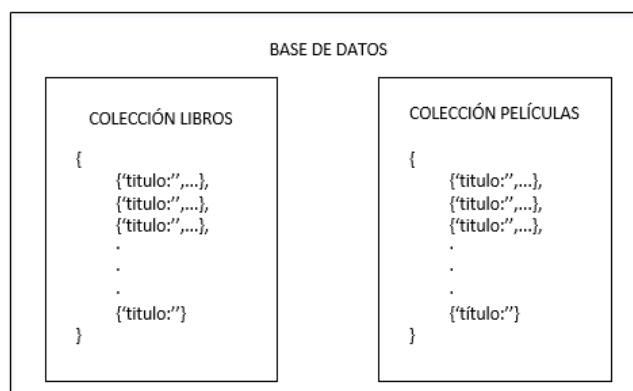


Figura 6. Base de Datos Orientada a Documentos con distintas colecciones.

Según DB-ENGINES [3] y su ranking de base de datos, la Base de Datos Documental con mayor popularidad es MongoDB [5,19,20].

Este producto se lanzó en 2009 y es una de las Bases de Datos NoSQL más populares en la actualidad. Cuenta con las siguientes características:

- Su implementación es fácil y rápida.
- Es ideal para aplicaciones que necesiten almacenar datos semiestructurados y para entornos que requieran escalabilidad horizontal.
- Permite consultas por múltiples atributos y aplicar funciones sobre la base de datos.
- Cuenta con la posibilidad de indexación y de anidamiento de documentos.

3.2.3 Bases de Datos Orientadas a Grafos

Este tipo de bases de datos se modelan mediante un sistema de grafos formados por nodos y relaciones entre ellos, dejando de lado el sistema de fila y columna empleado por los tipos anteriores. Son la opción más adecuada cuando los datos están muy relacionados y conectados entre sí.

Se basan en la teoría de grafos, por lo que es posible recorrer la base de datos accediendo a los nodos y recorriendo las relaciones establecidas entre los mismos. Se puede ver un ejemplo de una estructura de grafo en la Figura 7.

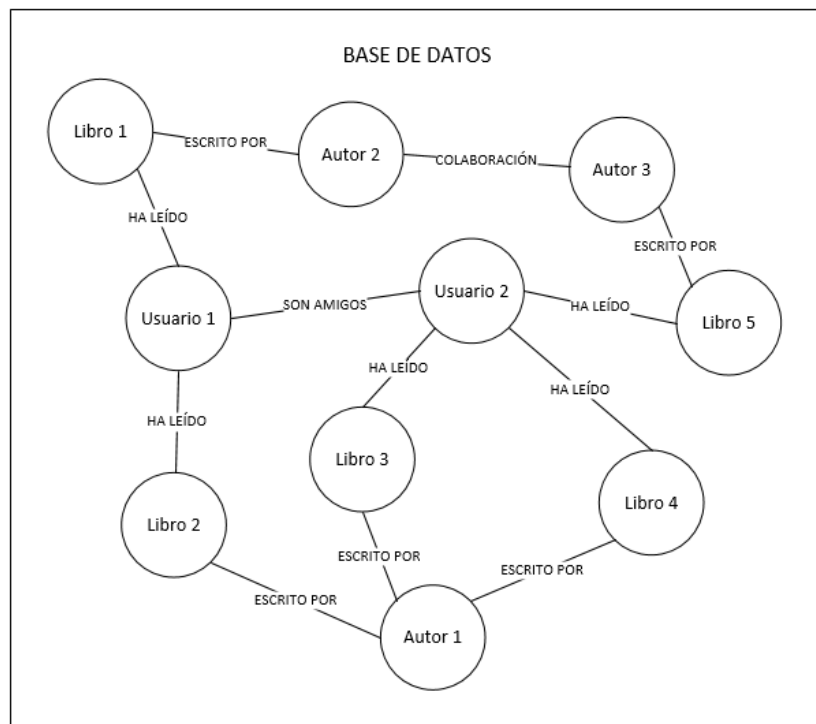


Figura 7. Esquema de una Base de Datos Orientada a Grafos.

Los grafos están formados por dos tipos de elementos, los nodos o vértices, que pueden almacenar atributos o tipos complejos, y las aristas que los unen y que representan las relaciones que existen entre ellos, pudiendo tener asociada una propiedad que explica el tipo de relación.

Además, las relaciones entre esos nodos pueden ser dirigidas o no, es decir, pueden o no ser simétricas. Por ejemplo, como se ve en la Figura 7 de la página anterior, las relaciones “colaboración” y “son amigos”, que hacen referencia a dos autores que han participado en el mismo libro y a dos usuarios que se siguen, son no dirigidas, ya que son relaciones que son iguales en ambos sentidos. Sin embargo, la relación “escrito por”, que indica que un usuario ha leído un libro es dirigida, porque un libro no puede leer a un usuario.

Según DB-ENGINES [3] y su ranking de base de datos, la Base de Datos Orientada a Grafos con mayor popularidad es Neo4j [7,18].

El lanzamiento de Neo4j se produjo en 2007 y se trata de un proyecto *open source* con una versión gratuita y una versión comercial. Este producto tiene las siguientes características:

- Su arquitectura está diseñada para optimizar la gestión, el almacenamiento y el recorrido de nodos y relaciones.
- Puede almacenar un volumen muy elevado de nodos.
- Cuenta con una gran flexibilidad y escalabilidad, es posible añadir nodos y relaciones a un grafo ya existente sin dañar la estructura.
- Es capaz de recorrer 4 millones de nodos por segundo.
- Tiene un lenguaje propio, Cypher.

3.2.4 Bases de Datos de Familia de Columnas

Las Bases de Datos de Familia de Columnas se parecen mucho a las Bases de Datos Relacionales, pero, en este caso, las columnas estarán formadas por dos elementos, un nombre y un valor.

Las columnas hacen referencia a los distintos atributos que almacena cada elemento que se introduce en la base de datos. Cada columna cuenta con el nombre de dicho atributo y un valor asociado. Todas las filas no tienen por qué tener el mismo número de columnas o, si lo tienen, no tienen por qué tener los mismos nombres. Esto se puede ver en la Figura 8.

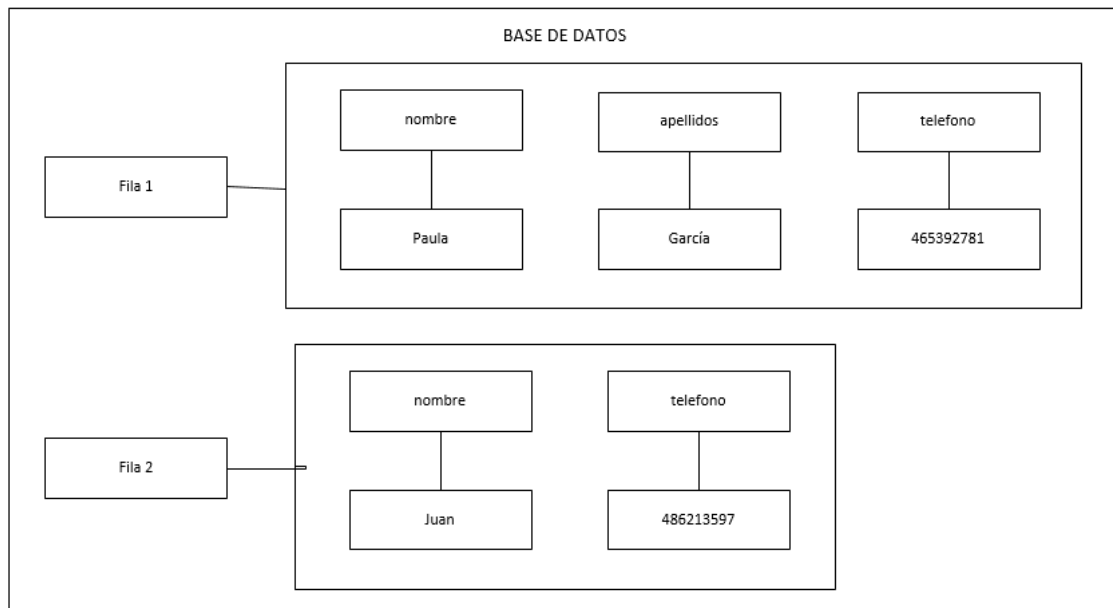


Figura 8. Esquema de una Base de Datos de Familia de Columnas

Si en la base de datos existen varias columnas relacionadas o que se usan normalmente juntas, se conocen con el nombre de familia de columnas.

Según DB-ENGINES [3] y su ranking de base de datos, la Base de Datos de Familia de Columnas con mayor popularidad es Cassandra [6,16,17].

Inicialmente desarrollada por Facebook en 2008, fue liberada como proyecto *Open Source* y convertida en un producto de alto nivel por Apache.

Es una mezcla entre Base de Datos de Familia de Columnas y Base de Datos Clave-Valor y tiene las siguientes características:

- Almacena grandes volúmenes de datos de forma distribuida.
- La comunicación entre los nodos se realiza mediante un protocolo P2P (peer-to-peer).
- La información se encuentra replicada en los nodos, lo que permite operaciones de baja latencia.
- Ofrece soporte para multi data center.
- Los datos se encuentran desnormalizados.
- Es masivamente escalable linealmente.
- Tiene una gran tolerancia a particiones y alta disponibilidad, pero a cambio, es eventualmente consistente.
- Emplea un lenguaje reducido de SQL, CQL (Cassandra *Query Language*).

3.3 COMPARATIVA DE LOS TIPOS DE BASES DE DATOS NoSQL

En la Tabla 2, es posible ver una comparativa de los cuatro tipos de Bases de Datos NoSQL. Las características para las que se realiza esta comparativa son generales, aunque por supuesto algunas dependen del producto NoSQL que se utilice. Además, en algunos casos, aunque todos los tipos cumplan una determinada característica, hay alguno en concreto que la proporciona en mayor medida, por lo que se destaca ese tipo.

CARACTERÍSTICAS	CLAVE-VALOR	DOCUMENTOS	GRAFOS	FAMILIA DE COLUMNAS
LECTURAS Y ESCRITURAS SENCILLAS	X			X
ESCRITURAS MASIVAS		X	X	X
ALTA DISPONIBILIDAD				X
MECANISMOS DE REPLICACIÓN		X		X
MÚLTIPLES SERVIDORES		X		X
SOPORTA UN VOLUMEN DE DATOS ELEVADO	X	X	X	X
MODELO DE DATOS SENCILLO	X			
GRAN FLEXIBILIDAD	X	X	X	X
CONSULTAS COMPLEJAS		X		
DATOS MUY RELACIONADOS			X	
DATOS COMPLEJOS	X	X	X	X
INDEXACIÓN		X	X	X
ANIDAMIENTO		X		
RENDIMIENTO CONSTANTE (el volumen de la base de datos no afecta al rendimiento de las consultas)		X	X	
INTEGRIDAD DE LOS DATOS	NO	NO	NO	NO
ORDENACIÓN DE RESULTADOS		X	X	X
ESCALABILIDAD	X	X	X	X
PRODUCTOS	Redis, Riak, SimpleDB, DynamoDB, Oracle BerkeleyDB, ...	MongoDB, CouchDB, CouchBase, RethinkDB, Cloudant	Neo4J, Titan	Cassandra, Hbase, BigTable...
LENGUAJE	Lenguaje del producto	Lenguaje del producto	Cypher	Cassandra Query Language
USOS	Logging, atributos temporales en aplicaciones web (carrito), almacenamiento de grandes objetos como imágenes y audio, caché de una Base de Datos Relacional para mejorar el rendimiento.	Gestión documental, sitios web con alta carga de lecturas y escrituras.	Redes sociales, motores de recomendaciones, aplicaciones geoespaciales...	Estadísticas en tiempo real, aplicaciones distribuidas geográficamente en múltiples centros de datos, aplicaciones que toleran cierta inconsistencia en las réplicas.

Tabla 2. Comparativa de los tipos de Bases de Datos NoSQL.

4. CASO DE ESTUDIO

En este capítulo del trabajo se va a realizar un análisis comparativo del rendimiento de varias consultas sobre varios productos de Bases de Datos NoSQL. Para ello, se van a llevar a cabo las siguientes acciones:

- Escoger el *dataset*.
- Decidir qué productos NoSQL se van a comparar.
- Determinar el aspecto que se va a analizar.
- Diseñar las consultas que se van a ejecutar.
- Diseñar las bases de datos donde se van a guardar los datos.
- Limpiar el *dataset* para los dos productos escogidos.
- Ejecutar las consultas.
- Comparar los resultados obtenidos.

En los siguientes subapartados se explicará cada acción en detalle para intentar llegar a una conclusión sobre el producto óptimo en cuanto al aspecto a analizar.

4.1 CAPTURA DE LOS DATOS

El dataset escogido en la base de datos DBLP *Computer Science Bibliography*, considerada como la mayor recopilación de referencias bibliográficas académicas específicamente centrada en la informática. En particular, almacena los datos relativos a la gran mayoría de las revistas científicas y congresos académicos sobre informática.

Estos datos se descargan como un único fichero XML [24], que se encuentra comprimido y tiene un tamaño de 445 MB, ocupando expandido 2.33 GB.

