UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Introducción a las bases de datos NoSQL. Sistemas de bases de datos orientados a grafos.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Tejero Gómez, Erik

Tutor/a: Mota Herranz, Laura

CURSO ACADÉMICO: 2021/2022

Resumen

La gestión de datos supone un desafío cuya dificultad se ha visto incrementada en un periodo de tiempo relativamente corto. Con el nacimiento de Internet, el manejo de grandes volúmenes de información ha pasado a ser un problema y una oportunidad de negocio al mismo tiempo.

En este trabajo, se realizará un recorrido sobre la evolución de las tecnologías destinadas al almacenamiento de datos. Tras ello, una visión más detallada sobre las bases de datos NoSQL aportarán un conocimiento específico sobre esta alternativa de modelado de datos empleada actualmente. Una vez conocidos el contexto en el que nos encontramos y las tecnologías disponibles, nos centramos en las bases de datos en grafo. Esta tecnología, distante de las bases de datos tradicionales, se centra en la importancia de las relaciones entre entidades a la hora de modelarlas basándose en el mundo real. De esta forma, se desarrollarán las principales características, peculiaridades y puntos fuertes de esta tecnología empleada por grandes empresas tecnológicas.

Con el objetivo de poner en práctica los conocimientos adquiridos y demostrar la eficacia del modelado de datos en grafo, un ejemplo práctico real construido en Neo4j concluirá el proyecto. Mediante la representación de la infraestructura de metro de Valencia, se realizarán consultas sobre la base de datos poblada mediante el lenguaje Cypher y el uso de algoritmos específicos sobre grafos como Dijkstra.

Por último, una conclusión sobre las diferentes alternativas tecnológicas disponibles pondrá punto y final al trabajo, incluyendo un punto de vista general y una reflexión subjetiva sobre el panorama actual.

Palabras clave: Bases de datos NoSQL, bases de datos en grafo.

Abstract

Data management is a challenge whose difficulty has increased in a relatively short period of time. With the birth of the Internet, handling large volumes of information has become a problem and a business opportunity at the same time.

In this work, a journey will be made on the evolution of technologies for data storage. After that, a more detailed view of NoSQL databases will provide specific knowledge about this currently used data modeling alternative. Once we know the context and the available technologies, we will focus on graph databases. This technology, distant from traditional databases, focuses on the importance of relationships between entities when modeling them based on the real world. In this way, the main characteristics, peculiarities and strengths of this technology used by large technology companies will be developed.

In order to put into practice the acquired knowledge and demonstrate the effectiveness of graph data modeling, a real practical example built in Neo4j will conclude the project. Through the representation of the Valencia metro infrastructure, queries will be made on the database populated using the Cypher language and the use of specific algorithms on graphs such as Dijkstra.

Finally, a conclusion on the different technological alternatives available will complete the work, including an overall point of view and a subjective thinking about the current panorama.

Keywords: *NoSQL database, graph database.*

Resum

La gestió de dades suposa un repte la dificultat del qual s'ha vist incrementada en un període de temps relativament curt. Amb el naixement de l'Internet, el maneig de grans volúmens d'informació ha passat a ser un problema i una oportunitat de negoci alhora.

En aquest treball, es realitzarà un recorregut sobre l'evolució de les tecnologies destinades a l'emmagatzematge de dades. A continuació, una visió més detallada de les bases de dades NoSQL proporcionarà un coneixement específic d'aquesta alternativa de modelat de dades utilitzada actualment. Una vegada coneguts el context en el que ens trobem i les tecnologies disponibles, ens centrem en les bases de dades en graf. Aquesta tecnologia, llunyana de les bases de dades tradicionals, es centra en la importància de les relacions entre entitats a l'hora de modelarles basant-se en el món real. D'aquesta forma, es desenvoluparan les principals característiques, peculiaritats i punts forts d'aquesta tecnologia utilitzada per grans empreses tecnològiques.

Amb l'objectiu de posar en pràctica el coneixements adquirits i demostrar l'eficàcia del modelat de dades en graf, un exemple pràctic real construït en Neo4j concluirà el projecte. Mitjançant la representació de la infraestructura del metre de València, es realitzaran consultes sobre la base de dades poblada mitjançant el llenguatge Cypher i l'ús d'algoritmes específics sobre grafs com Dijkstra.

Per últim, una conclusió sobre les diferents alternatives tecnològiques disponibles posarà punt i final al treball, incloent un punt de vista general i una reflexió subjectiva sobre el panorama actual.

Paraules clau: *Base de dades NoSQL, base de dades en graf.*

Índice

Índice de Ilustraciones

Índice de tablas

1 Introducción

En las últimas décadas el uso de nuevas tecnologías se ha incorporado de manera radical a todos los ámbitos de la sociedad. Tras este hecho, surge una gran problemática: la gestión de los datos generados.

En paralelo a la evolución y mejoras tecnológicas desarrolladas, han surgido modelos y estrategias cuyo objetivo ha sido optimizar esa gestión de información. Desde las bases de datos relacionales hasta los últimos modelos NoSQL, se han producido adaptaciones a un ecosistema cambiante tras la aparición e implantación de la Web en un periodo de tiempo relativamente corto.

Actualmente se conocen más de 200 arquitecturas diferentes NoSQL y ese número sigue creciendo. Estos almacenes de datos NoSQL se pueden categorizar en cuatro tipos según cómo se almacenan esos datos: Clave-valor, Columnar, Documental y en Grafo

En este trabajo fin de grado se persiguen tres objetivos:

- Revisar las características de esos cuatro tipos de bases de datos y presentar las estrategias empleadas para gestionar los datos.
- Profundizar en las bases de datos en grafos de manera teórica, presentando sus características, limitaciones y trascendencia.
- Presentar la solución a un caso práctico de utilidad real en un sistema de bases de datos orientado a grafos.

Con este proyecto, se pretende introducir al lector en las tecnologías de gestión de la información, haciendo hincapié en los modelos orientados a grafos y destacando su enorme utilidad en la actualidad.

1.1 Objetivos

El objetivo de este proyecto es la introducción a los diferentes sistemas de bases de datos NoSQL existentes y el modelado de los datos en grafo en detalle. Tras un desarrollo teórico, se pretende poner en práctica los conocimientos adquiridos mediante una aplicación práctica.

Para ello, el objetivo principal del proyecto se encuentra subdividido en cuatro subobjetivos a conseguir:

1. Presentar el contexto en el que se ha producido la evolución de las tecnologías de almacenamiento de datos.
2. Introducir las diferentes alternativas de bases de datos NoSQL, presentando sus principales características y puntos fuertes.
3. Desarrollar teóricamente propiedades básicas en el almacenamiento de datos como son la consistencia y la fragmentación, entrando en detalle sobre las diversas implementaciones disponibles actualmente. Además, se presenta el teorema CAP

englobando los múltiples modelos existentes y estableciendo unas reglas sobre las propiedades de los mismos.

4. Implementar con un sistema de gestión de bases de datos en grafo un caso de uso poniendo en práctica los conocimientos adquiridos sobre un escenario real.

1.2 Motivación

La gestión de la información en la actualidad resulta un desafío en el ámbito de las tecnologías y el modelado de datos. Por ello, una visión global de las diferentes alternativas existentes resulta de gran ayuda al tratar de hacer frente a su aplicación en escenarios del día a día.

La principal motivación que me ha llevado a afrontar el proyecto ha sido mi interés en el almacenamiento de los datos, en especial sobre el desafío que supone gestionar grandes volúmenes de información. Debido a que las bases de datos NoSQL son relativamente recientes y presentan unas características especiales con respecto a las tradicionales, resultan más llamativas.

Con motivo de la evolución de la web y en particular de las redes sociales, me resulta interesante poder conocer la implementación del almacenamiento de los datos sobre grafos. Por ello, mediante la lectura de artículos y documentos en referencia a este tema pretendo poder resolver un desafío de gestión de datos sobre un escenario real como es el que se presenta en la última parte del proyecto.

Finalmente, me resulta motivador el hecho de aprender sobre la gestión de datos en la actualidad y poder emplear una plataforma destinada a ello como es Neo4j y un lenguaje (Cypher) empleado únicamente con este fin.

1.3 Metodologías

La metodología empleada para el cumplimiento de los objetivos descritos anteriormente se resume en los siguientes pasos:

- Realizar una revisión bibliográfica sobre la evolución de las bases de datos, así como de las bases de datos NoSQL en detalle.
- Explorar las principales características y ventajas de un modelo de almacenamiento de datos frente a otros, comparando las conclusiones de diferentes autores sobre la materia.
- Adquirir conocimiento sobre las tecnologías específicas de bases de datos en grafo, como son Neo4j y Cypher.
- Realizar una reflexión sobre los datos recogidos y conocimientos adquiridos durante la búsqueda de información para elaborar una conclusión.

A partir de estas metodologías se pretende obtener las herramientas necesarias para la elaboración del trabajo, a partir de una importante base teórica y ciertos conocimientos prácticos para tratar de demostrar los argumentos presentes el proyecto.