El *dataset* está formado por 8 tipos de elementos: *article*, *inproceedings*, *incollection*, *proceedings*, *book*, *phdthesis*, *mastersthesis* y *www*. Para cada documento almacenado, aparte de su tipo, se tienen distintos atributos como el título, la fecha, los autores que lo han elaborado, el número de páginas, la *url*, etc. [25]

Una vez escogido y descargado el *dataset*, es necesario revisar el fichero para estudiar la estructura y poder escoger los productos NoSQL adecuados. Debido a su tamaño, hay algunos programas (Sublime Text, Notepad ++, Pycharm o Visual Code) que han dado problemas a la hora de abrirlo y visualizar los datos. Finalmente se ha escogido EmEditor [26] para ello. Se puede ver una captura de estos datos en la Figura 9.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE dblp SYSTEM "dblp.dtd">
<dblp>
  <article mdate="2017-05-28" key="journals/acta/Saxena96">
    <author>Sanjeev Saxena</author>
    <title>Parallel Integer Sorting and Simulation Amongst CRCW Models.</title>
    <pages>607-619</pages>
    <year>1996</year>
    <volume>33</volume>
    <journal>Acta Inf.</journal>
    <number>7</number>
    <url>db/journals/acta/acta33.html#Saxena96</url>
    <ee>https://doi.org/10.1007/BF03036466</ee>
  </article>
  <article mdate="2017-05-28" key="journals/acta/Simon83">
    <author>Hans Ulrich Simon</author>
    <title>Pattern Matching in Trees and Nets.</title>
    <pages>227-248</pages>
    <year>1983</year>
    <volume>20</volume>
    <journal>Acta Inf.</journal>
    <url>db/journals/acta/acta20.html#Simon83</url>
    <ee>https://doi.org/10.1007/BF01257084</ee>
  </article>
  .
  .
  .
</dblp>

```

Figura 9. Fragmento de datos del fichero “dblp.xml”

4.2 PRODUCTOS NoSQL ESCOGIDOS

Como se vio en el capítulo 3, hay cuatro tipos de Bases de Datos NoSQL: clave-valor, grafos, documentales y de familia de columnas. Tras analizar el dataset escogido se toma la decisión de realizar el caso de estudio únicamente con una Base de Datos Orientada a Documentos y una Base de Datos Orientada a Grafos.

Se descartan las Base de Datos Clave-Valor porque son más adecuadas para casos de *logging*, atributos temporales en aplicaciones web (carrito) o como caché de una Base de Datos Relacional.

Se descartan las Bases de Datos de Familia de Columnas ya que suelen ser empleadas para aplicaciones distribuidas geográficamente en *data warehouses*, algo que no ocurre en este caso.

Además, comparados con los otros dos tipos de bases de datos, las Bases de Datos Clave-Valor y las Bases de Datos de Familia de Columnas llevan asociadas unos tiempos de respuesta elevados a la hora de realizar consultas. Como el estudio comparativo que se va a llevar a cabo consiste en ejecutar distintas consultas en los productos para la comparación del rendimiento, se decide descartar estos dos tipos de bases de datos.

Una vez escogidos los modelos de bases de datos que se van a usar, es necesario seleccionar un producto de cada uno. Se ha optado por “MongoDB” y por “Neo4j”. Ambos son los productos más populares de cada tipo respectivamente.

Tras la selección de productos, hay que proceder a su instalación en el equipo. Esto se explica en los siguientes subapartados.

4.2.1 Instalación de MongoDB

Para la instalación de MongoDB hay que seguir los siguientes pasos:

1. Descargar el instalador de MongoDB. En este caso, la versión es *MongoDB Community Edition 3.6*. En la página web oficial es posible encontrar una guía de instalación. [27,28]
2. Lanzar la instalación y dar permisos para realizar cambios en el equipo. La instalación se realiza en la ruta:
C:\Program Files\MongoDB\Server\3.6\bin
3. Crear un directorio de datos:
C:\data\db
4. Es necesario tener *Python 2.7* instalado en el equipo. [29]. La instalación se realiza en la ruta:
C:\Python27

4.2.2 Instalación de Neo4j

Para la instalación de Neo4j hay que seguir los siguientes pasos:

1. Descargar el instalador de Neo4j en el equipo. En este caso, la versión descargada es *Neo4j Community Edition 3.4*. En la página web oficial es posible encontrar una guía de instalación. [30, 31]
2. Lanzar el instalador y dar permisos para realizar cambios en el equipo. En este caso, la instalación se realiza en la ruta:
D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10

4.3 ASPECTO A COMPARAR

Para poder analizar el rendimiento de las consultas en los dos productos seleccionados con anterioridad, es necesario escoger un aspecto a comparar. En este caso, será el tiempo de respuesta, es decir, el tiempo que tarda la base de datos en proporcionar el resultado una vez ejecutada la consulta.

En Neo4j, conocer el tiempo de respuesta es muy sencillo, ya que te lo proporciona directamente junto con el resultado de la consulta.

Para MongoDB, obtener el tiempo de respuesta depende del tipo de consulta que se realice.

- Para consultas sencillas que emplean la función *find*, es posible conocer el tiempo de respuesta añadiendo al final de la consulta “*.explain(“executionStats”)*”. El resultado que ofrece esta función es una explicación de la ejecución de la consulta. Entre la información que proporciona se encuentra el atributo *executiontimeMillis*, que indica el tiempo total en milisegundos que tarda la consulta en ejecutarse. [32,33]
- En el caso de consultas que empleen *aggregation framework* obtener el tiempo de respuesta es más complicado, ya que la función *explain* no funciona. Para poder medir el tiempo se ha decidido realizar un script en Python que cogerá el tiempo previo a la consulta y el tiempo posterior y los restará. El código se puede ver en la Figura 10. [35,36]

```
def answer_time(bef_time,aft_time):  
  
    bf_obj = datetime.strptime(bef_time, '%Y-%m-%d %H:%M:%S.%f')  
    bf_milli = bf_obj.timestamp() * 1000  
  
    af_obj = datetime.strptime(aft_time, '%Y-%m-%d %H:%M:%S.%f')  
    af_milli = af_obj.timestamp() * 1000  
  
    exec_time = af_milli - bf_milli  
  
    return exec_time
```

Figura 10. Código para calcular los tiempos de respuesta en MongoDB

4.4 DISEÑO DE LAS CONSULTAS

A la hora de analizar el rendimiento de un producto mediante el tiempo, es adecuado realizar consultas de distinta complejidad y que involucren desde elementos concretos de la base de datos hasta el *dataset* completo que esta contiene.

La consulta más básica consiste en obtener todos los elementos de una base de datos. A partir de ahí, la manera de ir aportando complejidad a las consultas es ir concretando el tipo de elemento que se quiere recuperar. Para obtener elementos concretos de una base de datos es necesario aplicar filtros y realizar agrupaciones, lo que da complejidad a las consultas.

Los tipos de consultas que se van a diseñar son las siguientes:

- Recorrer toda la base de datos, también llamado *table scan*.
 - Ejemplo: *Mostrar todas las publicaciones de la base de datos.*

- Filtrado de la base de datos por una condición → operador *where*.
 - Ejemplo: Mostrar todos los artículos en libros.
- Filtrado de la base de datos por una condición controlada con operadores lógicos.
 - Ejemplo: *Publicaciones entre los años 1990 y 2000*.
- Aplicar una función de agregación con un filtrado.
 - Ejemplo: *Número de artículos en la base de datos*.
- Realizar una agrupación por un determinado campo y una ordenación para dicho campo → operador *group by*.
 - Ejemplo: *Publicaciones de un autor concreto por año*.
- Realizar una agrupación aplicando una condición sobre un grupo → operador *having*.
 - Ejemplo: *Número de documentos con más de 3 autores*.
- Simulación del operador *join*, realizando una consulta de un elemento a otro de su mismo tipo.
 - Ejemplo: *Coautores de un autor concreto*.
- Comprobación de la no pertenencia de un elemento a un conjunto → operador *not in*.
 - Ejemplo: *Autores que sólo han publicado un único tipo de documento*.

Los ejemplos que se han puesto para cada tipo son las consultas que se van a ejecutar en las bases de datos que se van a diseñar en el siguiente apartado para calcular el tiempo de respuesta y medir el rendimiento.

4.5 DISEÑO DE LAS BASES DE DATOS

A la hora de realizar el diseño de la base de datos, es necesario buscar la decisión más eficiente para el dataset escogido. En los siguientes subapartados se diseñarán las bases de datos asociadas a cada producto.

4.5.1 Diseño de la base de datos para MongoDB

Para diseñar una Base de Datos Orientada a Documentos, la principal decisión que hay que tomar es el número de colecciones en las que va a estar dividida la base de datos. Para este *dataset* existen dos posibles opciones.

La primera es tener una única colección con todos los documentos juntos, obteniendo una estructura similar a la del fichero de datos original. Un ejemplo de cómo quedaría la colección se puede ver en la Figura 11.

```
{
  { "_id":<ObjectId1>,
    "title": "",
    "year": 0000,
    "pages": "123-456",
    "author": ["Eva Manso", "Daniel Vaquero"],
    "url": "www.miurl.com",
    .
    .
    .
  },
  {doc2,...},
  {doc3,...},
  .
  .
  .
  {docn,...}
}
```

Figura 11. Esquema de la base de datos con una colección.

La segunda opción es tener dos colecciones, una que guarde documentos con la información de los artículos y otra que guarde la información de los autores. Para este caso, ambas colecciones deben estar relacionadas, para poder dar respuesta a consultas que pidan aspectos de ambos tipos de documentos. En la Figura 12 se ve un ejemplo del resultado de esta opción.

<pre>{ { "_id":<ObjectId1>, "title": "", "year": 0000, "pages": "123-456", "author": [<ObjectId17>, <ObjectId23>] "url": "www.miurl.com", . . . }, {pub2,...}, {pub3,...}, . . . {pubn,...} }</pre>	<pre>{ { "_id":<ObjectId17>, "name": "Eva Manso", }, { "_id":<ObjectId23>, "name": "Daniel Vaquero", }, {aut3,...}, . . . {autn,...} }</pre>
---	--

Figura 12. Esquema de la base de datos con dos colecciones.

Para tomar esta decisión, lo primero que hay que analizar es el tipo de consultas que se van a realizar, es decir, hay que saber cómo se van a usar los datos. Observando las consultas diseñadas en el punto anterior, se ve que la mayoría mezclan documentos y autores. Como no existen los *joins* en este tipo de base de datos, para hacer consultas con datos relacionados en las dos colecciones sería necesario hacer más de una consulta, lo que reduciría la eficiencia y

el rendimiento. Además, en el *dataset*, los autores son un atributo de las publicaciones, no tienen ningún atributo propio, por lo que tener una colección de documentos que contienen una única entrada con el nombre del autor tampoco resulta eficiente.

Tras escoger el número de colecciones, hay que ver si es necesario definir índices. En algunos casos, los índices pueden mejorar el rendimiento de las búsquedas mientras que, en otros, no suponen ninguna diferencia. Para este caso, se va a comprobar si los índices mejoran los tiempos de respuesta, ejecutando las consultas con y sin los índices definidos.

Por lo tanto, para MongoDB, se diseñará una base de datos con una única colección que almacene todas las publicaciones del *dataset*. El esquema será el de la figura 11, vista anteriormente.

4.5.2 Diseño de la base de datos para Neo4j

Para diseñar una Base de Datos Orientada a Grafos, la decisión que hay que tomar es respecto al tipo de nodos que van a formar el grafo, así como las relaciones que se establecen entre ellos. Tras el análisis del *dataset*, se detectan dos posibilidades.

La primera consiste en tener un único tipo de nodo. Este tipo de nodo almacenaría toda la información de la publicación, teniendo como resultado un grafo con un número de nodos igual al número de publicaciones de las que está formado el *dataset*. Los nodos que tuviesen en su atributo autor elementos comunes estarían relacionados.

La segunda posibilidad implica la existencia de dos tipos de nodos, uno para los autores y otro para las publicaciones. El nodo de tipo autor almacenará nombres de autores y el nodo de tipo publicación almacenará distintos atributos como son el título o la fecha.

En la Figura 13 se pueden ver los nodos para las dos opciones.

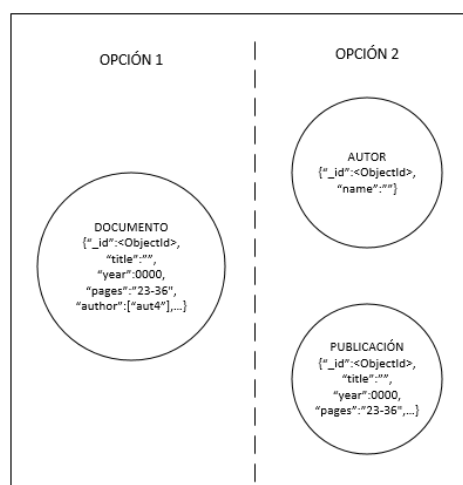


Figura 13. Posibles nodos para la base de datos de Neo4j.

Para tomar la decisión más adecuada, es necesario revisar las consultas diseñadas en el apartado 4.4. En la consulta 5, podemos observar el término coautor, que hace referencia a los autores que han colaborado en una publicación. La mejor manera de representar esta relación de colaboración entre autores es con la segunda opción, es decir, que haya dos tipos de nodos. Habiendo dos tipos de nodos distintos, queda claro que, si existe una relación entre dos nodos autor es porque han colaborado. Si solo hay un tipo de nodo que guarda toda la información y se encuentran relacionados aquellos nodos que tienen autores comunes, no es posible saber realmente cuáles de esos autores que figuran dentro de los nodos son los que han colaborado.

Además, con las Bases de Datos Orientadas a Grafos, lo que más importa son las relaciones entre los nodos, no el contenido, y la mejor manera de representar las relaciones que figuran en las consultas es teniendo dos tipos de nodos.

Por último, serán más eficientes las búsquedas en el grafo ya que, accediendo a los nodos autores, tendremos directamente los documentos que han creado y con qué otros autores han colaborado y, accediendo a los nodos publicaciones, tendremos directamente los autores de esos documentos. Si tuviésemos únicamente un tipo de nodo, habría que recorrer todo el grafo e ir comprobando los atributos de cada uno para obtener la misma información. Esto reduciría en gran medida el rendimiento de la base de datos.

En las Figuras 14 y 15 se pueden ver los esquemas completos de ambas opciones. A simple vista, es más fácil conocer los autores comunes entre dos nodos en el segundo esquema que en el primero. Además, quedan más claras las posibles relaciones entre los distintos nodos.

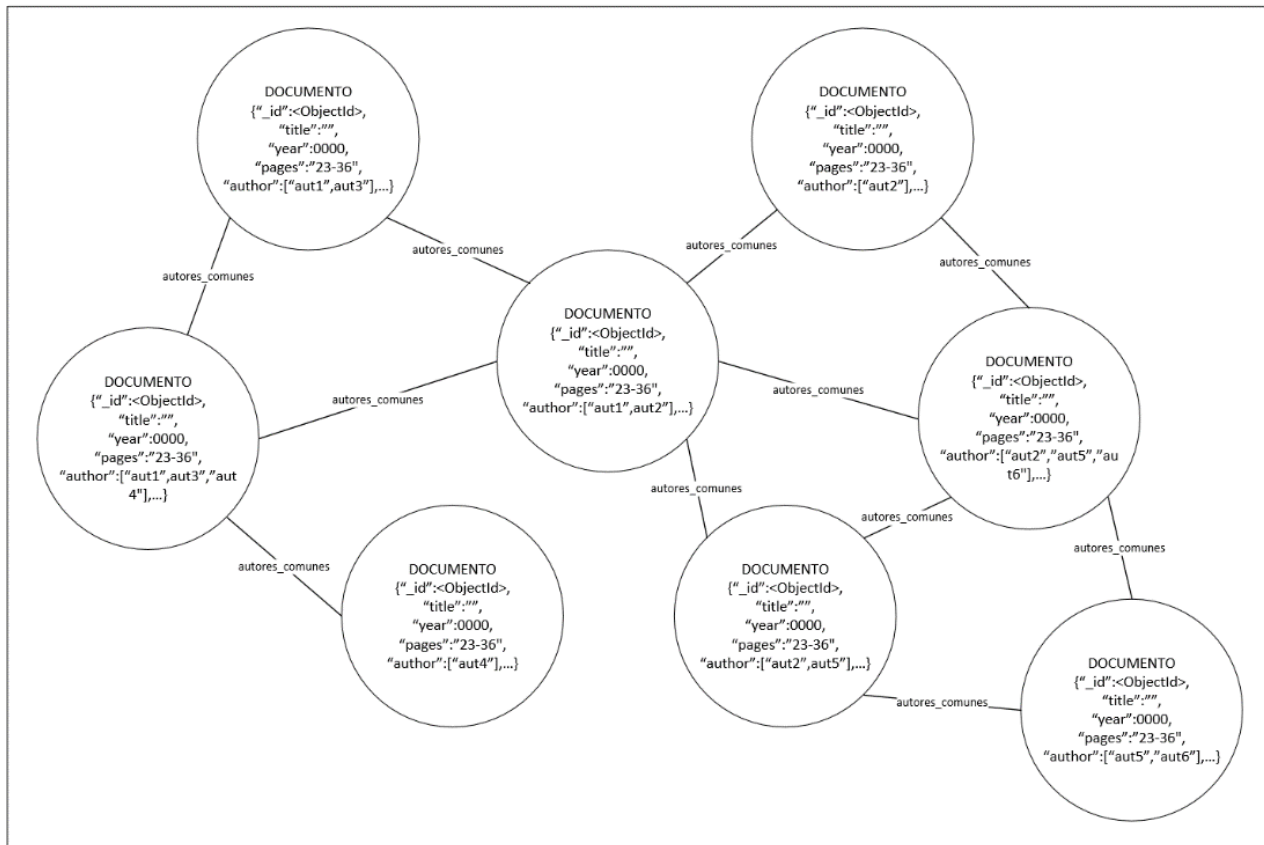


Figura 14. Esquema de la base de datos Neo4j con un tipo de nodo.

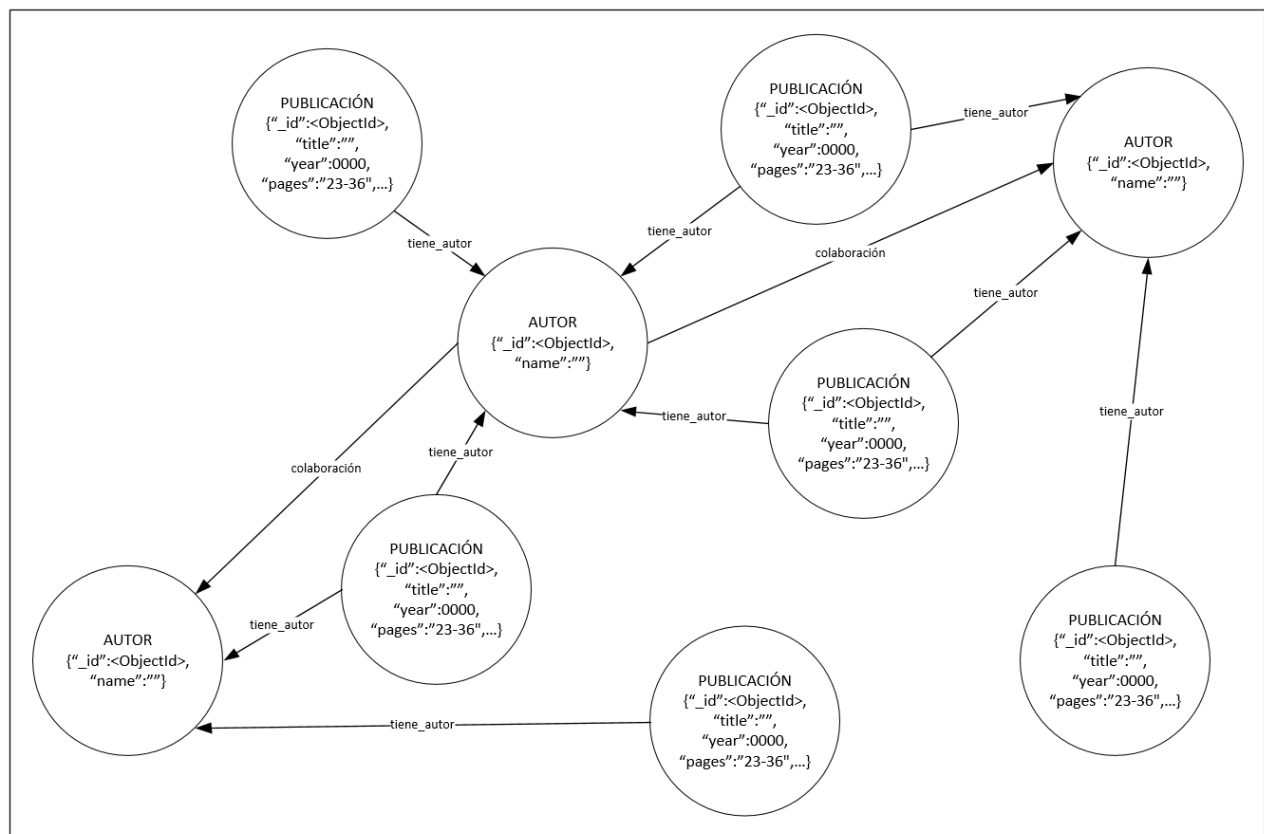


Figura 15. Esquema de la base de datos Neo4j con dos tipos de nodos.

Sabiendo ya el número de nodos que va a tener el grafo, es necesario definir el contenido. En los nodos “autor”, la única información que se almacenará será el nombre del autor, ya que no existen más datos de los autores. En los nodos “publicación” se almacenará el resto de los atributos que tienen las publicaciones en el *dataset*.

Una vez decididos los tipos de nodos que van a formar parte de la base de datos y la información que van a contener, hay que definir las relaciones entre ellos. En un primer momento, como se ha podido ver en la Figura 15, se habían establecido dos tipos de relaciones. La primera relación era la de colaboración entre autores, para poder dar respuesta al término coautor. La segunda relación que iba a existir era la de “tiene_autor” entre los nodos “publicación” y los nodos “autor” para poder obtener los autores de una publicación y todas las publicaciones de un autor. Ambas relaciones iban a ser dirigidas y unidireccionales, ya que Neo4j permite que en las relaciones unidireccionales los dos nodos conectados sean accesibles en ambos sentidos. Sin embargo, esta estructura daba lugar a una gran cantidad de relaciones entre los nodos, ya que, si tienes las publicaciones conectadas a los autores, ya se ve cuáles son los autores que han colaborado en un documento, no es necesario crear relaciones entre ellos, es decir, la información que proporcionaban las dos relaciones era redundante. Esto afectaba al rendimiento a la hora de realizar las consultas, llegando incluso a colapsar la base de datos con prácticamente todas las consultas. Como consecuencia, ha sido necesario buscar otro tipo de relaciones entre los nodos.

Finalmente, se decidió que era necesaria una única relación y que se iba a establecer entre los nodos autor y documento, siendo el origen la publicación y el destino el autor. Esto es así porque las publicaciones son los elementos principales de la base de datos y, a partir de ellos, podemos sacar todos los autores que han colaborado en un documento. Además, debido a la característica de Neo4j mencionada anteriormente, esta relación también permite obtener los documentos en los que ha colaborado un autor. Esta relación no contendrá ningún tipo de atributo, ya que estos se almacenan en los nodos y contará con la etiqueta “tiene_autor”.

Por lo tanto, la base de datos de Neo4j contará con dos tipos de nodos, “autor” y “publicación”, y con un tipo de relación entre ellos, “tiene_autor”, siendo esta relación dirigida y unidireccional. Para esta base de datos también se realizarán pruebas con y sin índices para comparar los tiempos de respuesta. El esquema resultante se puede ver en la Figura 16.

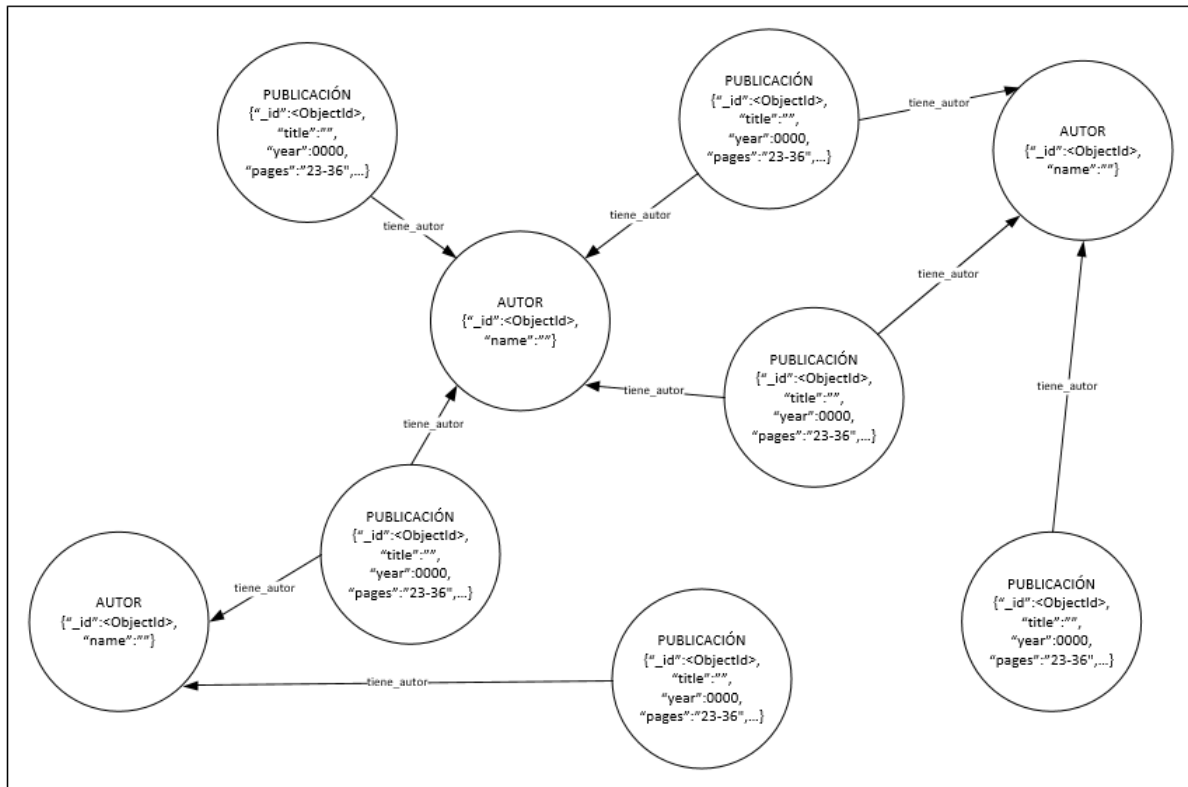


Figura 16. Esquema de la base de datos de Neo4j

4.6 LIMPIEZA DEL DATASET

Debido al tamaño del *dataset*, se ha tomado la decisión de reducir los elementos que contiene. Para ello, se limitan los tipos de documentos a tres: *article*, *inproceedings* e *incollection*, siendo respectivamente artículos de revista, artículos en congresos y artículos en libros. También se va a reducir el número de atributos a tres: *author*, *title* y *year*. Por último, será necesario crear un atributo nuevo llamado *type*, donde se almacenará el tipo de documento que se está guardando.

El fichero final con toda la información se guardará en un documento con el formato adecuado en función de la base de datos en la que se vaya a almacenar dicho documento para realizar las pruebas.

En los siguientes subapartados se explica la limpieza específica que se ha hecho para cada adaptar los datos a la base de datos en la que se van a introducir.

4.6.1 Limpieza de los datos para MongoDB

A la hora de almacenar información en MongoDB, lo mejor es hacerlo mediante un fichero en formato JSON. Para ello, se van guardando en un diccionario los datos que necesitamos del fichero XML, siendo estos los tipos de documentos y atributos escogidos. A

continuación, este diccionario se guardará en un fichero JSON llamado *dblp.json*. Un ejemplo de la estructura resultante de este documento se puede ver en la Figura 17. En el [Anexo A](#) está disponible el código de limpieza junto con una explicación.

```
{
  "dblp": [
    {
      "author": [
        {
          "Author": "Sanjeev Saxena"
        }
      ],
      "type": "article",
      "title": "Parallel Integer Sorting and Simulation Amongst CRCW Models.",
      "year": 1996
    },
    {
      "author": [
        {
          "Author": "Hans Ulrich Simon"
        }
      ],
      "type": "article",
      "title": "Pattern Matching in Trees and Nets.",
      "year": 1983
    },
    {
      "author": [
        {
          "Author": "Nathan Goodman"
        },
        {
          "Author": "Oded Shmueli"
        }
      ],
      "type": "article",
      "title": "NP-complete Problems Simplified on Tree Schemas.",
      "year": 1983
    }
  ],
}
```

Figura 17. Esquema resultante de la limpieza para la base de datos de MongoDB.

4.6.2 Limpieza de los datos para Neo4j

Para almacenar los datos en Neo4j, el formato ideal es CSV. Debido al *dataset* elegido, se ha tomado la decisión de que existan los siguientes elementos en el grafo:

- Nodos *Author*. En ellos se guardan los nombres de los autores.
- Nodos *Publication*. En ellos se guardan los datos relativos a los diversos documentos: título, año y tipo.
- Relaciones *Has_Author*. Son las que relacionan los nodos *Publication* y los nodos *Author* e indica los autores que tiene una publicación.

Por lo tanto, para este caso concreto serán necesarios tres ficheros CSV:

- *authors.csv*, donde se almacenan los nodos *Author*.
- *publications.csv*, donde se almacenan los nodos *Publication*.
- *PA_relationships.csv*, donde se almacenan las relaciones *Has_Author*.

En el [Anexo B](#) está disponible el código de limpieza explicado.

4.7 EJECUCIÓN DE LAS CONSULTAS

Como paso previo a la ejecución de las consultas, es necesario arrancar el servidor e importar los datos a las bases de datos. A continuación, hay que transformar las consultas a los lenguajes correspondientes de cada producto. Por último, se ejecutarán las consultas sobre los dos productos, obteniendo así tanto los resultados de cada una de ellas como los tiempos de respuesta.

4.7.1 Ejecución de las consultas en MongoDB

Antes de ejecutar las consultas en la base de datos es necesario realizar dos acciones. La primera es arrancar el servidor. Para ello, se abre una terminal de Windows y se escriben dos mandatos. Esto se puede ver en la Figura 18.