1.4 Estructura de la memoria

En este trabajo de fin de grado se ha optado por la estructuración de la memoria en los siguientes apartados:

Apartado 1: establecimiento de los objetivos a conseguir mediante el desarrollo de la memoria, las motivaciones para llevarlo a cabo y las metodologías seguidas.

Apartado 2: presentación de la evolución de las bases de datos hasta la aparición de las bases de datos NoSQL. Una vez establecido el punto de partida de la tecnología a desarrollar, se presentan las diferentes alternativas de bases de datos NoSQL, describiendo sus principales características y ventajas. A continuación, se detallan algunas propiedades como la consistencia y fragmentación debido a su importancia en el almacenamiento de información, desarrollando el teorema CAP y su aplicación en este ámbito.

Apartado 3: mediante un desarrollo también teórico y presentación de casos de uso y principales características, se pretende facilitar la comprensión sobre las bases de datos en grafo y demostrar su eficacia en comparación con el resto de las alternativas disponibles.

Apartado 4: el uso de Neo4j y el lenguaje de programación Cypher suponen las herramientas más populares para implementar la tecnología comentada, por lo que se realiza un desarrollo sobre éstas y sus usos en la actualidad.

Apartado 5: planteamiento, diseño (conceptual, lógico y físico) y solución del problema planteado: implementación de la red de Metrovalencia. Aplicación de los conocimientos adquiridos y demostración de la eficacia del modelo presentado.

Apartado 6: conclusiones tras el estudio del tema desarrollado.

Apartado 7: bibliografía.

Apartado 8: objetivos de desarrollo sostenible.

1.5 Plan de trabajo

A continuación, se van a establecer las técnicas empleadas para planificar y estimar el esfuerzo de las diferentes fases en las que se dividirá el desarrollo del proyecto, además de las horas necesarias para elaborar cada una de las fases.

1.5.1 Planificación de las fases

Para estimar el tiempo que se va a dedicar a cada una de las fases del proyecto, es necesario destacar que dichas fases se repartirán en jornadas de 8 horas. Una vez aclarado este matiz, podemos apreciar que el proyecto está compuesto por las siguientes fases:

- Revisión bibliográfica
- Desarrollo teórico

- Diseño de la red de Metrovalencia
- Extracción y carga de datos
- Explotación

A continuación, se presenta el diagrama de Gantt en donde se muestran los días en los que se han llevado a cabo las tareas. Además de la fecha de inicio y finalización de cada una de ellas, se especifica el número de jornadas que han ocupado dichas tareas, donde el valor "1" significa una jornada completa, es decir, 8 horas y el valor "½" alude a media jornada, es decir, 4 horas.

	REVISIÓN BIBLIOGRÁFICA	DESARROLLO TEÓRICO	DISEÑO DE LA RED DE METROVALENCIA	EXTRACCIÓN Y CARGA DE DATOS	EXPLOTACIÓN
INICIO	01/12/21	04/03/22	15/04/22	14/05/22	01/06/22
FINALIZACIÓN	04/03/22	15/04/22	14/05/22	1/06/22	23/06/22
JORNADAS	16	8	2.5	4	2
ESTADO	Terminado	Terminado	Terminado	Terminado	Terminado
01/12/21	1				
4/12/21	1				
13/12/21	1				
22/12/21	1				
23/12/21	1				
30/12/21	½				
25/01/22	½				
27/01/22	1				
2/02/22	1				
4/02/22	1				
11/02/22	½				
14/02/22	½				
23/02/22	1				
26/02/22	1				
30/02/22	1				
04/03/22	1				
6/03/22		1			
14/03/22		1			
23/03/22		1			
26/03/22		1			
28/03/22		½			
30/03/22		1			
1/04/22	½	½			
10/04/22		1			
15/04/22	½	½			
17/04/22			1		
25/04/22		½	½		
14/05/22			1		
16/05/22	½			½	
19/05/22				1	
22/05/22				1	
23/05/22	½			½	
1/06/22				1	
6/06/22					1
23/06/22					1

Tabla 1 Diagrama de Gantt: Planificación de las fases y estimación temporal

1.5.2 Presupuesto

El costo económico para el desarrollo del proyecto se puede obviar debido a que se han empleado herramientas disponibles de forma gratuita en la plataforma de la universidad. Por tanto, el único costo del proyecto han sido las horas dedicadas por el alumno.

2 Bases de datos NoSQL

Las bases de datos NoSQL (un término acuñado a principios del siglo XXI en referencia a no-SQL o "no-relacional") proporcionan un mecanismo de almacenamiento y recuperación de datos distinto al utilizado en las bases de datos relacionales [Beeri90]. De esta forma, establecen unas estrategias sobre la gestión de los datos distantes al modelo tradicional, motivadas por la aparición de la Web 2.0, el *Big Data* y las aplicaciones web en tiempo real. El contexto en el que nace esta tecnología es de gran relevancia, ya que justifica las estrategias tomadas para cada tipo de base de datos NoSQL.

2.1 Evolución de las tecnologías de bases de datos

La evolución de los sistemas de almacenamiento de datos se ha visto condicionada principalmente por el tamaño de esta información. Desde la primera generación a mediados de los años 60 hasta la actualidad comprobamos como, paralelamente a los avances tecnológicos, se logra optimizar de forma progresiva el manejo de estos datos. Podemos ilustrar en orden cronológico los cambios en el diseño que se han llevado a cabo:

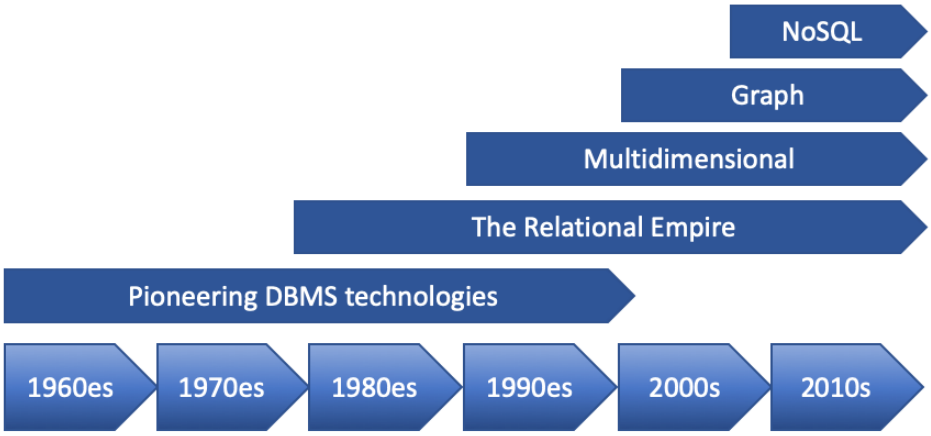

Ilustración 1: Evolución de las tecnologías de bases de datos[1]

- La primera generación se establece a mediados de los años 60 y principios de los 70 [Berners-Lee01]. Estos sistemas estaban basados en modelos de datos en red y jerárquicos, almacenando los datos como registros enlazados. El procedimiento empleado a la hora de realizar operaciones en este modelo consistía en navegar a través de registros relacionados mediante punteros hasta encontrar el dato solicitado. El principal inconveniente era la complejidad que presentaba a la hora de modificar y escribir en programas de aplicación, incluso para consultas sencillas (conceptos como encontrar y buscar no estaban contemplados aún). Algunos

[1] http://graphdatamodeling.com/GraphDataModeling/History.html

ejemplos de esta generación son IDS (en red), IMS (estructura jerárquica de ordenación de los datos, desarrollado por IBM) e IDMS (en red).

- La segunda generación surgió debido, entre otras cosas, a la necesidad de la operación de búsqueda. La tecnología de almacenamiento de datos de esta época está basada en el modelo de datos relacionales propuesto por Edgar Codd. La idea era representar los datos estructurados como tuplas que se agrupan en relaciones indexadas por una clave que identifica a cada registro unívocamente, facilitando así la búsqueda de registros. Debido a este descubrimiento, SQL (*Structured Query Laguaje*) se convertiría el lenguaje estándar para definir, manipular y consultar información.
- A comienzo de los años 80, los programadores que manejaban estructuras complejas de datos y operaciones específicas se encontraban con dos problemas: las limitaciones en el modelado del modelo relacional de datos y los contratiempos ocasionados al mapear los objetos en tablas de una base de datos relacional (término conocido como *object-relacional impedance mismatch*). Estos problemas llevaron a la aparición de los sistemas de base de datos orientadas a objetos (OIDs). Sin embargo, estos sistemas no se impusieron completamente debido a la enorme inversión realizada en los sistemas relacionales. Por otro lado, los defensores de las bases de datos relacionales intentaron extender el modelo relacional incorporando claves orientadas a objetos. Sistemas de bases de datos relacionales como Oracle y DB2 adoptaron dichas extensiones almacenando datos como relaciones planas en vez de objetos reales, pero el mapeo y los *joins* se realizan automáticamente.
- Desde comienzos de los años 2000, los avances en tecnologías web, redes sociales, dispositivos móviles e Internet de las cosas (IoT) llevaron a la aparición repentina de datos estructurados, semiestructurados y desestructurados generados por aplicaciones de alcance global. Estas aplicaciones requieren de bases de datos que dispongan de escalabilidad horizontal para poder adaptarse a la gran cantidad de datos, alta disponibilidad y tolerancia a fallos, confiabilidad en las transacciones y mantenibilidad del esquema de la base de datos para reducir los costes asociados a la evolución de ese esquema. Lograr esos requerimientos con los sistemas tradicionales de bases de datos es prácticamente inviable: el esquema relacional hace que la evolución de las bases de datos sea muy costosa y escalar estos sistemas se convierte en una tarea compleja y muy cara que deriva además en problemas de rendimiento y disponibilidad.