```
> cd C:\Program Files\MongoDB\Server\3.6\bin
> mongod.exe
```

```
C:\Users\Ester>cd C:\Program Files\MongoDB\Server\3.6\bin
C:\Program Files\MongoDB\Server\3.6\bin>mongod.exe
2018-07-10T04:45:29.248-0700 I CONTROL [initandlisten] MongoDB starting : pid=11960 port=27017 dbpath=C:\data\db\ 64-bit host=DESKTOP-5HCTT0C
2018-07-10T04:45:29.248-0700 I CONTROL [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2018-07-10T04:45:29.248-0700 I CONTROL [initandlisten] db version v3.6.2
2018-07-10T04:45:29.248-0700 I CONTROL [initandlisten] git version: 489d177dbd0f0420a8ca04d39fd78d0a2c539420
2018-07-10T04:45:29.248-0700 I CONTROL [initandlisten] OpenSSL version: OpenSSL 1.0.1u-fips 22 Sep 2016
2018-07-10T04:45:29.249-0700 I CONTROL [initandlisten] allocator: tcmalloc
2018-07-10T04:45:29.249-0700 I CONTROL [initandlisten] modules: none
2018-07-10T04:45:29.249-0700 I CONTROL [initandlisten] build environment:
2018-07-10T04:45:29.249-0700 I CONTROL [initandlisten] distmod: 2008plus-ssl
2018-07-10T04:45:29.249-0700 I CONTROL [initandlisten] distarch: x86_64
2018-07-10T04:45:29.249-0700 I CONTROL [initandlisten] target_arch: x86_64
2018-07-10T04:45:29.249-0700 I CONTROL [initandlisten] options: {}
2018-07-10T04:45:29.250-0700 I - [initandlisten] Detected data files in C:\data\db\ created by the 'wiredTiger' storage engine, so setting the ac
tive storage engine to 'wiredTiger'.
2018-07-10T04:45:29.250-0700 I STORAGE [initandlisten] wiredtiger_open config: create,cache_size=9677M,session_max=20000,eviction=(threads_min=4,thread
s_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_idle_time=100000),statist
ics_log=(wait=0),verbose=(recovery_progress),
2018-07-10T04:45:29.415-0700 I STORAGE [initandlisten] WiredTiger message [1531223129:414712][11960:140705859124304], txn-recover: Main recovery loop:
starting at 141/31232
2018-07-10T04:45:29.536-0700 I STORAGE [initandlisten] WiredTiger message [1531223129:535380][11960:140705859124304], txn-recover: Recovering log 141 t
hrough 142
2018-07-10T04:45:29.618-0700 I STORAGE [initandlisten] WiredTiger message [1531223129:618129][11960:140705859124304], txn-recover: Recovering log 142 t
hrough 142
2018-07-10T04:45:29.728-0700 I CONTROL [initandlisten]
2018-07-10T04:45:29.728-0700 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2018-07-10T04:45:29.728-0700 I CONTROL [initandlisten] ** Read and write access to data and configuration is unrestricted.
2018-07-10T04:45:29.729-0700 I CONTROL [initandlisten]
2018-07-10T04:45:29.729-0700 I CONTROL [initandlisten] ** WARNING: This server is bound to localhost.
2018-07-10T04:45:29.731-0700 I CONTROL [initandlisten] ** Remote systems will be unable to connect to this server.
2018-07-10T04:45:29.731-0700 I CONTROL [initandlisten] ** Start the server with --bind_ip <address> to specify which IP
2018-07-10T04:45:29.731-0700 I CONTROL [initandlisten] ** addresses it should serve responses from, or with --bind_ip_all to
2018-07-10T04:45:29.732-0700 I CONTROL [initandlisten] ** bind to all interfaces. If this behavior is desired, start the
2018-07-10T04:45:29.734-0700 I CONTROL [initandlisten] ** server with --bind_ip 127.0.0.1 to disable this warning.
2018-07-10T04:45:29.734-0700 I CONTROL [initandlisten]
2018-07-10T04:45:29.735-0700 I CONTROL [initandlisten]
2018-07-10T04:45:29.735-0700 I CONTROL [initandlisten] ** WARNING: The file system cache of this machine is configured to be greater than 40% of the to
tal memory. This can lead to increased memory pressure and poor performance.
2018-07-10T04:45:29.735-0700 I CONTROL [initandlisten] See http://dochub.mongodb.org/core/wt-windows-system-file-cache
2018-07-10T04:45:29.736-0700 I CONTROL [initandlisten]
2018-07-10T13:45:30.089+0200 W FTDC [initandlisten] Failed to initialize Performance Counters for FTDC: WindowsPdhError: PdhExpandCounterPathW failed
with 'El objeto especificado no se encontró en el equipo.' for counter 'Memory\Available Bytes'
2018-07-10T13:45:30.089+0200 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directory 'C:\data\db\diagnostic.data'
2018-07-10T13:45:30.094+0200 I NETWORK [initandlisten] waiting for connections on port 27017
```

Figura 18. Mandatos para iniciar el servidor de MongoDB en una terminal de Windows.

La segunda es importar el fichero *dblp.json* a la base de datos. Esto se intenta mediante el siguiente mando:

```
mongoimport --db dblp --collection documents --file dblp.json --jsonArray
```

En este caso, este mandato produce un error, por lo que se importará el fichero JSON con un script de Python, utilizando la librería *pymongo*. El código se puede ver en el [Anexo C](#). [37].

Ya se pueden ejecutar las consultas en la base de datos. Como la mayoría de ellas necesitan funciones del *aggregation framework* y, como ya se explicó en el punto 4.3, no es posible medir los tiempos de respuesta directamente, se ejecutarán las consultas con un script de Python, donde también está incluido el código para el cálculo del tiempo de respuesta y la conexión con la base de datos. El código completo se puede ver en el [Anexo D](#). [38]

A continuación, se pueden ver las consultas en el lenguaje propio de MongoDB junto con una breve explicación, el resultado de la consulta y el tiempo de respuesta de cada una. Hay dos tipos de tiempo de respuesta, los que se producen sin la utilización de índices y los que se producen habiendo definido índices. Los índices que se crean para MongoDB son 2: $\{“year”:1\}$ y $\{“Type”:1\}$, ya que se realizan varias consultas usando esos atributos. Para los autores no se crea ningún índice ya que están almacenados en listas y los índices, en estos casos, no proporcionan resultados fiables ya que no funcionan bien.

En la Figura 19 se puede ver el proceso de creación de índices. En un primer momento, el único índice que existe es “_id”, que se crea por defecto. En ese estado se ejecutan por primera vez las consultas. A continuación, se crean los dos índices indicados con anterioridad. Por último, se puede ver que la base de datos cuenta ahora con 3 índices.



Figura 19. Proceso de creación de índices para MongoDB.

CONSULTA 1: Mostrar todas las publicaciones de la base de datos.**CÓDIGO:**

```
db.documents.find()
```

EXPLICACIÓN: En esta consulta se realiza un recorrido de la base de datos mediante la función *find*, que devuelve todos los documentos almacenados en la base de datos.

RESULTADO DE LA CONSULTA:

```
> db.documents.find().pretty()
{
  "_id" : ObjectId("5b453436b1d5b10be0c5ea96"),
  "author" : [
    {
      "Author" : "Sanjeev Saxena"
    }
  ],
  "type" : "article",
  "title" : "Parallel Integer Sorting and Simulation Amongst CRCW Models.",
  "year" : 1996
}
{
  "_id" : ObjectId("5b453436b1d5b10be0c5ea97"),
  "author" : [
    {
      "Author" : "Hans Ulrich Simon"
    }
  ],
  "type" : "article",
  "title" : "Pattern Matching in Trees and Nets.",
  "year" : 1983
}
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 1: Mostrar todas las publicaciones de la base de datos
0.00 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 1: Mostrar todas las publicaciones de la base de datos
0.00 ms
```

CONSULTA 2: Mostrar todos los artículos en libros.**CÓDIGO:**

```
db.documents.find({"type":"incollection"})
```

EXPLICACIÓN: Se hace un filtrado de la base de datos, cogiendo solo aquellos documentos cuyo tipo se corresponde con *incollection*

RESULTADO DE LA CONSULTA:

```
{
  "_id" : ObjectId("5b453436b1d5b10be0c5ea96"),
  "author" : [
    {
      "Author" : "Sanjeev Saxena"
    }
  ],
  "type" : "article",
  "title" : "Parallel Integer Sorting and Simulation Amongst CRCW Models.",
  "year" : 1996
}
{
  "_id" : ObjectId("5b453436b1d5b10be0c5ea97"),
  "author" : [
    {
      "Author" : "Hans Ulrich Simon"
    }
  ],
  "type" : "article",
  "title" : "Pattern Matching in Trees and Nets.",
  "year" : 1983
}
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 2: Mostrar todos los artículos en libros.
0.00 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 2: Mostrar todos los artículos en libros.
0.00 ms
```

CONSULTA 3: Publicaciones entre los años 1990 y 2000.**CÓDIGO:**

```
db.documents.find({$and:[
    {"year":{"$gte":"1990"}},
    {"year":{"$lte":"2000"}}] })
```

EXPLICACIÓN: Se realiza un filtrado de la base de datos con el operador lógico “and”. Se muestran aquellos documentos cuyo año es mayor o igual a 1990 y menor o igual a 2000.

RESULTADO DE LA CONSULTA:

```
> db.documents.find({$and:[{"year":{"$gte":1990}},{"year":{"$lte":2000}}]}).pretty()
{
  "_id" : ObjectId("5b453436b1d5b10be0c5ea96"),
  "author" : [
    {
      "Author" : "Sanjeev Saxena"
    }
  ],
  "type" : "article",
  "title" : "Parallel Integer Sorting and Simulation Amongst CRCW Models.",
  "year" : 1996
}
{
  "_id" : ObjectId("5b453436b1d5b10be0c5eaa3"),
  "author" : [
    {
      "Author" : "Carol Critchlow"
    },
    {
      "Author" : "Prakash Panangaden"
    }
  ],
  "type" : "article",
  "title" : "The Expressive Power of Delay Operators in SCCS.",
  "year" : 1991
}
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 3: Publicaciones entre los años 1990 y 2000
0.00 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 3: Publicaciones entre los años 1990 y 2000
0.00 ms
```

CONSULTA 4: Número de artículos que hay en la base de datos.**CÓDIGO:**

```
db.documents.find({"type":"article"}).count()
```

EXPLICACIÓN:

En esta consulta, se filtra por tipo de documento, para coger solo los de tipo “article”. A continuación, se realiza un recuento mediante la función de agregación “count” para saber cuántos documentos de este tipo hay en la base de datos.

RESULTADO DE LA CONSULTA:

```
> db.documents.find({"type":"article"}).count()
1793615
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 4: Número de artículos en la base de datos
1931.25 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 4: Número de artículos en la base de datos
690.35 ms
```


CONSULTA 5: Publicaciones por año de un autor determinado.**CÓDIGO:**

```
db.documents.aggregate([{$match:{"author.Author":"Lila Kari"}},
                        {$group:{"_id":"$year","total":{$sum:1}}},
                        {$sort:{"_id":1}}, {allowDiskUse:true})
```

EXPLICACIÓN:

El autor que se ha escogido es Lila Kari. En primer lugar, se cogen aquellos documentos en cuya lista de autores aparezca Lila Kari. A continuación, se agrupan los resultados por año y se realiza un acumulador que vaya sumando los elementos del grupo. Por último, se ordena por año para que sea más adecuado visualmente. En esta memoria es necesario añadir *allowDiskUse:true* al final para que no se produzca un fallo de memoria y se pueda ejecutar la consulta.

RESULTADO DE LA CONSULTA:

```
> db.documents.aggregate([{$match:{"author.Author":"Lila Kari"}},
...  {$group:{"_id":"$year","total":{$sum:1}}},
...  {$sort:{"_id":1}},
...  {allowDiskUse:true})
{ "_id" : 1992, "total" : 2 }
{ "_id" : 1993, "total" : 6 }
{ "_id" : 1994, "total" : 5 }
{ "_id" : 1995, "total" : 5 }
{ "_id" : 1996, "total" : 6 }
{ "_id" : 1997, "total" : 3 }
{ "_id" : 1998, "total" : 2 }
{ "_id" : 1999, "total" : 7 }
{ "_id" : 2000, "total" : 3 }
{ "_id" : 2001, "total" : 7 }
{ "_id" : 2002, "total" : 4 }
{ "_id" : 2003, "total" : 4 }
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 5: Publicaciones por año de un autor determinado (Lila Kari)
7646.59 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 5: Publicaciones por año de un autor determinado (Lila Kari)
12458.67 ms
```

CONSULTA 6: Número de documentos con más de 3 autores.**CÓDIGO:**

```
db.documents.aggregate([{$unwind:"$author"},
                        {$group:{"_id":"$title","Authors":{$sum:1}}},
                        {$match:{"Authors":{$gt:3}}},
                        {$count:"Documentos"}],{allowDiskUse:true})
```

EXPLICACIÓN:

Para poder realizar el cálculo, lo primero es agrupar por documento, sumando el número de autores que tiene cada uno en su lista de autores. A continuación, se escogen aquellos cuyo acumulador sea mayor que 3. Por último, se cuenta el número de documentos que han pasado el filtro.

RESULTADO DE LA CONSULTA:

```
> db.documents.aggregate([{$unwind:"$author"},
...  {$group:{"_id":"$title","Authors":{$sum:1}}},
...  {$match:{"Authors":{$gt:3}}},
...  {$count:"Documentos"}],
...  {allowDiskUse:true})
{ "Documentos" : 1169841 }
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 6: Número de documentos con más de 3 autores
39441.45 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 6: Número de documentos con más de 3 autores
39528.62 ms
```

CONSULTA 7: Coautores de un autor concreto.**CÓDIGO:**

```
db.documents.aggregate([{$match:{"author.Author":"Lila Kari"}},
  {$unwind:"$author"},
  {$group:{"_id":"$author.Author"}},
  {$match:{"_id":{"$ne":"Lila Kari"}}}])
```

EXPLICACIÓN:

En primer lugar, se cogen aquellos documentos en los que aparece Lila Kari en su lista de autores. Para cada documento, se separa la lista de autores, quedando como resultado el documento tantas veces repetido como autores haya en la lista. Para los documentos resultantes se realizará una agrupación por autores y, de esos autores, pasarán el filtro todos aquellos que no sean Lila Kari.

RESULTADO DE LA CONSULTA:

```
> db.documents.aggregate([{$match:{"author.Author":"Lila Kari"}},
...   {$unwind:"$author"},
...   {$group:{"_id":"$author.Author"}},
...   {$match:{"_id":{"$ne":"Lila Kari"}}}])
{ "_id" : "Zachary Kincaid" }
{ "_id" : "Rani Siromoney" }
{ "_id" : "Hanan Lutfiyya" }
{ "_id" : "Przemyslaw Prusinkiewicz" }
{ "_id" : "Laura F. Landweber" }
{ "_id" : "Carlos Martín-Vide" }
{ "_id" : "Leonard M. Adleman" }
{ "_id" : "Salah Hussini" }
{ "_id" : "Donald Sannella" }
{ "_id" : "Giorgio Ausiello" }
{ "_id" : "Kai Salomaa" }
{ "_id" : "Helmut Jürgensen" }
{ "_id" : "Masami Ito" }
{ "_id" : "Mark Perry" }
{ "_id" : "Juraj Hromkovic" }
{ "_id" : "Cezar Cămpăanu" }
{ "_id" : "Andrei Paun" }
{ "_id" : "Jarkko Kari" }
{ "_id" : "Natasa Jonoska" }
{ "_id" : "Mark Daley" }
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 7: Coautores de un autor determinado (Lila Kari)
7897.91 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 7: Coautores de un autor determinado (Lila Kari)
10360.28 ms
```

CONSULTA 8: Autores que han publicado documentos de un solo tipo.**CÓDIGO:**

```
db.documents.aggregate([{$unwind:"$author"},
                        {$group:{"_id":"$author.Author","Tipos":{"addToSet":"$type"}}},
                        {$out:"AutoresTipos"}],{allowDiskUse:true})

db.AutoresTipos.aggregate([{$unwind:"$Tipos"},
                           {$group:{"_id":"$_id","types":{"sum:1},"unique_type":{"push":"$Tipos"}}},
                           {$match:{"types":{"$eq:1}}}],{allowDiskUse:true}).pretty()
```

EXPLICACIÓN:

Para realizar esta consulta, hay que dividirla en dos. En la primera parte, se separa la lista de autores, quedando como resultado el documento repetido tantas veces como autores haya en esa lista. A continuación, se agrupa por autor, añadiendo los tipos de esa lista a un set (para evitar repeticiones de los tipos). Este resultado se guarda en una nueva colección, formada por documentos con dos campos: nombre del autor y conjunto de tipos que ha publicado. Se añade *allowDiskUse:true* para evitar problemas de memoria.

RESULTADO DE LA CONSULTA:

```
> show collections
documents
> db.documents.aggregate([{$unwind:"$author"},
... {$group:{"_id":"$author.Author","Tipos":{"addToSet":"$type"}}},
... {$out:"AutoresTipos"}],
... {allowDiskUse:true})
> show collections
AutoresTipos
documents
> db.AutoresTipos.aggregate([{$unwind:"$Tipos"},
... {$group:{"_id":"$_id","types":{"sum:1},"unique_type":{"push":"$Tipos"}}},
... {$match:{"types":{"$eq:1}}}],
... {allowDiskUse:true}).pretty()
{
  "_id" : "'Maseka Lesaana",
  "types" : 1,
  "unique_type" : [
    "article"
  ]
}
{
  "_id" : "(David) Jing Dai",
  "types" : 1,
  "unique_type" : [
    "article"
  ]
}
{
  "_id" : "(Max) Zong-Ming Cheng",
  "types" : 1,
  "unique_type" : [
    "article"
  ]
}
{
  "_id" : "(Zhou) Bryan Bai",
  "types" : 1,
  "unique_type" : [
    "Inproceedings"
  ]
}
```

TIEMPO DE RESPUESTA SIN ÍNDICES:

```
CONSULTA 8: Autores que han publicado documentos de un solo tipo
68448.01 ms
```

TIEMPO DE RESPUESTA CON ÍNDICES:

```
CONSULTA 8: Autores que han publicado documentos de un solo tipo
65546.72 ms
```

A continuación, se van a calcular los tiempos de respuesta para las mismas consultas en la base de datos diseñada para Neo4j.

4.7.2 Ejecución de las consultas en Neo4j

Lo primero que hay que hacer para trabajar con Neo4j es importar los ficheros CSV creados en la limpieza y que añadirán los nodos y las relaciones a la base de datos. Para ello, hay crear una carpeta llamada *import* dentro de la carpeta de *neo4j-community-3.4.0-alpha10*.

En ella se pegarán los tres ficheros CSV. A continuación, se abre una terminal en Windows y se realiza el siguiente mandato que se puede ver en la Figura 20.

```
> cd D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10
> bin\neo4j-admin import --mode csv --database dblp.db --nodes
import\authors.csv --nodes import\documents.csv --relationships import\PA_relationships.csv
```

```
D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10>bin\neo4j-admin import --mode csv --database dblp.db --nodes impor
t\authors.csv --nodes import\documents.csv --relationships import\PA_relationships.csv
ADVERTENCIA: This command does not appear to be running with administrative rights. Some commands may fail e.g.
Start/Stop
Neo4j version: 3.4.0-alpha10
Importing the contents of these files into D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10\data\databases\dblp.db:

Nodes:
  D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10\import\authors.csv
  D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10\import\documents.csv
Relationships:
  D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10\import\PA_relationships.csv

Available resources:
  Total machine memory: 19.90 GB
  Free machine memory: 15.29 GB
  Max heap memory : 4.42 GB
  Processors: 8
  Configured max memory: 13.93 GB
  High-I/O: false

Import starting 2018-07-12 21:59:34.900+0200
  Estimated number of nodes: 6.25 M
  Estimated number of node properties: 20.63 M
  Estimated number of relationships: 13.17 M
  Estimated number of relationship properties: 0.00
  Estimated disk space usage: 1.40 GB
  Estimated required memory usage: 1.07 GB

InteractiveReporterInteractions command list (end with ENTER):
  c: Print more detailed information about current stage
  i: Print more detailed information
```

```
(1/4) Node import 2018-07-12 21:59:35.015+0200
  Estimated number of nodes: 6.25 M
  Estimated disk space usage: 1002.97 MB
  Estimated required memory usage: 1.07 GB
..... 5%
..... 10%
..... 15%
..... 20%
..... 25%
..... 30%
..... 35%
..... 40%
..... 45%
..... 50%
..... 55%
..... 60%
..... 65%
..... 70%
..... 75%
..... 80%
..... 85%
..... 90%
..... 95%
..... 100%
```

```
(2/4) Relationship import 2018-07-12 22:00:04.472+0200
  Estimated number of relationships: 13.17 M
  Estimated disk space usage: 427.18 MB
  Estimated required memory usage: 1.07 GB
..... 5%
..... 10%
..... 15%
..... 20%
..... 25%
..... 30%
..... 35%
..... 40%
..... 45%
..... 50%
..... 55%
..... 60%
..... 65%
..... 70%
..... 75%
..... 80%
..... 85%
..... 90%
..... 95%
..... 100%
```

```
(3/4) Relationship linking 2018-07-12 22:00:20.454+0200
  Estimated required memory usage: 1.06 GB
..... 5%
..... 10%
..... 15%
..... 20%
..... 25%
..... 30%
..... 35%
..... 40%
..... 45%
..... 50%
..... 55%
..... 60%
..... 65%
..... 70%
..... 75%
..... 80%
..... 85%
..... 90%
..... 95%
..... 100%
```

```
(4/4) Post processing 2018-07-12 22:00:25.645+0200
  Estimated required memory usage: 1020.01 MB
..... 5%
..... 10%
..... 15%
..... 20%
..... 25%
..... 30%
..... 35%
..... 40%
..... 45%
..... 50%
..... 55%
..... 60%
..... 65%
..... 70%
..... 75%
..... 80%
..... 85%
..... 90%
..... 95%
..... 100%
```

```
IMPORT DONE in 53s 470ms.
Imported:
  6071162 nodes
  11744123 relationships
  20120702 properties
Peak memory usage: 1.09 GB
D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10>
```

Figura 20. Carga de los ficheros CSV a la base de datos de Neo4j.

Tras crear la base de datos, es necesario añadir en el fichero de configuración de Neo4j *neo4j.conf* la sentencia *dbms.active_database=dblp*. Esto hace que, al arrancar la consola de Neo4j, se coja por defecto la base de datos indicada en la sentencia.

A continuación, hay que arrancar el servidor. Una pequeña guía de cómo hacerlo se encuentra en la carpeta *neo4j-community-3.4.0-alpha10* que se crea en la instalación, en el fichero README.txt. Para iniciar el servidor, se abre una terminal en Windows y se escriben los siguientes mandatos cuyo resultado se puede ver en la Figura 21.

```
> cd D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10
> bin\neo4j console
```

```
D:\>cd D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10
D:\Usuarios\Ester\Documents\neo4j-community-3.4.0-alpha10>bin\neo4j console
ADVERTENCIA: This command does not appear to be running with administrative rights. Some commands may fail e.g.
Start/Stop
2018-07-10 22:24:52.567+0000 INFO ===== Neo4j 3.4.0-alpha10 =====
2018-07-10 22:24:52.616+0000 INFO Starting...
2018-07-10 22:24:57.526+0000 INFO Bolt enabled on 127.0.0.1:7687.
2018-07-10 22:25:01.115+0000 INFO Started.
2018-07-10 22:25:02.218+0000 WARN Low configured threads: (max={} - required={} )={ } < warnAt={ } for { }
2018-07-10 22:25:02.243+0000 INFO Remote interface available at http://localhost:7474/
```

Figura 21. Mandatos para arrancar el servidor de Neo4j.

Como se ve en la Figura 21, al final aparece la siguiente dirección:

<http://localhost:7474/>

Será necesario abrir el navegador e introducir dicha dirección en la barra de direcciones. Con esto, ya tenemos acceso a la base de datos y se pueden ejecutar las consultas. En la Figura 22 se muestra la interfaz de Neo4j.

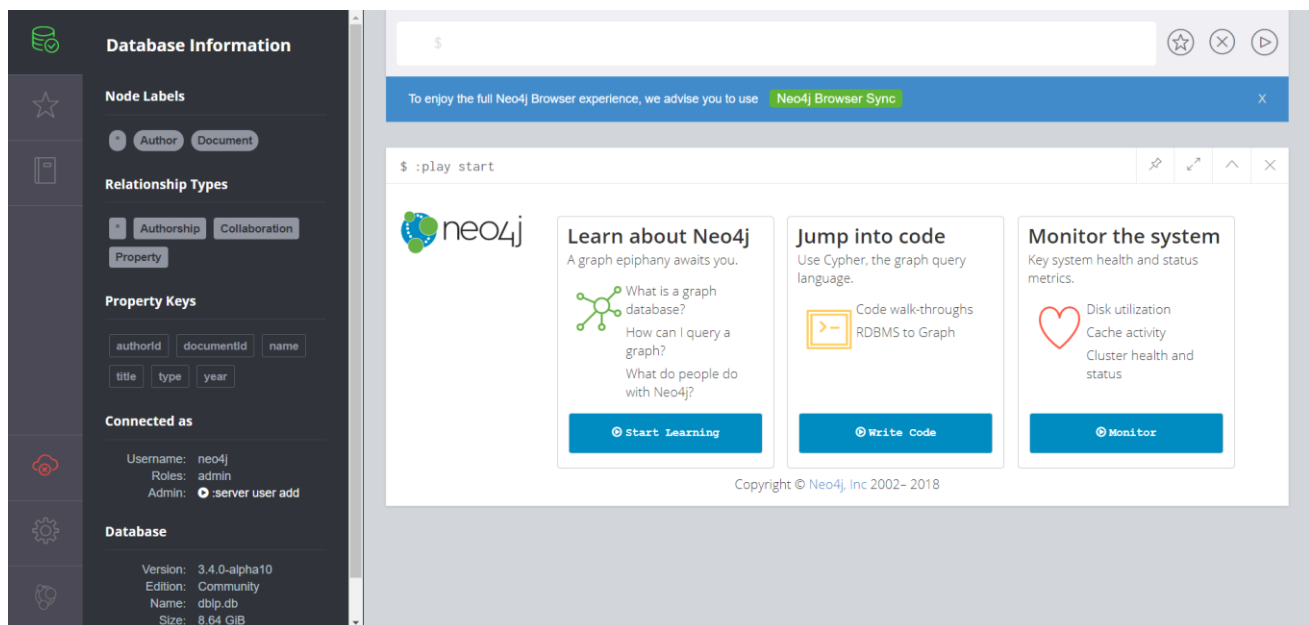


Figura 22. Interfaz de Neo4j en el navegador.

Por último, es necesario transformar las consultas a lenguaje Cypher. Para ello, se tomarán como base las consultas de MongoDB, ya que es más fácil la conversión desde ese lenguaje que desde el lenguaje natural. A continuación, se pueden ver las consultas en el lenguaje propio de Neo4j junto con una breve explicación, el resultado de la consulta y el tiempo de respuesta de cada una. Hay dos tipos de tiempo de respuesta, los que se producen sin la utilización de índices y los que se producen habiendo definido índices. Los índices que se crean para Neo4j son 3, *:Author(name)*, *:Publication(year)* y *:Publication(Type)* ya que se realizan varias consultas usando esos atributos. No será necesario definirlos en las consultas ya que Neo4j selecciona automáticamente los índices existentes.

En la Figura 23 se puede ver el proceso de creación de índices. En un primer momento, no existe ningún índice. En ese estado se ejecutan por primera vez las consultas. A continuación, se crean los tres índices indicados con anterioridad. Por último, se puede ver que la base de datos cuenta ahora con 3 índices.

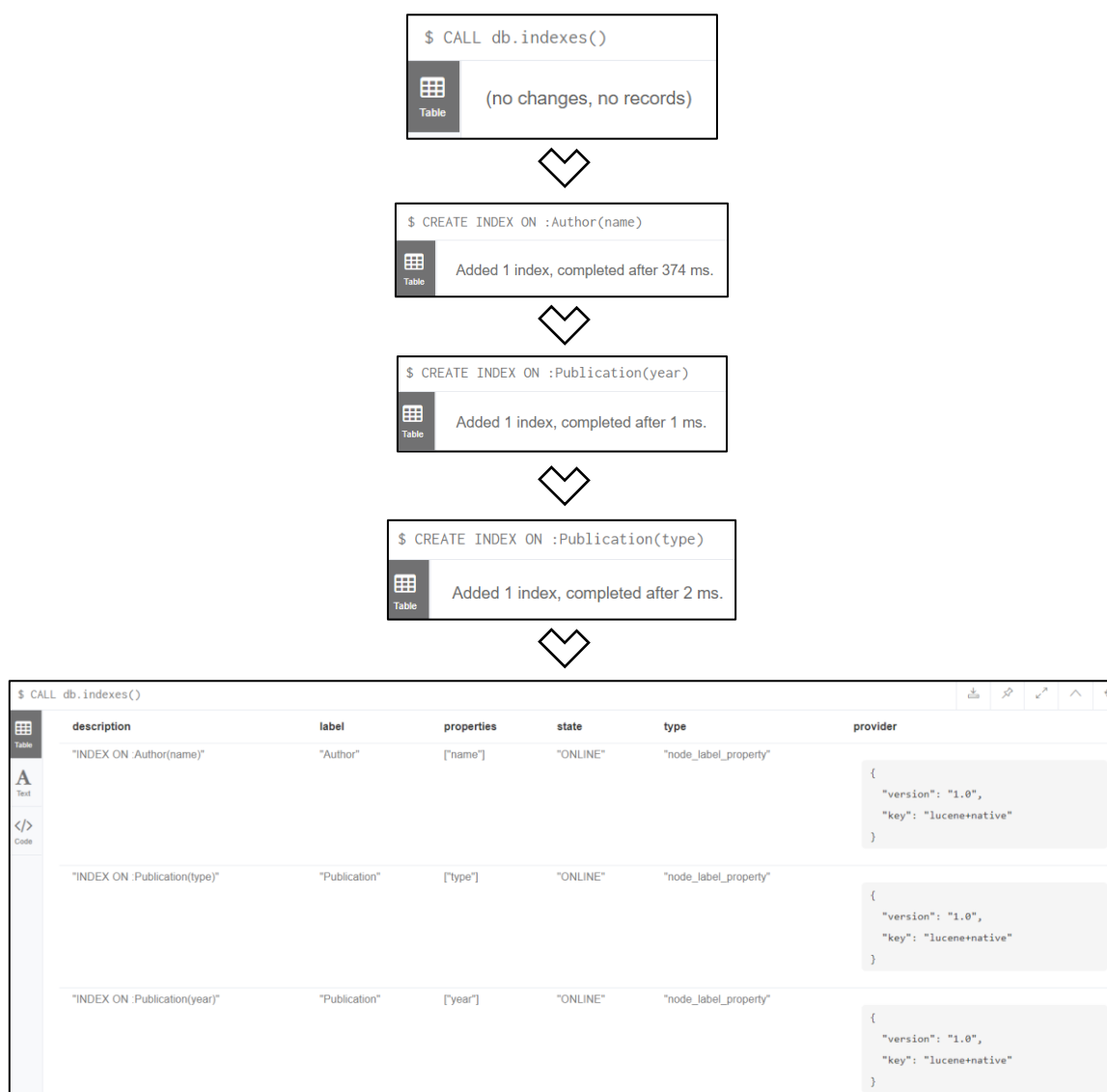


Figura 23. Creación de índices para Neo4j.

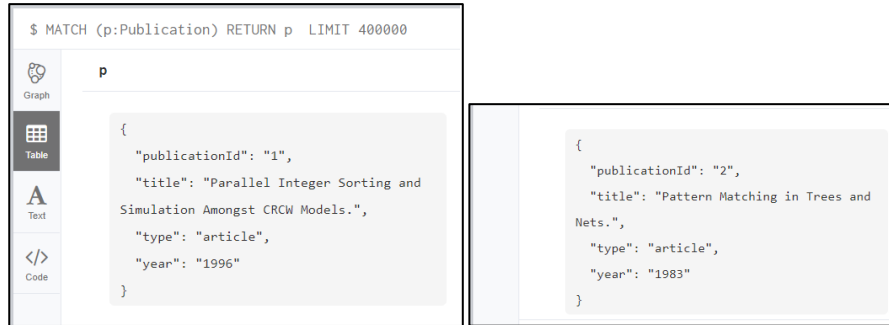
CONSULTA 1: Mostrar todas las publicaciones de la base de datos.

CÓDIGO:

```
MATCH (p:Publication)
RETURN p
LIMIT 400000
```

EXPLICACIÓN: Para esta consulta se cogen todos los nodos de tipo publicación y se muestran. Es necesario poner un límite porque si no la base de datos se colapsa debido a la gran cantidad de nodos que existen (casi 4 millones).

RESULTADO DE LA CONSULTA



TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 400000 records after 301 ms and completed after 5930 ms, displaying first 1000 rows.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 400000 records after 1 ms and completed after 2138 ms, displaying first 1000 rows.

CONSULTA 2: Mostrar todos los artículos en libros.

CÓDIGO:

```
MATCH (p:Publication)
WHERE p.type = "incollection"
RETURN p AS Incollections
```

EXPLICACIÓN: Se cogen todos los nodos de tipo publicación y se filtran para quedarnos con aquellos que son de tipo *incollection* (artículos en libros). Con el término *AS* se le da un nombre a la columna en la que se devuelven los datos.

RESULTADO DE LA CONSULTA



TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 46750 records after 2 ms and completed after 9824 ms, displaying first 1000 rows.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 46750 records after 354 ms and completed after 1150 ms, displaying first 1000 rows.

CONSULTA 3: Publicaciones entre los años 1990 y 2000.**CÓDIGO:**

```
MATCH (p:Publication)
WHERE toInteger(p.year) >= 1990 AND toInteger(p.year) <= 2000
RETURN p AS Publications_1990_2000
```

EXPLICACIÓN: Se cogen los nodos de tipo publicación. Para cada uno, se transforma el atributo *year* a tipo *integer*, ya que al proceder de un fichero CSV estaba como tipo *string* y se le aplican dos condiciones, que sea mayor que 1990 y menor que 2000.

RESULTADO DE LA CONSULTA

\$ MATCH (p:Publication) WHERE toInteger(p.year) >= 1990 AND toInteger(p.year) <= 2000 RETURN p AS Publications_1990_2000	
Graph	Publications_1990_2000
Table	<pre>{ "publicationId": "1", "title": "Parallel Integer Sorting and Simulation Amongst CRCW Models.", "type": "article", "year": "1996" }</pre>
Text	
Code	<pre>{ "publicationId": "14", "title": "The Expressive Power of Delay Operators in SCCS.", "type": "article", "year": "1991" }</pre>

TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 482014 records after 1 ms and completed after 11629 ms, displaying first 1000 rows.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 482014 records in less than 1 ms and completed after 6690 ms, displaying first 1000 rows.

CONSULTA 4: Número de artículos que hay en la base de datos.**CÓDIGO:**

```
MATCH (p:Publication)
WHERE p.type = "article"
RETURN count(p) AS Number_of_Articles
```

EXPLICACIÓN: Se cogen los nodos de tipo publicación y se les aplica una condición para devolver aquellos que son de tipo *article*. Por último, se hace un recuento para saber el número de publicaciones que cumplen esta condición.

RESULTADO DE LA CONSULTA

\$ MATCH (p:Publication) WHERE p.type = "article" RETURN count(p) AS Number_of_Articles	
Graph	Number_of_Articles
Table	1793615

TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 1 records after 3092 ms and completed after 3092 ms.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 1 records after 1300 ms and completed after 1300 ms.

CONSULTA 5: Publicaciones por año de un autor determinado.**CÓDIGO:**

```
MATCH (p:Publication)-[r:Has_Author]->(a:Author{name:"Lila Kari"})
RETURN count(p.title) AS Publications, p.year
ORDER BY p.year
```

EXPLICACIÓN: Para esta consulta, como relaciona publicaciones y autores, es necesario coger aquellos elementos de la base de datos que cumplan la relación que une los nodos publicación con los nodos autor. Para este caso, además, se especifica que solo son válidos aquellas publicaciones que se relacionan con el nodo autor cuyo atributo *name* tiene el valor "Lila Kari". A continuación, se realiza un recuento de las publicaciones que cumplen esta relación y se agrupan por año. Por último, se ordenan los resultados para una mejor visualización.

RESULTADO DE LA CONSULTA

\$ MATCH (p:Publication)-[r:Has_Author]->(a:Author{name:"Lila Kari"}) RETURN count(p.title) AS Publications, p.year ORDER BY p.year

Publications	p.year
2	"1992"
6	"1993"
5	"1994"
5	"1995"
6	"1996"
3	"1997"
2	"1998"
7	"1999"
3	"2000"
7	"2001"
4	"2002"
4	"2003"
8	"2004"
11	"2005"
5	"2006"
4	"2007"

TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 27 records after 7710 ms and completed after 7711 ms.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 27 records after 467 ms and completed after 467 ms.

CONSULTA 6: Número de documentos con más de 3 autores.**CÓDIGO:**

```
MATCH (p:Publication)-[r:Has_Author]->(a:Author)
WITH p, count(a) AS Num_Authors
WHERE Num_Authors > 3
RETURN count(p)
```

EXPLICACIÓN: En esta consulta se cogen los elementos que cumplen la relación *Has_Author*. Para cada documento que cumple esta relación, se cuenta el número de autores conectados y, si este número es mayor que 3, se añadirá al contador que se va a devolver.

RESULTADO DE LA CONSULTA

\$ MATCH (p:Publication)-[r:Has_Author]->(a:Author) WITH p, count(a) AS Num_Authors WHERE Num_Authors > 3 RETURN count(p)

count(p)
1156362

TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 1 records after 490005 ms and completed after 490005 ms.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 1 records after 29114 ms and completed after 29114 ms.

CONSULTA 7: Coautores de un autor concreto.**CÓDIGO:**

```
MATCH (p:Publication)-[r:Has_Author]->(a:Author{name:"Lila Kari"}),
      (p:Publication)-[r:Has_Author]->(o:Author)
WHERE o.name <> "Lila Kari"
RETURN DISTINCT o.name
```

EXPLICACIÓN: Hay que coger aquellos nodos *Publication* que cumplen dos relaciones, la primera es que estén relacionados con el nodo *Author* de Lila Kari. La segunda es que estén relacionados con nodos *Author*. Al filtrar en la primera relación los nodos por un autor concreto y, en la segunda relación, llamar a los nodos *Publication* de la misma forma, los nodos *Publication* de la segunda relación serán todos aquellos que, entre sus autores, tengan un autor con el nombre de Lila Kari. A continuación, queremos quedarnos con aquellos autores que no se llamen Lila Kari, por lo que habrá que coger todos los nodos *Author* de la segunda y comparar los nombres. Por último, como puede haber repeticiones de autores, será necesario devolver cada nombre una única vez.

RESULTADO DE LA CONSULTA

The screenshot shows the MongoDB query interface with the following query: `$ MATCH (p:Publication)-[r:Has_Author]->(a:Author{name:"Lila Kari"}), (p:Publication)->(o:Author) WHERE o.name <> "Lila Kari" RETURN DISTINCT o.name`. The results are displayed in a table with the column `o.name`. The list of authors includes: "Arto Salomas", "Oleorghe Plean", "Solomon Marcus", "Rani Sirononey", "Oleg Oleg", "Mark Daley", "Shinosuke Seki", "Bo Cu 0001", "Carlos Martin Vider", "Hanan Luliyys", "Kajana Mahalingam", "Natalia Jonoska", "Eugen Czeidler", "Elena Czeidler", "Gabriel Thierin", and "Petr Souk".

o.name
"Arto Salomas"
"Oleorghe Plean"
"Solomon Marcus"
"Rani Sirononey"
"Oleg Oleg"
"Mark Daley"
"Shinosuke Seki"
"Bo Cu 0001"
"Carlos Martin Vider"
"Hanan Luliyys"
"Kajana Mahalingam"
"Natalia Jonoska"
"Eugen Czeidler"
"Elena Czeidler"
"Gabriel Thierin"
"Petr Souk"

TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 74 records after 1 ms and completed after 3753 ms.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 74 records after 2 ms and completed after 4 ms.

CONSULTA 8: Autores que han publicado documentos de un solo tipo.**CÓDIGO:**