Los requerimientos antes descritos pueden ser logrados a costa de sacrificar aquél que no es necesario desde el punto de vista de la aplicación [Cattell11]. Esto dio lugar a la cuarta generación de tecnologías de almacenamiento de datos y a una nueva tendencia de bases de datos no relacionales, conocidos como almacenes NoSQL, cuyo propósito es satisfacer los requerimientos de escalabilidad y alta disponibilidad de las aplicaciones antes mencionadas.

- La quinta generación tuvo lugar ya entrados los años 2000 con una categoría emergente de bases de datos que le daba especial importancia a los requisitos de alta escalabilidad y fiabilidad de las aplicaciones modernas OLTP (procesamiento de transacciones en línea), las cuáles manejan grandes volúmenes de datos más allá de un solo centro de datos. Esto se logra mediante una nueva arquitectura donde la escalabilidad y el rendimiento son mejorados, conociéndose como almacenes de datos NewSQL.

Por último, cabe mencionar que se conocen más de 200 arquitecturas diferentes NoSQL y ese número sigue creciendo. Esto supone un gran desafío para los diseñadores y arquitectos de aplicaciones que de desean migrar de las gestiones de datos empresariales actuales a aquellas de gran escala.

2.2 Modelos de datos NoSQL

Los almacenes de datos NoSQL se pueden categorizar en cuatro tipos según cómo se almacenan esos datos:

- Clave-valor.
- Columnar.
- Documental.
- En grafo

Cada uno de estos modelos presenta unas características que hacen que se adapte mejor a unos escenarios de aplicación en concreto.

2.2.1 Bases de datos clave-valor

Este es el modelo de almacenamiento NoSQL más popular y simple, en el cual los datos se representan como pares (clave, valor) de manera eficiente, altamente escalable, con estructuras con búsqueda por clave como tablas hash distribuidas (DHTs) y árboles *Log Structured-Merge* (LMS-*trees*) [Atzeni17], un método de organización de archivos particular. Debido a la estructura sin esquema de los valores almacenados, la indexación y la búsqueda basada en estos valores no está soportado por el sistema. Cualquier escenario que requiera realizar consultas sobre la estructura interna del valor de los datos debe realizarse por la aplicación cliente. Por tanto, los almacenes clave-valor son apropiados únicamente para aplicaciones que utilizan solamente una única clave para acceder a los datos. Por ejemplo, un carrito de compras online, la configuración y el perfil del usuario y la información de la sesión en una página web. Este modelo de datos tan simple destaca por la facilidad a la hora de particionar los datos y las consultas eficientes de datos, lo que se refleja en la alta escalabilidad de esta arquitectura [Gutierrez08].

Muchos sistemas soportan funcionalidades adicionales como, por ejemplo, indexar y realizar consultas sobre el contenido de los valores de tipos de datos específicos. *Redis* y *Aerospike* soportan el tipo de datos de lista, permitiendo realizar operaciones atómicas sobre valores de la lista (introducir valores en una de estas sin tener que remplazar el valor completo, por ejemplo). IBM *Spinnaker, HyperDex* y *Yahoo Pnuts* soportan tipos de datos tabulares con un esquema flexible o fijo, en el cuál cada fila de la tabla está identificada únicamente por una clave. Por último, Oracle NoSQL soporta operaciones multiclave.

Los almacenes clave-valor se pueden agrupar según la forma de persistir los datos: en memoria (proporcionando un acceso extremadamente rápido a la información), persistentes (con una alta disponibilidad del acceso a los datos) e híbridos (en primer lugar, manteniendo los datos en memoria y después persistiéndolos cuando se satisfacen ciertas condiciones).

Las operaciones típicas de proporciona este modelo son:

- *get*(clave): recupera un valor (o una lista de valores con versiones diferentes) asociados con la clave.
- *put*(clave, valor): añade el par clave-valor a la base de datos sólo si no hay una clave igual ya. De lo contrario, el valor almacenado se actualiza con una nueva versión. Cabe destacar que para actualizar una mínima parte del valor almacenado se necesita reemplazar el valor entero.
- delete(clave): elimina la clave y sus valores asociados.

Los detalles sobre las operaciones antes mencionadas dependen de factores como la consistencia del modelo y su indexación. Estas operaciones de clave única no permiten manipular varios valores en una única operación. Estas operaciones pueden ser realizadas mediante el protocolo de comunicación REST.

Para finalizar con esta arquitectura, cabe enumerar los sistemas más representativos de la misma: *Redis, Riak KV, Oracle NoSQL, Hyperdex, Yahoo Pnuts, Oracle Berkeley DB y Project Voldemort*, en orden de mayor a menor relevancia según el ranking DB-Engines[2].

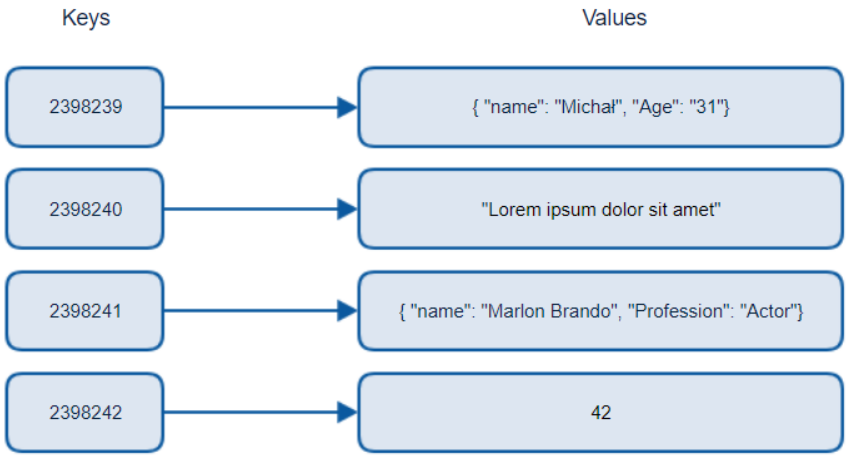

Ilustración 2: Base de datos clave-valor

2.2.2 Bases de datos columnares

Este modelo, también conocido como *column-family store*, fue inspirado por *Google Bigtable*, en el cual los datos están representados en un formato tabular de filas y un número fijo de columnas (*column-families*). Una *column-family* está formada por un número aleatorio de columnas relacionadas unas con las otras y accedidas conjuntamente con frecuencia. Esto justifica el porqué de que los datos de este modelo se almacenen físicamente por *column-family* en vez de por fila. El esquema de una *column-family* es flexible ya que sus columnas pueden ser añadidas o eliminadas en tiempo de ejecución. De esta forma, este modelo se trata de un almacén clave-valor extendido en el cual el valor está representado como una secuencia de pares (clave, valor).

[2] https://db-engines.com

Adicionalmente, este modelo normalmente permite que se pueda almacenar varias versiones de cada valor de la celda, indexados por marcas de tiempo. Algunas bases de datos de este tipo proporcionan agregados adicionales. Por ejemplo, *Apache Cassandra* permite anidar una *column-family* en otras *column-families* llamadas "*super column-families*".

Este modelo destaca a la hora de realizar lecturas en la base de datos, especialmente con grandes volúmenes de información. Sin embargo, a la hora de trabajar con aplicaciones transaccionales, donde los accesos son distintos en la mayor parte de los casos, los nuevos datos deben distribuirse a través de toda la base de datos. En ese caso, el modelo relacional tradicional sería incluso más rápido que este modelo [Avrilia12].

Los sistemas más representativos de este sistema son *Apache Cassandra, Apache Hbase, Hypertable y Google Cloud Bigtable*, en orden del ranking mencionado anteriormente.