```
MATCH (p:Publication) -[r:Has_Author]-> (a:Author)
WITH a, count(DISTINCT p.type) as Types
WHERE Types = 1
RETURN a.name
```

EXPLICACIÓN: Primero hay que coger los nodos que cumplen la relación definida. A continuación, para cada nodo *Author* que cumplen esa relación, se calculan los tipos, sin repetición, de todas las publicaciones que han elaborado. Si este contador es igual a 1, es decir, solo hay un tipo, se devolverá el nombre del autor.

RESULTADO DE LA CONSULTA

The screenshot shows the MongoDB query interface with the following query: `$ MATCH (p:Publication) -[r:Has_Author]-> (a:Author) WITH a, count(DISTINCT p.type) as Types WHERE Types = 1 RETURN a.name`. The results are displayed in a table with the column `a.name`. The list of authors includes: "Xianggang Zhang", "Cristina Riua", "Hanna Isaksson", "Kenneth M. Brown", "Elin Ramon", "Yu-Fan Hsuah", "James D. Eynard", "José A. Scott", "Ekaterina N. Skachko", "Maria Teresa Lameiras", "M. Khammar", "Marek Dawgaj", "Armel Le Bar", "Tomás Dorta", "Merja Kyti", and "B. R. Chen".

a.name
"Xianggang Zhang"
"Cristina Riua"
"Hanna Isaksson"
"Kenneth M. Brown"
"Elin Ramon"
"Yu-Fan Hsuah"
"James D. Eynard"
"José A. Scott"
"Ekaterina N. Skachko"
"Maria Teresa Lameiras"
"M. Khammar"
"Marek Dawgaj"
"Armel Le Bar"
"Tomás Dorta"
"Merja Kyti"
"B. R. Chen"

TIEMPO DE RESPUESTA SIN ÍNDICES:

Started streaming 1459553 records after 244141 ms and completed after 249551 ms, displaying first 1000 rows.

TIEMPO DE RESPUESTA CON ÍNDICES:

Started streaming 1459553 records after 27425 ms and completed after 32835 ms, displaying first 1000 rows.

4.8 COMPARATIVA DE LOS RESULTADOS

Una vez ejecutadas las consultas, es necesario comparar los resultados obtenidos. Para facilitar la comparación, en la Figura 24 se pueden ver las consultas que se han realizado.

CONSULTA 1: Mostrar todas las publicaciones de la base de datos.
 CONSULTA 2: Mostrar todos los artículos en libros.
 CONSULTA 3: Publicaciones entre los años 1990 y 2000.
 CONSULTA 4: Número de artículos en la base de datos.
 CONSULTA 5: Publicaciones de un autor concreto por año.
 CONSULTA 6: Número de documentos con más de 3 autores.
 CONSULTA 7: Coautores de un autor concreto.
 CONSULTA 8: Autores que sólo han publicado un único tipo de documento.

Figura 24. Consultas ejecutadas en las bases de datos.

En la Tabla 3 es posible ver una comparativa de los tiempos entre productos y, dentro de cada producto, en función del uso de índices. Se encuentran marcados los tiempos más bajos para cada consulta.

Producto	Índices	CONSULTAS							
		1	2	3	4	5	6	7	8
MondoDB	SIN	0.00 ms	0.00 ms	0.00 ms	1931 ms	7646 ms	39441 ms	7897 ms	68448 ms
	CON	0.00 ms	0.00 ms	0.00 ms	690 ms	12458 ms	39528 ms	10360 ms	65546 ms
Neo4j	SIN	5930 ms	9824 ms	11629 ms	3092 ms	7711 ms	490005 ms	3753 ms	249551 ms
	CON	2138 ms	1150 ms	6690 ms	1300 ms	467 ms	29114 ms	4 ms	32835 ms

Tabla 3. Tabla comparativa de los tiempos de respuesta.

Como se puede ver en la Tabla 3, las diferencias entre ambos productos son significativas. En primer lugar, se van a comparar los resultados obtenidos con MongoDB. A continuación, se van a comparar los resultados obtenidos con Neo4j. Por último, se van a comparar los resultados de ambos productos.

Para MongoDB, se puede ver que no hay diferencias significativas para el rendimiento entre realizar las consultas con índices o sin ellos. Si se realiza un análisis por consulta, se observa lo siguiente:

- Para las tres primeras consultas los tiempos son iguales. Son consultas tan rápidas que no les afecta la definición de índices.
- Para la consulta 4, al emplear un índice que ordena los tipos de documento por orden alfabético y pedir aquellos documentos que sean de tipo *article*, es entendible que el uso de ese índice mejore el rendimiento.

- Para la consulta 5, empeora el rendimiento con los índices porque se pretende ordenar las publicaciones de un autor por año. Como lo primero que se hace es separar las listas de autores de cada documento, es posible que la base de datos ordene dichos documentos por año, con lo que, al realizar la búsqueda del autor, ésta se va realizando año por año, reduciendo la eficiencia. Si los autores no hubieran estado recogidos en listas, se hubiese podido crear un índice para ellos y la consulta hubiera sido mucho más rápida.
- Para las consultas 6 y 7 la diferencia es mínima, ya que son consultas que se realizan sobre los autores de los documentos y no se ven afectadas por los índices.
- Para la consulta 8, al estar involucrado el atributo *type*, es lógico que exista una pequeña diferencia, pero, al tardar con ambas opciones más de un minuto, no se puede decir que la opción con uso de índices mejore el rendimiento demasiado.

Como conclusión para MongoDB se puede decir que, de manera general, daría igual usar índices o no. Pero, si sólo se realizan consultas de uno de los tipos que se han probado, podría llegar a mejorar el rendimiento de la base de datos la definición de índices.

Para Neo4j es posible observar una gran mejora del rendimiento con el uso de índices. Esto es así porque en la base de datos de este producto, al estar los autores separados de los documentos, se han podido crear un índice específico para ellos, que claramente afecta al rendimiento de la base de datos. Con la combinación de los tres índices diseñados se aumenta la eficiencia de las consultas, logrando grandes diferencias de tiempos.

A la hora de comparar ambos productos entre sí, se puede decir que las diferencias entre ambos son notables. Estas diferencias se pueden separar en dos grupos.

- Para las consultas 1-4, que en MongoDB se realizan con una función *find*, los tiempos en este producto son menores que en Neo4j. Esto puede deberse a que la búsqueda en colecciones es mucho más eficiente con el uso de esa función.
- Para las consultas 5-8, el rendimiento de Neo4j es mayor. Los tiempos que destacan se corresponden con el uso de índices lo que nos indica que la definición del índice que ordena los autores proporciona una mejora muy significativa en cuanto al rendimiento. Además, se observa que los mejores tiempos de Neo4j con respecto a MongoDB son aquellos en los que se las consultas trabajan con las relaciones entre publicaciones y autores, aprovechando la característica que define las Bases de Datos Orientadas a

Grafos. Por ejemplo, para la consulta 7 en la que se obtienen los coautores, la búsqueda es muy superior en Neo4j, lo que demuestra la importancia de las relaciones en este tipo de base de datos ya que en MongoDB los autores se encuentran como atributo de los documentos, mientras que en Neo4j los autores forman nodos independientes que se obtienen mediante su relación con los documentos. Otro motivo para que el rendimiento sea mayor con Neo4j es que estas consultas en MongoDB emplean *aggregation framework*, lo que permite realizar consultas más complejas y completas, pero aumenta los tiempos de respuesta.

Como conclusión general se puede decir que, claramente, el rendimiento del producto depende en gran medida del tipo de consultas que se van a realizar, así como del diseño de la base de datos.

Como conclusión específica para los productos, es difícil llegar a un acuerdo sobre qué producto es mejor. Claramente, Neo4j es recomendable si nos interesan las relaciones entre los elementos de la base de datos, pero si necesitamos realizar consultas sobre la totalidad de la colección, es más eficiente MongoDB.

5. CONCLUSIONES Y TRABAJOS FUTUROS

En este último capítulo se va a realizar un análisis del trabajo realizado, así como una definición de los pasos a seguir para poder mejorar esta comparativa.

5.1 CONCLUSIONES

Como ya se indicó en el punto 1.2, el objetivo principal de este trabajo era el de realizar un análisis comparativo del rendimiento de distintas consultas aplicadas a varios tipos de productos de Bases de Datos NoSQL. Como se puede ver en el punto 4.8, este objetivo se ha cumplido, ya que en él se puede ver una comparación del rendimiento para dos productos NoSQL, MongoDB y Neo4j.

Por otro lado, también se han cumplido los múltiples objetivos parciales, ya que se ha realizado un análisis de la evolución histórica de los sistemas de almacenamiento. Se han estudiado los distintos tipos de Bases de Datos NoSQL y, con lo aprendido, se ha desarrollado el caso de estudio. Este último objetivo parcial, se desglosó en varios objetivos más pequeños, que se han ido cumpliendo de manera sucesiva, ya que para realizar un caso de estudio, se ha tenido que elegir un dataset, diseñar las consultas más adecuadas, diseñar las bases de datos que tendrían que dar los resultados de la manera más adecuada posible para dichas consultas, se limpió el dataset para adaptarlo a las bases de datos diseñadas, se ejecutaron las consultas sobre dichas bases de datos y se realizó una comparativa de los resultados obtenidos.

A nivel personal, la realización de este trabajo ha sido muy positiva, ya que me ha permitido estudiar las Bases de Datos NoSQL y aplicarlas en un caso práctico, algo que no había visto en la asignatura de Bases de Datos. Además, he podido aplicar y profundizar en varios conceptos aprendidos con la realización del Máster en *Data Science*, como son el uso de los dos productos utilizados en el caso de estudio (MongoDB y Neo4j) o la limpieza de datos.

Como conclusión, he de decir que las Bases de Datos NoSQL me han parecido muy útiles y deberían enseñarse junto con las Bases de Datos Relacionales en el grado, ya que ambos tipos conviven en la actualidad y es importante conocer las características de ambos para poder tomar decisiones respecto a su uso.

5.2 TRABAJOS FUTUROS

Terminado el caso de estudio, es el momento de pensar de qué forma se podría mejorar el análisis de rendimiento. Algunas mejoras podrían ser las siguientes:

- Comparar más productos NoSQL. Esta mejora se puede dar de dos formas:

- Añadir productos de otros tipos de Bases de Datos NoSQL, como pueden ser Cassandra (Base de Datos de Familia de Columnas) y Redis (Bases de Datos Clave-Valor)
- Añadir productos diferentes para los mismos tipos. Couchbase y Microsoft Azure son dos productos de los dos tipos de bases de datos que se han visto en el análisis, Base de Datos Orientada a Documentos y Base de Datos Orientada a Grafos respectivamente.
- Diseñar consultas específicas para cada tipo o producto y aplicarlas al resto para analizar cómo se comportan.
- Realizar otro tipo de consultas a la base de datos. En este caso, todas las que se realizan son de búsqueda, pero sería conveniente realizar inserciones, borrados y actualizaciones para comprobar la eficiencia de los distintos productos.
- Comparar el rendimiento con distintos *datasets*. Sería conveniente variar la complejidad y el tamaño de los *datasets*, ya que la decisión para escoger un producto normalmente del tipo de datos que se va a almacenar.
- Añadir a la comparativa las Bases de Datos Relacionales, ya que son las principales competidoras de las Bases de Datos NoSQL.

Todas estas son las mejoras que se podrían realizar para mejorar el análisis del rendimiento y poder facilitar la toma de decisión en cuanto a qué tipo de base de datos o qué tipo de producto es mejor usar.

6. REFERENCIAS

1. Sullivan, D. (2015). *NoSQL for mere mortals*. Addison-Wesley.
2. Apuntes del máster. Asignatura “Bases de Datos No Convencionales”. Máster Data Science. Belén Vela Sánchez y Jose María Caverio Barca. 2018.
3. *DB-Engines Ranking*. Disponible en: <https://db-engines.com/en/ranking>
4. *Página oficial de Redis*. Disponible en: <https://redis.io/>
5. *Página oficial de MongoDB*. Disponible en: <https://www.mongodb.com/>
6. *Página oficial de Cassandra*. Disponible en: <http://cassandra.apache.org/>
7. *Página oficial de Neo4J*. Disponible en: <https://neo4j.com/>
8. Piattini, M. y de Miguel Castaño, A. (1999). *Fundamentos y modelos de bases de datos*. RA-MA.
9. *Principales características de las Bases de Datos Orientadas a Objetos*. Disponible en: <https://blog.powerdata.es/el-valor-de-la-gestion-de-datos/bid/404366/principales-caracter-sticas-de-las-bases-de-datos-orientadas-a-objetos>
10. *Bases de Datos Open Source. ¿Por qué elegimos Mysql para nuestro proyecto?* Disponible en: <https://churriwifi.wordpress.com/2010/01/03/bases-de-datos-opensource-¿porque-elegimos-mysql-para-nuestro-proyecto/>
11. *Base de Datos Multidimensional (MDB)*. Disponible en: <https://searchdatacenter.techtarget.com/es/definicion/Base-de-datos-multidimensional-MDB>
12. “SQL o NoSQL”, esa es la cuestión. Disponible en: <https://bites.futurespace.es/2017/12/01/sql-o-nosql-esa-es-la-cuestion/>
13. *Escalabilidad Horizontal y Vertical*. Disponible en: <https://www.oscarblancarteblog.com/2017/03/07/escalabilidad-horizontal-y-vertical/>
14. *Bases de Datos NoSQL: La guía definitiva*. Disponible en: <https://blog.pandorafms.org/es/bases-de-datos-nosql/>
15. *Bases de Datos NewSQL*. Disponible en: <http://www.ana2lp.mx/bases-de-datos/bases-de-datos-newsql/>
16. *Cassandra, la dama de las bases de datos NoSQL*. Disponible en: <https://www.paradigmadigital.com/dev/cassandra-la-dama-de-las-bases-de-datos-nosql/>
17. *Apache Cassandra*. Disponible en: https://es.wikipedia.org/wiki/Apache_Cassandra

18. *Neo4j: qué es y para qué sirve una base de datos orientada a grafos*. Disponible en: <https://bbvaopen4u.com/es/actualidad/neo4j-que-es-y-para-que-sirve-una-base-de-datos-orientada-grafos>
19. *MongoDB: qué es, cómo funciona y cuándo podemos usarlo (o no)*. Disponible en: <https://www.genbetadev.com/bases-de-datos/mongodb-que-es-como-functiona-y-cuando-podemos-usarlo-o-no>
20. *MongoDB*. Disponible en: <https://es.wikipedia.org/wiki/MongoDB>
21. *Redis in action*. Disponible en: <https://redislabs.com/ebook/part-1-getting-started/chapter-1-getting-to-know-redis/>
22. *No sólo clave-valor. Redis te da alas*. Disponible en: <https://www.paradigmadigital.com/techbiz/no-solo-clave-valor-redis-te-da-alas/>
23. *Base de datos Redis: Todo lo que debes saber*. Disponible en: <http://www.antweb.es/servidores/redis-todo-lo-que-debes-saber>
24. *Descarga de la base de datos DBLP*. Disponible en: <http://dblp.uni-trier.de/xml/>
25. *Documentación de la base de datos DBLP*. Disponible en: <http://dblp.uni-trier.de/xml/docu/dblp.xml.pdf>
26. *EmEditor*. Disponible en: <https://www.emeditor.com/>
27. *MongoDB Download Center*. Disponible en: <https://www.mongodb.com/download-center?jmp=nav#previous>
28. *Install MongoDB Community Edition on Windows*. Disponible en: <https://docs.mongodb.com/v3.6/tutorial/install-mongodb-on-windows/>
29. *Python Downloads*. Disponible en: <https://www.python.org/downloads/>
30. *Download Neo4j*. Disponible en: <https://neo4j.com/download/>
31. *The Neo4j Developer Manual*. Disponible en: <https://neo4j.com/docs/developer-manual/current/introduction/>
32. *MongoDB, función `cursor.explain()`*. Disponible en: <https://docs.mongodb.com/manual/reference/method/cursor.explain/>
33. *MongoDB, explicación de los resultados de la función `explain`*. Disponible en: <https://docs.mongodb.com/manual/reference/explain-results/#executionstats>
34. *MongoDB: How can I see the execution time for the aggregate command?*. Disponible en: <https://stackoverflow.com/questions/14021605/mongodb-how-can-i-see-the-execution-time-for-the-aggregate-command>

35. *Convert python datetime to timestamp in milliseconds.* Disponible en: <https://stackoverflow.com/questions/41635547/convert-python-datetime-to-timestamp-in-milliseconds/41635888>
36. *Datetime – Basic date and time types.* Disponible en: <https://docs.python.org/3.3/library/datetime.html#strptime-strptime-behavior>
37. *Pymongo.* Disponible en: <https://api.mongodb.com/python/3.6.1/>
38. *Pymongo: Aggregation Examples.* Disponible en: <https://api.mongodb.com/python/3.6.1/examples/aggregation.html>
39. *Can't get allowDiskUse:True to work with pymongo.* Disponible en: <https://stackoverflow.com/questions/27272699/cant-get-allowdiskusetrue-to-work-with-pymongo>

7. ANEXOS

7.1 ANEXO A: CÓDIGO FUENTE XML_TO_JSON.JSON

A la hora de realizar la limpieza del documento `dblp.xml` para obtener el documento `dblp.json` se siguen los siguientes pasos.

1. Se importan las librerías y paquetes necesarios.

```
import xml.etree.ElementTree as ET
import json
from collections import OrderedDict
```

2. Se crea un parseador para los caracteres especiales.