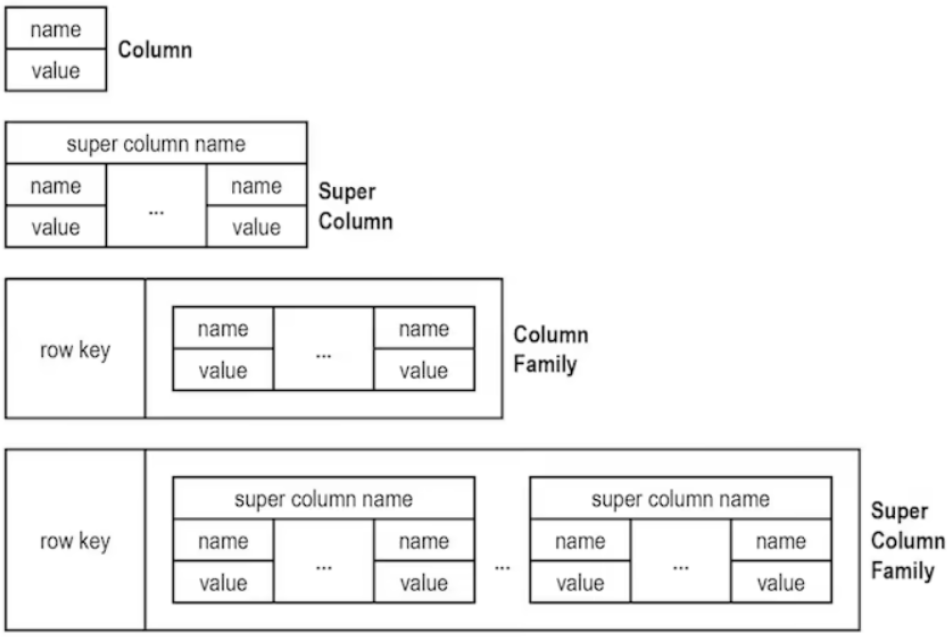

Ilustración 3: Base de datos columnar[3]

2.2.3 Bases de datos documentales

Este modelo se cataloga como un almacén clave-valor extendido en el cual el valor se representa como un documento codificado en formatos estándares semiestructurados como XML, JSON o BSON (JSON binario). Un documento tiene esquema flexible en el que se pueden añadir o eliminar sus atributos en tiempo de ejecución. A diferencia del contenido opaco de los valores en el modelo clave-valor, en este caso conocemos el formato de los documentos, soportando índices y funcionalidades de búsqueda basados en su nombre de atributo y valores.

Las bases de datos documentales son ideales para aquellas aplicaciones en las que los datos pueden ser representados en un formato de documento. Por ejemplo, una entrada de blog que incluya varios atributos anidados (etiquetas, cometarios, imágenes, vídeos...) puede ser

[3] https://blog.logrocket.com/nosql-wide-column-stores-demystified/

representado fácilmente en formato de documento. Este modelo está también pensado para las aplicaciones web, por dos motivos. En primer lugar, estas aplicaciones tienen una evolución continua de su esquema de datos y se pueden beneficiar del modelo de datos flexible de los almacenes documentales. En el caso del modelo relacional, añadir extensiones al esquema es caro y requiere la creación de nuevas tablas o la suma de nuevas columnas a las tablas existentes. En segundo lugar, las aplicaciones web modernas soportan modelos de datos como JSON con una estrecha integración con leguajes de programación populares como Python, Javascript y Ruby. Esto reduce extremadamente la adaptación de impedancias objeto-relacional entre esos lenguajes y las bases de datos documentales, de manera que las estructuras orientadas a objetos pueden ser mapeadas en documentos fácilmente.

Los sistemas más representativos en esta categoría son *MongoDB, Amazon DynamoDB, Couchbase, Apache CouchDB y ArangoDB* de mayor a menor relevancia en el ranking.

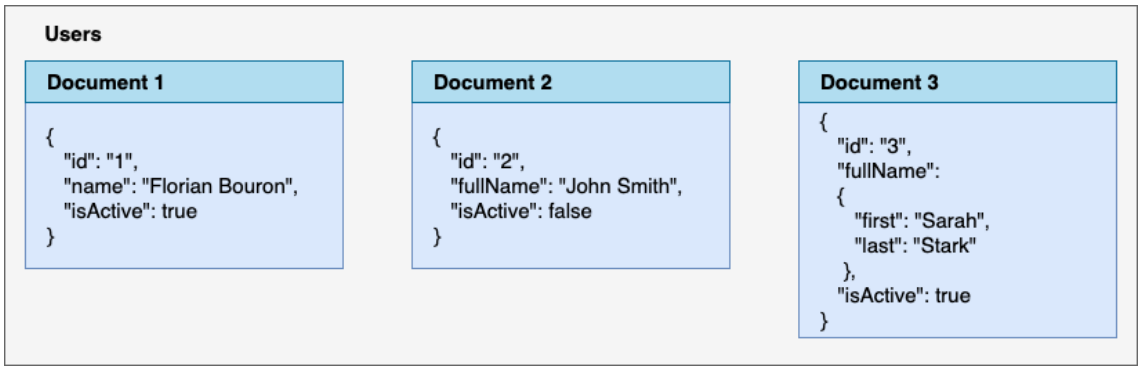

Ilustración 4: Base de datos documental[4]

2.2.4 Bases de datos en grafo

El creciente número de colecciones de datos orientadas a grafos conllevó la necesidad de recorrer relaciones entre entidades de forma eficiente. Este modelo está basado en la teoría de grafos tradicional, donde un grafo consiste en vértices que representan entidades y aristas que conforman las relaciones entre estas [Gutierrez08].

Existen diferentes estructuras de grafos. Por ejemplo, distinguimos entre grafos dirigidos (todas las relaciones son simétricas) y no dirigidos (cada arista "a" del vértice "o" al vértice "d" es una tupla dirigida), etiquetados (los vértices y aristas están etiquetados con valores escalares que representan sus roles en el dominio), grafos con atributos (los vértices y aristas contienen atributos clave-valor que representan sus propiedades), grafos anidados (cada vértice puede ser a su vez un grafo), etc.

En lo relativo a técnicas de almacenamiento, se pueden categorizar en bases de datos en grafo nativas y no nativas. El modelo lógico de un grafo no nativo está implementado sobre una base de datos diferente al modelo en grafo, como una de tipo documental o relacional, necesitando índices para navegar transversalmente entre grafos. Sin embargo, la búsqueda por

[4] https://aws.plainenglish.io/how-to-choose-the-right-database-31af8b0260d3

índices puede suponer un peor rendimiento, especialmente en búsquedas en profundidad. En el caso de las nativas, el almacenamiento se adapta según las características de los datos de los modelos del grafo. Por ejemplo, Neo4j presenta un procesamiento del grafo en tiempo real muy eficiente a través del acceso directo sin índices a los nodos adyacentes de un nodo.

Las principales ventajas que presenta frente al modelo relacional son: la rapidez en ejecutar consultas sobre conjuntos de datos asociativos, el mapeo más sencillo de la estructura de aplicaciones orientadas a objetos, una mejor escalabilidad y una mayor adaptación del esquema ante unos datos cambiantes.

Los sistemas más representativos de este modelo son *Neo4j y Titan*.

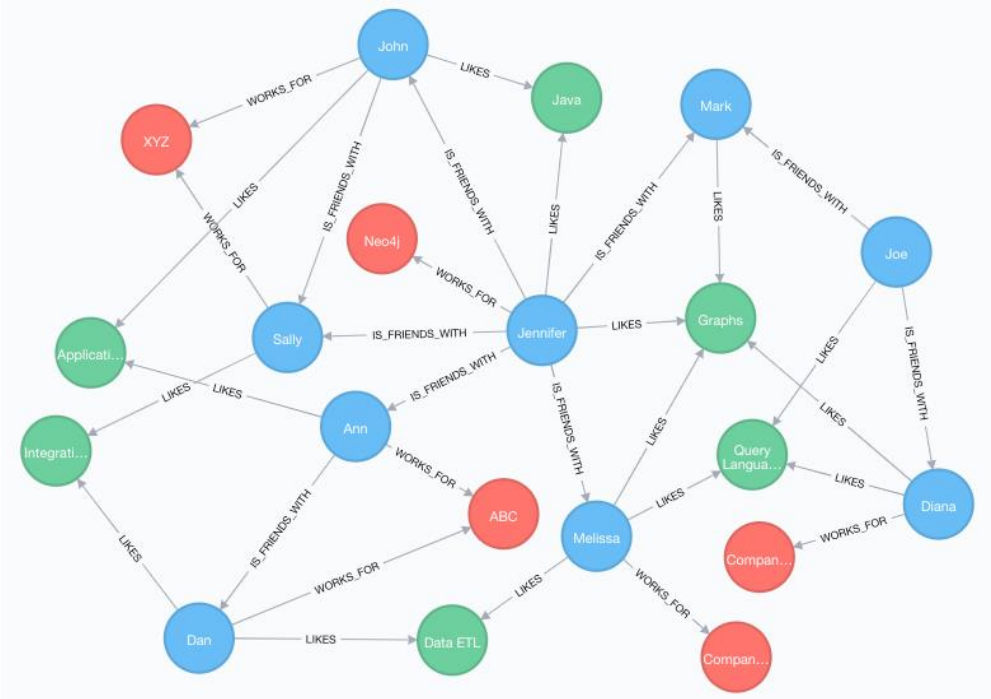

Ilustración 5: Base de datos en grafo[5]

2.3 Modelos de consistencia

Un modelo de consistencia determina el efecto de las operaciones concurrentes en datos compartidos desde el punto de vista de diferentes clientes del sistema. Esto determina el orden en el que se ejecutan las operaciones según sean lecturas individuales y escrituras en un sistema no transaccional o transacciones que comprenden lecturas y escrituras múltiples en un sistema transaccional [Kapali76]. A continuación, se presentarán varios modelos de consistencia presentes en las bases de datos NoSQL.

[5] https://neo4j.com/

2.3.1 Consistencia estricta o atómica

Este modelo dicta un ordenamiento total y en tiempo real de todas las operaciones en un sistema no transaccional en el cuál una operación de lectura lee el resultado de la actualización más reciente. Representan la forma más estricta de consistencia. Respecto a los datos replicados o particionados, estos modelos proporcionan una abstracción de una imagen única del sistema [Beeri90]. Esta abstracción simplifica la programación de los sistemas distribuidos y previene comportamientos anómalos de las aplicaciones, como lectura sucia, lectura no repetible y lectura fantasma. Sin embargo, esto necesita utilizar estrategias costosas y difícilmente escalables para confirmar las actualizaciones a través de las réplicas (vía protocolos de consenso como 2PC) y un control de concurrencia (como bloqueo estricto en dos fases (S2PL)). Estas estrategias tienen un impacto severo en la disponibilidad y el rendimiento, especialmente en redes con una extensión considerable. Un ejemplo de consistencia estricta sería el siguiente, donde la lectura de un dato x devuelve el valor correspondiente a la escritura más reciente sobre ese dato (suponiendo que existe un reloj de tiempo global):

Ilustración 6: Ejemplo de consistencia estricta

De forma similar, la consistencia secuencial impone un ordenamiento total de todas las operaciones, mostrando la apariencia de un sistema único. Sin embargo, es un modelo más débil de aislamiento transaccional comparado con la consistencia estricta, asegurando el mismo ordenamiento que la consistencia secuencial. Está implementado típicamente utilizando control de concurrencia multiversiones (MVCC), el cuál incrementa la concurrencia de las transacciones. Sin embargo, el ordenamiento monótono de las operaciones requiere utilizar un mecanismo secuencial global, que puede suponer un cuello de botella en redes amplias. Ejemplo de consistencia secuencial:

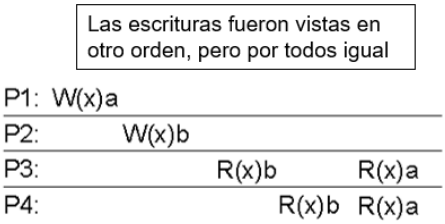

Ilustración 7: Ejemplo de consistencia secuencial

Debido a que la gestión de una imagen consistente de todos los datos en un sistema de almacenamiento distribuido es extremadamente costosa, algunos sistemas NoSQL presentan un soporte transaccional limitado introduciendo agregados atómicos. Los agregados permiten almacenar datos desnormalizados de manera conjunta en vez de tener datos normalizados dispersos. Una sola entidad puede estar representada por un agregado como un par clave-valor en almacenes clave-valor, una *column-family*, *super column-family*, toda la fila de una tabla en una base de datos *column-family* o un documento en un almacén documental.

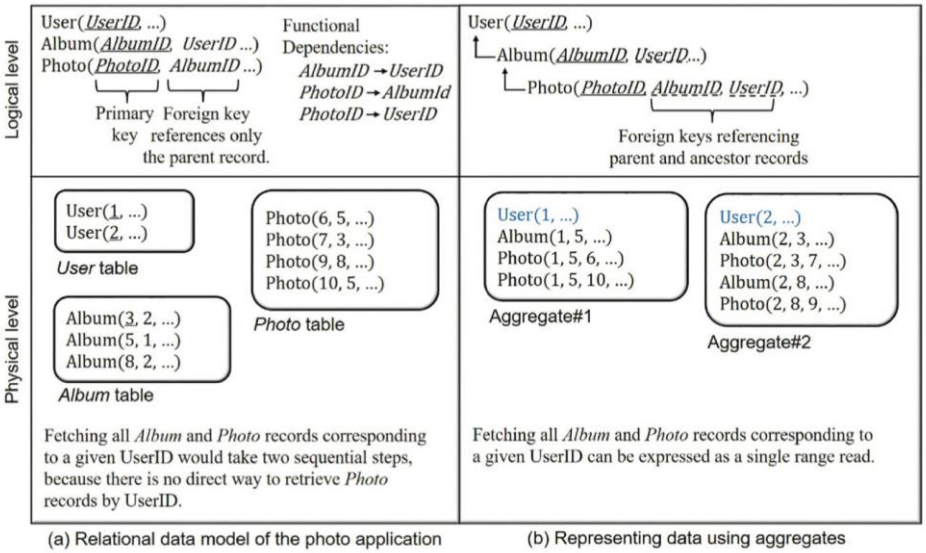

Ilustración 8: Ejemplo de uso de agregados

2.3.2 Consistencia eventual

La consistencia eventual no dicta ningún ordenamiento de las operaciones, pero asegura la convergencia gradual y eventual de las réplicas de valores idénticos tras recibir el mismo conjunto de actualizaciones propagadas asíncronamente. Los resultados de las consultas en un sistema con consistencia eventual pueden no devolver las actualizaciones más recientes, debido a que conlleva un tiempo actualizar las réplicas a través de un clúster. En la consistencia estricta, los datos se envían a cada réplica en el momento en que la consulta se realiza, causando un retardo debido a que la respuesta a cada consulta debe esperar mientras que se actualizan las réplicas.

Cuando no se produce un fallo, factores como el número de réplicas en el esquema de replicación, la carga del sistema y el retardo en las comunicaciones determinan la longitud de la ventana de inconsistencia. Este concepto de consistencia satisface a aquellas aplicaciones para las cuáles la alta disponibilidad de la solicitud de datos es tan crítica que incluso un pequeño impacto en la misma supone el descontento del usuario y pérdida de ingresos (por ejemplo, medios de comunicación). Por otra parte, la consistencia eventual permite comportamientos anómalos de las aplicaciones, lo que aumenta la complejidad del diseño de aplicaciones, *testing* y depuración. También aparecen conflictos al escribir en el mismo dato. Esto requiere el uso de detección de conflictos y mecanismos de resolución para prevenir la divergencia permanente de las réplicas aplicando una secuencia idéntica de actualizaciones en estas. Se utilizan cuatro estrategias:

- Reconciliación semántica impulsada por el cliente. Es utilizado en *Voldemort*.
- Reconciliación sintáctica basada en marcas de tiempo, como *last-write-wins* (la última escritura gana), donde cada réplica modificada tiene asignada una marca de tiempo. El conflicto es resuelto al seleccionar aquella con la mejor marca de tiempo (por ejemplo, la más reciente y precisa). Es utilizado por *Apache Cassandra, Amazon Dynamo, COPS y CouchDB.*

- Reconciliación sintáctica por relojes vectoriales, donde un ordenamiento parcial a través de las operaciones de lectura de cada dato se establece capturando su relación potencial *happens-before* (ocurre antes). Por tanto, el sistema se libera de la carga de la sincronización del reloj en todos los servidores. *Amazon Dynamo y Riak KV* utilizan esta estrategia.
- Reconciliación sintáctica por tipos de datos replicados conmutativos (CRDT), donde la convergencia de réplicas de ciertas estructuras de datos se garantiza sin considerar el orden de ejecución de las operaciones. Bases de datos NoSQL como *Walter y Riak KV* utilizan esta estrategia.

Ninguna de las estrategias de control de concurrencia anteriores puede satisfacer las necesidades de las aplicaciones OLTP, debido a que estas no pueden manejar el problema de la sección crítica. Por ejemplo, dos transacciones concurrentes pueden modificar el mismo dato al mismo tiempo. La convergencia de las réplicas puede asegurarse también utilizando estrategias de recuperación. Algunas de ellas son:

- Recuperación de lectura. El coordinador de una operación de lectura lo envía inicialmente a todas las réplicas correspondientes. Espera hasta que recibe las respuestas de un número configurable de réplicas. A continuación, determina la última versión de los valores devueltos y se lo envía al usuario. Además, las réplicas obsoletas son actualizadas asíncronamente.
- Traspaso insinuado (*hinted-handoff*). Asegura que todas las peticiones de lectura son aplicadas a sus réplicas objetivo. Supone que el coordinador de una operación de escritura notifica el fallo temporal de una réplica. A continuación, almacena todas las actualizaciones relacionadas con la réplica en una tabla local *"hinted-handoff"*. Recuperando la réplica, todas las actualizaciones son ejecutadas en esta. Sin embargo, en el caso de un fallo grave en la memoria, la tabla podría perderse.
- Anti-entropía, utilizando árboles de Merkle, en los que cada miembro de un grupo de réplicas periódicamente crea un árbol de Merkle y se lo envía a otros miembros. El árbol recibido por un nodo réplica puede ser comparado de forma eficiente con su propio árbol para determinar porciones de datos asíncronamente para enviárselo a réplicas obsoletas. El principal inconveniente de esta estrategia es que requiere ancho de banda para transferir árboles a través de la red.