```
parser = ET.XMLParser(encoding='ISO-8859-1')

parser.entity["agrave"] = 'à'
parser.entity["uuml"] = 'ü'
parser.entity["Eacute"] = 'É'
parser.entity["eacute"] = 'é'
parser.entity["aacute"] = 'á'
parser.entity["iacute"] = 'í'
parser.entity["ouml"] = 'ö'
parser.entity["ccedil"] = 'ç'
parser.entity["egrave"] = 'è'
parser.entity["auml"] = 'ä'
parser.entity["uacute"] = 'ú'
parser.entity["aring"] = 'å'
parser.entity["oacute"] = 'ó'
parser.entity["szlig"] = 'ß'
parser.entity["oslash"] = 'ø'
parser.entity["yacute"] = 'ÿ'
parser.entity["iuml"] = 'ï'
parser.entity["igrave"] = 'ï'
parser.entity["ocirc"] = 'ô'
parser.entity["icirc"] = 'ï'
parser.entity["Uuml"] = 'Ü'
parser.entity["euml"] = 'ë'
parser.entity["acirc"] = 'â'
parser.entity["atilde"] = 'ã'
parser.entity["Uacute"] = 'Ù'
parser.entity["Aacute"] = 'À'
parser.entity["ntilde"] = 'ñ'
parser.entity["Auml"] = 'Ä'
parser.entity["Oslash"] = 'Ø'
parser.entity["Ccedil"] = 'Ç'
parser.entity["otilde"] = 'õ'
```

```
parser.entity["times"] = '×'
parser.entity["Ouml"] = 'Ö'
parser.entity["reg"] = '®'
parser.entity["Aring"] = 'Å'
parser.entity["Oacute"] = 'Ò'
parser.entity["ograve"] = 'ò'
parser.entity["yuml"] = 'ÿ'
parser.entity["eth"] = 'ð'
parser.entity["aelig"] = 'æ'
parser.entity["AElig"] = 'Æ'
parser.entity["Agrave"] = 'Á'
parser.entity["Iuml"] = 'İ'
parser.entity["micro"] = 'μ'
parser.entity["Acirc"] = 'Â'
parser.entity["Otilde"] = 'Õ'
parser.entity["Egrave"] = 'È'
parser.entity["ETH"] = 'Ð'
parser.entity["ugrave"] = 'ù'
parser.entity["ucirc"] = 'û'
parser.entity["thorn"] = 'þ'
parser.entity["THORN"] = 'Þ'
parser.entity["Iacute"] = 'Î'
parser.entity["Icirc"] = 'Ï'
parser.entity["Ntilde"] = 'Ñ'
parser.entity["Ecirc"] = 'Ê'
parser.entity["Ocirc"] = 'Ô'
parser.entity["Ograve"] = 'Ó'
parser.entity["Igrave"] = 'Í'
parser.entity["Atilde"] = 'Ã'
parser.entity["Yacute"] = 'Ý'
parser.entity["Ucirc"] = 'Û'
parser.entity["Euml"] = 'Ë'
```

3. Se parsea el documento “dblp.xml”

```
tree = ET.parse("dblp.xml", parser=parser)
root = tree.getroot()
```

4. Se crea el diccionario auxiliar donde se irán almacenando los datos que nos interesa mantener y las listas con los tipos de documentos y atributos que queremos guardar.

```
dblp = OrderedDict()
dblp["dblp"]=[]
document_types = ["article","inproceedings","incollection"]
attributes = ["title","author","year"]
```

5. Se recorre el fichero dblp.xml y se guarda en el diccionario auxiliar creado en el paso 3 el contenido de aquellas etiquetas que pertenecen a las listas también definidas en el paso 3.

```
for child in root:
    documento = {"author":[]}
    if child.tag in document_types:
        documento["type"]=child.tag
        for elem in child:
            if elem.tag in attributes:
                if elem.tag == "year":
                    documento[elem.tag]=int(elem.text)
                elif elem.tag == "author":
                    autor = {"Author":elem.text}
                    documento[elem.tag].append(autor)
            else:
                documento[elem.tag]=elem.text

    dblp["dblp"].append(documento)
```

6. Por último, el diccionario se guarda en un fichero JSON llamado “dblp.json”.

```
with open("dblp.json","w") as f:
    json.dump(dblp,f,indent=4)
```

7.2 ANEXO B: CÓDIGO FUENTE XML_TO_CSV.CSV

A la hora de realizar la limpieza del documento dblp.xml para obtener los distintos ficheros CSV se siguen los siguientes pasos.

1. Se importan las librerías y paquetes necesarios.

```
import xml.etree.ElementTree as ET
import pandas as pd
from collections import OrderedDict
```

2. Se crea un parseador para los caracteres especiales.

```
parser = ET.XMLParser(encoding='ISO-8859-1')
```

```
parser.entity["grave"] = 'à'
parser.entity["uuml"] = 'ü'
parser.entity["Eacute"] = 'É'
parser.entity["eacute"] = 'é'
parser.entity["aacute"] = 'á'
parser.entity["iacute"] = 'í'
parser.entity["ouml"] = 'ö'
parser.entity["ccedil"] = 'ç'
parser.entity["egrave"] = 'è'
parser.entity["auml"] = 'ä'
parser.entity["uacute"] = 'ú'
parser.entity["aring"] = 'å'
parser.entity["oacute"] = 'ó'
parser.entity["szlig"] = 'ß'
parser.entity["oslash"] = 'ø'
parser.entity["yacute"] = 'ÿ'
parser.entity["iuml"] = 'ï'
parser.entity["igrave"] = 'î'
parser.entity["ocirc"] = 'ô'
parser.entity["icirc"] = 'ï'
parser.entity["Uuml"] = 'Ü'
parser.entity["euml"] = 'ë'
parser.entity["acirc"] = 'â'
parser.entity["atilde"] = 'ã'
parser.entity["Uacute"] = 'Ù'
parser.entity["Aacute"] = 'À'
parser.entity["ntilde"] = 'ñ'
parser.entity["Auml"] = 'Ä'
parser.entity["Oslash"] = 'Ø'
parser.entity["Ccedil"] = 'Ç'
parser.entity["otilde"] = 'õ'
```

```
parser.entity["times"] = '×'
parser.entity["Ouml"] = 'Ö'
parser.entity["reg"] = '®'
parser.entity["Aring"] = 'Å'
parser.entity["Oacute"] = 'Ò'
parser.entity["ograve"] = 'ò'
parser.entity["yuml"] = 'ÿ'
parser.entity["eth"] = 'ð'
parser.entity["aelig"] = 'æ'
parser.entity["AElig"] = 'Æ'
parser.entity["Agrave"] = 'À'
parser.entity["Iuml"] = 'Ï'
parser.entity["micro"] = 'μ'
parser.entity["Acirc"] = 'Â'
parser.entity["Otilde"] = 'Õ'
parser.entity["Egrave"] = 'È'
parser.entity["ETH"] = 'Ð'
parser.entity["ugrave"] = 'ù'
parser.entity["ucirc"] = 'û'
parser.entity["thorn"] = 'þ'
parser.entity["THORN"] = 'Þ'
parser.entity["Iacute"] = 'Î'
parser.entity["Icirc"] = 'Î'
parser.entity["Ntilde"] = 'Ñ'
parser.entity["Ecirc"] = 'Ê'
parser.entity["Ocirc"] = 'Ô'
parser.entity["Ograve"] = 'Ò'
parser.entity["Igrave"] = 'Ì'
parser.entity["Atilde"] = 'Ã'
parser.entity["Yacute"] = 'ÿ'
parser.entity["Ucirc"] = 'Û'
parser.entity["Euml"] = 'Ë'
```

3. Se parsea el documento “dblp.xml”

```
tree = ET.parse("dblp.xml", parser=parser)
root = tree.getroot()
```

4. Se crea una lista con los tipos de documentos que queremos guardar.

```
publication_types = ["article", "inproceedings", "incollection"]
```

5. Se crean los diccionarios que se necesitan para los nodos y las relaciones entre ellos.

Nodos de los autores

```
authors_dict = OrderedDict()
authors_dict["authorId:ID(Authors)"] = []
authors_dict["name"] = []
authors_dict["LABEL"] = []
```

Nodos de los documentos

```
publications_dict = OrderedDict()
publications_dict["publicationId:ID(Publications)"] = []
publications_dict["title"] = []
publications_dict["type"] = []
publications_dict["year"] = []
publications_dict["LABEL"] = []
```

Relaciones entre documentos y autores

```
relationshipsPA_dict = OrderedDict()
relationshipsPA_dict["START_ID(Publications)"] = []
relationshipsPA_dict["END_ID(Authors)"] = []
relationshipsPA_dict["TYPE"] = []
```

6. Se inicializan los ids que se emplean para identificar autores y documentos.

```
authorId = 1
documentId = 1
```

7. Se crea un diccionario auxiliar para evitar repeticiones de los autores en los nodos.

```
aux_authors_dict = OrderedDict()
```


8. Se recorre el xml para crear los nodos que componen el grafo.

```
for child in root:
    if child.tag in publication_types:
        author_rel_list = []
        year = None
        for elem in child:
            if elem.tag == "title":
                # Creación nodo publicación
                publications_dict["publicationId:ID(Publications)"].append(publicationId)
                myId = publicationId
                publicationId += 1
                publications_dict["title"].append(elem.text)
                publications_dict[":LABEL"].append("Publication")
                publications_dict["type"].append(child.tag)
            elif elem.tag == "author":
                # Creación nodo autor (si no está ya)
                if elem.text not in aux_authors_dict:
                    authors_dict["authorId:ID(Authors)"].append(authorId)
                    aux_authors_dict[elem.text] = authorId
                    authorId += 1
                    authors_dict["name"].append(elem.text)
                    authors_dict[":LABEL"].append("Author")
                author_rel_list.append(aux_authors_dict[elem.text])
            elif elem.tag == "year":
                publications_dict["year"].append(int(elem.text))
                year = elem.text
        if year == None:
            publications_dict["year"].append(0)
        # Se crean las relaciones
        for author in range(len(author_rel_list)):
            # Relación publicación-autor
            relationshipsPA_dict["START_ID(Publications)"].append(myId)
            relationshipsPA_dict["END_ID(Authors)"].append(author_rel_list[author])
            relationshipsPA_dict[":TYPE"].append("Has_Author")
```

9. Una vez completados los diccionarios de nodos y relaciones se transforman en dataframe.

```
authors = pd.DataFrame(authors_dict)
publications = pd.DataFrame(publications_dict)
PA_relationships = pd.DataFrame(relationshipsPA_dict)
```

10. Se copian los objetos dataframe creados en el punto 9 en los ficheros CSV.

```
authors.to_csv("authors.csv", index=False)
publications.to_csv("documents.csv", index=False)
PA_relationships.to_csv("PA_relationships.csv", index=False)
```

7.3 ANEXO C: CÓDIGO FUENTE IMPORT_DB_MONGODB.PY

Para poder importar el dataset a MogoDB se siguen los siguientes pasos.

1. Importar las librerías necesarias.

```
import pymongo
import json
```

2. Realizar una conexión con MongoDB.

```
conex = pymongo.MongoClient()
```

3. Crear la base de datos y la colección en la que se van a guardar los documentos.

```
db = conex.dblp
col = db.documents
```

4. Cargar el documento “dblp.json” generado en la limpieza.

```
with open("dblp.json", 'r')
as f:
    my_json= json.load(f)
```

5. El atributo “dblp” del fichero “dblp.json” contiene en una lista todas las publicaciones con sus atributos. Se guarda esa lista en la variable “*documents*”.

```
documents =
```

6. Recorrer la lista creada en el punto 5 e ir insertando los documentos en la colección.

```
for doc in
documents:
    col.insert(doc)
```

7.4 ANEXO D: CÓDIGO FUENTE MONGODB_QUERY.PY

Para poder calcular los tiempos de respuesta de las consultas, es necesario realizar los siguientes pasos.

1. Importar las librerías necesarias y crear la conexión con la base de datos.

```
from pymongo import MongoClient
from datetime import datetime

db = MongoClient().dblp
```

2. Realizar las consultas. Todas ellas tendrán la misma estructura:

- a. Calcular el tiempo antes de la consulta.
- b. Ejecutar la consulta.
- c. Calcular el tiempo después de la consulta.
- d. Transformar ambos tiempos a milisegundos y restarlos.
- e. Imprimir el tiempo de respuesta.

Consulta 1: Mostrar todas las publicaciones de la base de datos

```
before_time = str(datetime.now())

result = db.documents.db.find()

after_time = str(datetime.now())

execution_time = answer_time(before_time,after_time)

#print(list(result))
print("{0:.2f}".format(execution_time) + "ms")
```

Consulta 2: Mostrar todos los artículos en libros.

```
before_time = str(datetime.now())

result = db.documents.find({"type":"incollection"})

after_time = str(datetime.now())

execution_time = answer_time(before_time,after_time)

#print(list(result))
print("{0:.2f}".format(execution_time) + "ms")
```

Consulta 3: Publicaciones entre los años 1990 y 2000

```
before_time = str(datetime.now())

result = db.documents.find({ "$and": [
    {"year": {"$gte": 1990}},
    {"year": {"$lte": 2000}} ]})

after_time = str(datetime.now())

execution_time = answer_time(before_time, after_time)

#print(result)
print("{0:.2f}".format(execution_time) + "ms")
```

Consulta 4: Número de artículos en la base de datos

```
before_time = str(datetime.now())

result = db.documents.find({"type": "article"}).count()

after_time = str(datetime.now())

execution_time = answer_time(before_time, after_time)

#print(list(result))
print("{0:.2f}".format(execution_time) + "ms")
```

Consulta 5: Publicaciones por año de un autor determinado.

```
before_time = str(datetime.now())

result = db.documents.aggregate([ {"match": {"author.Author": "Lila Kari"} },
    {"$group": {"_id": "$year", "total": {"$sum": 1}}},
    {"$sort": {"_id": 1}},
    {"allowDiskUse": true} ])

after_time = str(datetime.now())

execution_time = answer_time(before_time, after_time)

#print(list(result))
print("{0:.2f}".format(execution_time) + "ms")
```

Consulta 6: Número de documentos con más de 3 autores.

```

before_time = str(datetime.now())

result = db.documents.aggregate([{"$unwind":"$author"},
    {"$group":{"_id":"$title","Authors":{"$sum":1}}},
    {"$match":{"Authors":{"$gt":3}}},
    {"$count":"Documentos"}],
    {allowDiskUse:true})

after_time = str(datetime.now())

execution_time = answer_time(before_time,after_time)

#print(list(result))
print("{0:.2f}".format(execution_time) + "ms")

```

Consulta 7: Número de documentos con más de 3 autores.

```

before_time = str(datetime.now())

result = db.documents.aggregate([{"$match":{"author.Author":"Lila Kari"}},
    {"$unwind":"$author"},
    {"$group":{"_id":"$author.Author"}},
    {"$match":{"_id":{"$ne":"Lila Kari"}}}])

after_time = str(datetime.now())

execution_time = answer_time(before_time,after_time)

#print(list(result))
print("{0:.2f}".format(execution_time) + "ms")

```

Consulta 8: Número de documentos con más de 3 autores.

```

before_time = str(datetime.now())

db.documents.aggregate([{"$unwind":"$author"},
    {"$group":{"_id":"$author.Author","Tipos":{"$addToSet":"$type"}}},
    {"$out":"AutoresTipos"}],
    {allowDiskUse:true})

result = db.AutoresTipos.aggregate([{"$unwind":"$Tipos"},
    {"$group":{"_id":"$_id","types":{"$sum":1},"unique_type":{"$push":"$Tipos"}}},
    {"$match":{"types":{"$eq":1}}}],
    {allowDiskUse:true})

after_time = str(datetime.now())

```