La mayoría de las bases de datos NoSQL, *como Apache Cassandre, Amazon Dynamo, Riak KV, Voldemort, Couch DB, Couchbase Server, MongoDB y Neo4j*, preservan este modelo de consistencia.

2.3.3 Consistencia causal regular

La inviabilidad de los relojes globales en los sistemas distribuidos motivó a los sistemas a considerar operaciones ordenadas parcialmente. Respectivamente, la consistencia causal impone un orden parcial (llamada relación *happens-before*) entre relaciones causalmente dependientes. De forma más precisa, una operación *b* es causalmente dependiente de una operación *a* si una de las siguientes tres condiciones tiene lugar: (1) *a* y *b* son emitidas por el mismo cliente y *b* ocurre después de *a*, *b* es una operación de lectura que devuelve el valor escrito por *a*, y estas operaciones son transitivamente dependientes. Intuitivamente, la

consistencia causal garantiza que cuando se confirma una operación de escritura en una réplica, todas las operaciones de lectura que la precedan causalmente ya han sido confirmadas en la réplica [Brzezinski04].

Debido a que esta consistencia no garantiza ninguna ordenación entre operaciones concurrentes causalmente independientes, la eficiencia de la implementación incrementa, ya que no hay necesidad de determinar un punto de serialización entre operaciones de lectura concurrentes no relacionadas por lo que pueden ser replicadas en cualquier orden. Sin embargo, hay un conflicto cuando el mismo dato es actualizado por operaciones concurrentes. Esto es indeseable, ya que el conflicto podría dar lugar a una divergencia continua de las réplicas. Adicionalmente, las operaciones de lectura pueden no siempre acceder a la última versión de datos leídos. Los conflictos de lectura son manejados normalmente utilizando las estrategias anteriormente descritas.

La consistencia causal se implementa típicamente mediante los siguientes tres pasos: (1) confirmando las actualizaciones en las réplicas locales correspondientes, (2) propagación asíncrona de las actualizaciones a réplicas remotas, y (3) realizando comprobaciones de la dependencia causal para determinar cuándo cada actualización puede ser aplicada en una réplica correspondiente. Esto conlleva a la necesidad de metadatos de dependencia causal (relojes de tiempo real o un híbrido de relojes físicos y lógicos) que son propagados a través de los distintos mensajes o con las actualizaciones.

La consistencia causal es una de las nociones más fuertes de consistencia que son compatibles con baja latencia, alta disponibilidad y tolerancia a particionamiento. También fuerza las siguientes garantías de sesión, en las que una sesión indica la secuencia de operaciones emitidas por un cliente dado en la base de datos:

- Lecturas monotónicas (MR), en las que operaciones de lectura sucesivas emitidas en un dato observan un orden no decreciente de las versiones del ítem. Por ejemplo, considera una red social donde las publicaciones de cada usuario son escritas para el perfil del usuario y el de sus amigos. De acuerdo con MR, cuando un usuario observa una publicación en el perfil del usuario, entonces esa operación de lectura del usuario incluirá la misma publicación (a menos que la publicación haya sido eliminada).

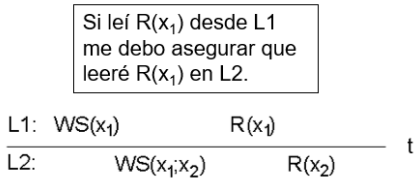

Ilustración 9: Ejemplo de lectura monotónica

- Escrituras monotónicas (MWs), en las que operaciones de escritura sucesivas emitidas en un dato tienen efecto en el mismo orden. De acuerdo con MWs, cuando un usuario de una red social realiza dos escrituras consecutivas, cualquier operación de lectura sucesiva que contenga la segunda publicación también contiene la primera.

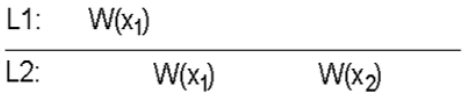

Ilustración 10: Ejemplo de escritura monotónica

- Lea sus escrituras (RMWs), en las que las operaciones de lectura emitidas en un dato siempre ven el efecto de la operación de escritura más reciente emitida en el mismo ítem. De acuerdo con RMW, cuando un usuario de una red social lee el muro de otro, observa su última publicación (a menos que la haya eliminado).

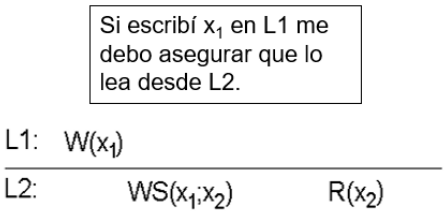

Ilustración 11: Ejemplo de "lea sus escrituras"

- Escrituras siguen a lecturas (WFR), donde las operaciones de escritura emitidas en un dato tienen efecto en versiones del ítem que son iguales o más nuevas que versiones vistas precediendo operaciones de lectura emitidas en el mismo ítem. De acuerdo con WFRs, cuando el usuario de una red social responde a un post, cualquier operación de lectura sucesiva que contiene la respuesta también contiene la publicación original.

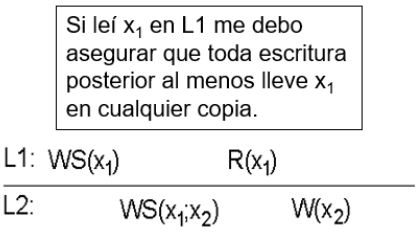

Ilustración 12: Ejemplo de "escrituras siguen a lecturas"

Ejemplo de secuencia con un almacenamiento causalmente consistente, pero no con uno del tipo secuencialmente consistente o superior:

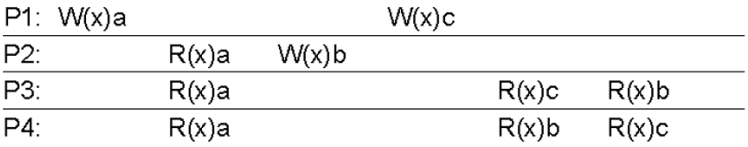

Ilustración 13: Ejemplo de consistencia causal

La consistencia causal está preservada en una gran cantidad de bases de datos académicas geo-replicadas, las cuales utilizan varias estrategias para representar metadatos de relaciones causales y seguimiento de dependencia. Estas estrategias hacen la implementación de la consistencia causal más costosa que la consistencia eventual, debido a que no hay base de datos comercial que la preserve.

2.3.4 Consistencia por objeto mediante línea de tiempo

La consistencia secuencial por objeto asegura un ordenamiento total de todas las operaciones en cada ítem de datos junto con su orden emitido por cada cliente. Fue utilizado inicialmente para el diseño de *Yahoo! Pnuts*, el sistema de almacenamiento utilizado por las aplicaciones web de *Yahoo*!. Evita utilizar la serialidad de las transacciones (en múltiples ítems de datos) debido a la tendencia de las aplicaciones web de emitir operaciones en ítems de datos únicos. Utilizando esta consistencia, actualizaciones concurrentes en el mismo ítem de datos son serializadas en su correspondiente réplica primaria. Esta estrategia puede realizar este ordenamiento asignando monótonamente números incrementales a las actualizaciones. A continuación, las actualizaciones (junto a sus números de secuencia asociados) son replicados de manera asíncrona en todas las réplicas correspondientes. Estos números de secuencia son necesarios para asegurar la distribución de las actualizaciones en orden. En este modelo, todas las operaciones de escritura y lectura fuertemente consistentes en un ítem de datos son reenviados a la réplica máster correspondiente. Sin embargo, las operaciones de lectura consistentes cronológicas en un dato son respondidas localmente [Jiaqing14]. Esto podría conllevar la lectura de datos obsoletos. Sin embargo, la operación de lectura de un cliente nunca devuelve una nueva versión antes de una antigua. Algunas bases de datos NoSQL, como *Yahoo Pnuts* preservan esta consistencia.

2.3.5 Aislamiento de instantáneas paralelas (PSI)

Esta noción de consistencia suaviza el aislamiento de instantáneas (SI) sobre bases de datos completamente geo-replicadas. Más precisamente, PSI no impone un ordenamiento global de las confirmaciones de las transacciones. En su lugar, fuerza SI localmente en las transacciones ejecutadas en cada centro de datos donde se utiliza un número de secuencia local. Esta relajación permite la propagación asíncrona de actualizaciones a través de centros de datos y mejora la escalabilidad y rendimiento del sistema. Algunas bases de datos como Walter imponen PSI en sus transacciones.

La siguiente tabla resume las características de los modelos de consistencia presentados. Hay que tener en cuenta que las aplicaciones modernas interactúan con una mezcla de datos fuerte y débilmente consistentes. Una estrategia es almacenar datos en una única base de datos que proporciona garantías de consistencia variables, donde el programador puede seleccionar un nivel de consistencia apropiado para cada operación. Esta estrategia es utilizada por *Oracle NoSQL, MongoDB, Yahoo Pnuts, Apache Cassandra* y *Amazon Dynamo*. Otra estrategia es almacenar datos en una mezcla de almacenes de datos con garantías heterogéneas de consistencia.

Characteristics		Suitable applications	Suitable NoSQL store data models
Strong consistency	– Dictating a total, real-time ordering of all operations/transactions – Providing a single, consistent image of the whole data – Severe impact on availability and performance (especially in wide-area networks) – Using a synchronous primary/ update-anywhere replication strategy	Scenarios requiring ACID reliability, such as financial services	It can be practically preserved over one or more *aggregates* (along with their corresponding replicas). An *aggregate* can be a key-value pair in *key-value stores*, a column-family, super column-family, the whole row of a table in *wide-column stores*, or a document in *document stores*.
Causal consistency	– Enforcing a partial ordering (called happens-before relation) among causally dependent operations – The efficiency of implementation is more than strong consistency and less than the eventual one. – The possibility of conflicting writes – Read operations may not always access the latest versions of read data items. – Using an asynchronous primary/ update-anywhere replication strategy	Online human-facing services, such as commenting services in social networks	It can be preserved in all data models while enforcing in a lot of academic NoSQL stores. However, owing to its communication overhead (of dependency tracking), no industrial NoSQL store enforces this model.
Per-object Timeline consistency	– Ensuring a total ordering of all operations on each data item along with respecting their order as issued by each client – Read operations may access stale data items. – A client's read operation never returns the new version of a data item before an old one. – Using an asynchronous primary replication strategy	Online human-facing services, such as the update of photos/albums	It can be practically preserved over an *aggregate* in a key-value, wide-column or document stores. It is enforced in some industrial NoSQL stores, such as Yahoo Pnuts (Cooper et al. 2008).
Parallel snapshot isolation	– Enforcing snapshot isolation on transactions executed in each datacenter – Preserving the causal ordering of transactions executed across datacenters – Using an asynchronous primary/ update-anywhere replication strategy	Similar to causal consistency	Preserved in some academic NoSQL stores, such as Walter (Sovran et al. 2011)
Eventual consistency	– It does not dictate any ordering of operations. – It ensures the gradual and eventual convergence of replicas to identical values after receiving the same set of asynchronously propagated updates. – It allows anomalous behaviors of applications. – It suffers from conflicting write operations on the same data item. – It uses an asynchronous primary/ update-anywhere replication strategy.	It suffices for many services in web applications for whom the high availability of data requests is so critical that even a tiny impact on it causes user dissatisfaction and loss of revenue.	It can be preserved in all data models. It is enforced in a lot of NoSQL stores.

Ilustración 14: Características de los modelos de consistencia [Gutierrez08]

2.4 La fragmentación

La fragmentación de los datos incluye soluciones donde los datos en una base de datos son divididos en varios fragmentos inconexos y propagados en diferentes nodos de almacenamiento. Esto se puede lograr mediante fragmentación horizontal y vertical. La fragmentación horizontal, típicamente utilizado en bases de datos NoSQL, divide los datos a nivel de fila (por ejemplo, filas en una *wide-column* o documentos en una colección) en particiones disjuntas. En la fragmentación vertical, la descomposición se realiza por filas, es decir, por registros.

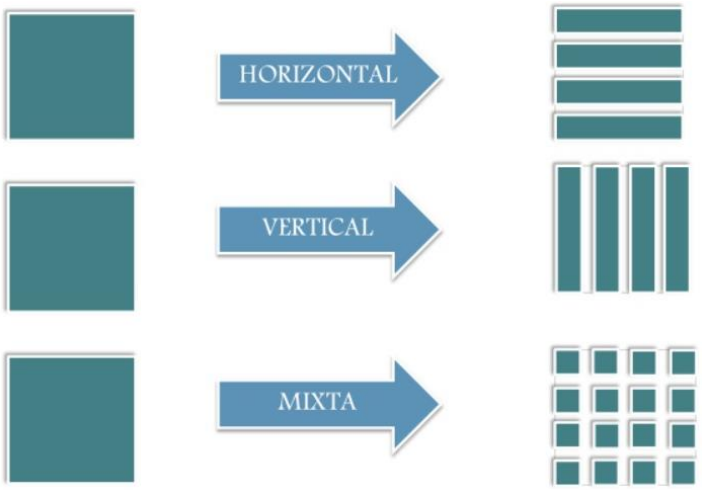

Ilustración 15: Fragmentación horizontal, vertical y mixta[6]

La fragmentación de los datos tiene algunas ventajas. En primer lugar, mejora la escalabilidad de un sistema debido a que aborda las situaciones en que grandes volúmenes de datos y altas tasas de peticiones abruman la base de datos y la capacidad de procesamiento de cualquier servidor único. En segundo lugar, mejora el rendimiento de un sistema al incrementar el grado de procesamiento concurrente en múltiples particiones. Finalmente, funciona bien para datos dispersados geográficamente, cuyas lecturas y escrituras ocurren principalmente dentro de una localización geográfica.

Existen múltiples estudios dedicados a mejorar las estrategias de fragmentación utilizadas por las bases de datos NoSQL. Sin embargo, todavía existen algunos retos que afectan la eficiencia de una partición con respecto al rendimiento y la latencia de las consultas del usuario. Primeramente, el número de solicitudes multi-partición (por ejemplo, el procesamiento de una consulta necesita contactar varias particiones) debería ser reducido. Eso reduce la cantidad de datos transmitidos en una red durante el procesamiento de la consulta. En segundo lugar, la distribución del procesamiento y carga de la base de datos debería ser uniforme con respecto a la capacidad de los nodos (potencia del procesador, velocidad del disco y capacidad de la red). En tercer lugar, la cantidad de datos transferida durante eventos de repartición (por ejemplo, al añadir un nodo o eliminarlo) debería reducirse. Finalmente, el almacenamiento, recuperación y manipulación del mapeo entre fragmentos y nodos de la base de datos debería ser eficiente.

Los modelos de datos tienen un efecto significativo al seleccionar la estrategia de fragmentación. Las bases de datos en grafo utilizan estrategias de fragmentación orientadas transversalmente, mientras que otros modelos de datos utilizan estrategias orientadas a la clave. Las estrategias de fragmentación mediante *hashing* orientadas a la clave son aplicables para aplicaciones, como un carrito de compras y un catálogo de productos, donde los datos están representados mediante pares clave-valor o documentos y cada ítem de datos puede ser accedido independientemente. Las estrategias orientadas trasversalmente, cuyos nodos menos conectados entre fragmentos y nodos altamente relacionados en grupo, son apropiadas para aplicaciones en cuyos objetos están vinculados como grafos y relaciones entre amigos, compras

[6] https://es.educaplay.com/recursos-educativos/3368406-bases_de_datos_distribuidas.html

de productos, y valoraciones son atravesados rápidamente. Ambos tipos de fragmentación, vertical y horizontal pueden ser proporcionados por las bases de datos *wide-column*. Para otros tipos de bases de datos, solo se estudia por ahora la fragmentación horizontal.

2.4.1 Fragmentación estática orientada a la clave

El objetivo de este tipo de fragmentación es equilibrar la utilización del espacio estático, mientras que los datos dinámicos y la carga de trabajo de las consultas son completamente desatendidos. Algunas de las estrategias empleadas son las siguientes:

- **Basadas en rango**

  Los ítems de datos se agrupan de acuerdo con los intervalos contiguos de claves, como el rango de un identificador único. Una ventaja es que las consultas de rango en pequeños intervalos son manejadas eficientemente ya que implican comunicarse con un único nodo o unos pocos nodos. En otras palabras, el número de particiones implicadas se reduce. Este tipo de particionamiento es muy útil para aplicaciones como servidores web y juegos online, los cuáles involucran estructuras de datos distribuidas que preservan el orden. Sin embargo, existen algunos inconvenientes. En primer lugar, las claves populares provocan que la carga de trabajo de las consultas sea mayor en algunos nodos más que en otros. En segundo lugar, la distribución heterogénea de inserciones de claves de datos provoca desequilibrios en la carga de datos de forma que el almacenamiento de los nodos queda desbalanceado. Finalmente, es necesario mantener una tabla de búsqueda central o directorio para almacenar el mapeo de las particiones de los nodos de almacenamiento [Petrov07]. Esta estrategia es utilizada por algunas bases de datos NoSQL como Google Bigtable, Apache Hbase, MongoDB, Yahoo Pnuts e IBM Spinnaker.

- ***Hashing* simple**

  Las claves de los ítems de datos son troceados de forma aleatoria a sus nodos a través de esquemas de *hashing* simples, como *hashing* por módulo, donde:

  Nº_de_nodo = Clave mod Nº_de_nodos

  Con este método, añadir o eliminar un nodo requiere la redistribución de una gran cantidad de ítems de datos a medida que el hash de las claves se reorganiza.

- ***Hashing* consistente**

  Este método considera el ámbito de una función hash como un anillo donde el ID de los nodos y las claves de los ítems de datos son distribuidos aleatoriamente a sus posiciones. El nodo que almacena un ítem de datos es el primer nodo encontrado al recorrer en el sentido de las agujas del reloj desde la posición del ítem de datos en el anillo. Por tanto, un nodo con posición p es responsable de un conjunto de ítems de datos cuyas claves hash corresponden a una parte del anillo entre *p*.predecesor y *p*. Cuando un nuevo nodo se añade y su clave hash corresponde a la posición q, el antiguo arco correspondiente a su sucesor inmediato se divide en dos arcos nuevos adyacentes entre *q*.predesor y *q*, al igual que *q* y *q*.sucesor. En el caso de eliminar o fallar un nodo, los antiguos arcos correspondientes al nodo y su sucesor se fusionan entre ellos.

De esta forma, en contraste con el *hashing* simple, añadir o eliminar un nodo únicamente implica redistribuir O(1/N) fracciones de ítems de datos que se almacenan en el sucesor del nodo, donde N es el número de nodos existentes. Esto significa que el *hashing* consistente escala mucho mejor que el *hashing* simple. Sin embargo, existen algunos inconvenientes. En primer lugar, todavía sufre de una pobre localidad de datos. También, para una distribución uniforme de ítems de datos en el espacio de claves, hay un desequilibrio de orden *O(log N)* entre *N* nodos en términos de ítems de datos almacenados y carga de trabajo de las consultas. En tercer lugar, la heterogeneidad de rendimiento de los nodos no se tiene en cuenta, lo que puede ocasionar desequilibrio en datos y carga de trabajo de las consultas. Finalmente, al añadir o eliminar un nodo, la redistribución de los datos puede abrumar la capacidad de carga de su sucesor.
Variantes del *hashing* consistente son utilizados comúnmente en DHTs, en los que los datos son almacenados y buscados de una forma totalmente descentralizada.

- ***Hashing* Hiperespacial**

  Las estrategias de particionamiento anteriores distribuyen los datos en función de la única dimensión de las claves de los ítems de datos. Estas estrategias resultan ineficientes en consultas a propiedades distintas a la clave primaria. Este problema puede ser abordado particionando en dimensiones múltiples. Como tal, el *hashing* hiperespacial es una extensión del *hashing* consistente, en el que múltiples atributos (en lugar de únicamente el atributo de clave) se tienen en cuenta para el mapeo de ítems de datos a nodos. Más específicamente, un ítem de datos con un conjunto de atributos se mapea en un espacio multidimensional (conocido como hiperespacio) donde cada dimensión se define por un atributo del ítem de datos. De acuerdo con esto, la posición del ítem de datos en el hiperespacio se determina a través del *hashing* del valor de cada atributo a lo largo de su dimensión correspondiente [Chen13].
  Un coordinador divide el hiperespacio entre zonas disjuntas; cada zona se asigna a un nodo virtual. Por tanto, un conjunto de pares zona-nodo, llamado mapa hiperespacial, se mantiene para cada hiperespacio y se distribuye a clientes y servidores que lo utilizan para la inserción, eliminación y búsqueda de ítems de datos. Este mapa puede cambiar añadiendo o eliminando nodos. Mediante este razonamiento geométrico, añadir más términos de búsqueda a una consulta limita el espacio de búsqueda a un conjunto de nodos más pequeño, incrementando la eficiencia de la consulta.
  Sin embargo, el volumen del hiperespacio aumenta exponencialmente con el número de atributos (o dimensiones); por tanto, su cobertura podría no ser factible incluso para grandes centros de datos.

2.4.2 Fragmentación orientada a la clave consciente de la carga de trabajo

En las estrategias de fragmentación anteriormente mencionadas, las consultas dinámicas y carga de trabajo no se tienen en cuenta. Por ejemplo, la función hash estática utilizada en el *hashing* consistente no puede abordar los puntos críticos (por ejemplo, nodos que están altamente cargados) causados por la distribución no uniforme de las consultas del mundo real.

Un particionamiento consciente de la carga de trabajo identifica los puntos críticos y a continuación ajusta dinámicamente la distribución de la carga. Este ajuste dinámico debe garantizar que no se produce el fenómeno ping-pong o la recolocación continua de un ítem de datos a través de los nodos [Petrov07]. Las siguientes estrategias son empleadas en este aspecto:

- **Migración de nodos virtuales**

  Esta estrategia asume que cada nodo físico puede albergar múltiples nodos virtuales de acuerdo con su capacidad y distribución de las consultas. Esto alivia los puntos críticos detectados mediante la reasignación de algunos de sus nodos virtuales a nodos ligeros. El sistema mantiene un conjunto de nodos de directorio centralizados. Estos reciben periódicamente la información de carga de diferentes nodos y, basado en esto, asignan nodos virtuales a servidores físicos.

- **Equilibrio de ítems**

  Mediante la estrategia propuesta por Karger y Ruhl, se realiza una redistribución de la carga entre nodos mediante la transferencia de ítems de datos entre estos. Algunos módulos como DBalancer pueden instalarse sobre una base de datos NoSQL, alimentados por una serie de particionadores basados en rango que soportan métodos de balanceo de carga propuestos por Karger y Ruhl.

- **Colocación de datos mediante autoajuste**

  Esta estrategia tiene como objetivo maximizar la localidad de patrones de acceso de datos cambiantes dinámicamente en una base de datos NoSQL clave-valor mediante la ubicación de datos más cercana a los clientes. Cada nodo rastrea e identifica sus ítems de datos críticos (que tienen el mayor número de solicitudes de escrituras/lecturas) utilizando un algoritmo de análisis de flujo. A continuación, se redistribuyen las réplicas de los puntos críticos detectados en nodos apropiados con respecto a su capacidad. La búsqueda de datos rápida se preserva utilizando una aproximación híbrida de un servicio de directorio altamente eficiente replicado en cada nodo y *hashing* consistente. El primero se utiliza para codificar la ubicación de ítems de puntos críticos reparticionados y el segundo para definir la localización de los no críticos. Esta estrategia se integró en el almacén clave-valor Infinispan; el rendimiento reportado fue seis veces mejor que el original.

2.4.3 Fragmentación orientada transversalmente

Teniendo en cuenta la alta conectividad de los datos en una base de datos en grafo, el particionamiento de los datos puede generar algunas dependencias entre particiones que, a su vez, incrementen la latencia de la red al atravesar particiones del grafo. Como resultado, el tiempo de ejecución al atravesar el grafo se incrementa. Un fragmentador de grafos debe tener como objetivos los siguientes puntos:

- Minimizar el número de enlaces entre particiones, así como la sincronización de estado requerida en réplicas de datos.
- Mantener una distribución equilibrada de datos y carga de trabajo de las consultas a través de las particiones.

- Preservar la calidad de las particiones asumiendo las modificaciones de la topología del grafo en grafos dinámicos y cambios de la capacidad del sistema elástico y patrones de acceso de las consultas.
- Reducir la memoria y los requerimientos de tiempo al no hacer suposiciones sobre la vista completa del grafo. En otras palabras, es preferible tener una partición local en cada servidor de manera que no es necesario sincronizar un estado global compartido entre servidores.

Sistemas de bases de datos en grafo como Neo4j, GraphchiDB, HypergraphDB, DEX, MAGS y Graphchi evitan el particionamiento de los datos del grafo.

2.5 Teorema CAP

Asumamos un sistema de almacenamiento distribuido que almacena un ítem de datos D replicado en tres nodos N1, N2 y N3, y un fallo de comunicación que divida la red (de los nodos) en dos subredes: {N1, N2} y {N3}. Si una solicitud de modificación se presenta en N3, entonces hay dos escenarios posibles. El primero, la solicitud se completa exitosamente, teniendo en cuenta que remediando la fragmentación el valor modificado de D se propaga a sus réplicas en N1 y N2. Este escenario prioriza la disponibilidad de la solicitud. Sin embargo, puede producir valores inconsistentes de D. En el segundo escenario, la solicitud es abortada, conociendo que el contacto con N1 y N2 no es posible hasta la restauración de la fragmentación. Este escenario prioriza la consistencia fuerte del ítem de datos D. Sin embargo, esto conlleva la indisponibilidad de la consulta. Este ejemplo simple presenta la cuestión de si la consistencia fuerte de datos y la disponibilidad de las consultas pueden ser logrados de manera simultánea [Brewer12].

Esta compensación fue observada por primera vez por Rothnie y Goodman. Sin embargo, la creciente popularidad comercial de la Web junto con la demanda incremental de replicación geográfica de datos y alta disponibilidad de las operaciones motivaron a Fox y Brewer a reclamar ese intercambio como el principio CAP. Este principio indica que <<sólo dos de las tres propiedades (Consistencia, Disponibilidad y tolerancia a particionamiento) pueden ser logradas simultáneamente por una base de datos distribuida>>. Más adelante, Gilbert y Lynch formalizaron y demostraron CAP, que se convirtió en el teorema CAP. En este contexto, las propiedades CAP se definen de la siguiente manera [Laudon96]:

- Consistencia (C): está vista como una propiedad cualitativa que denota coherencia. Los datos deben encontrarse sincronizados y replicados en todos los nodos simultáneamente. De esta forma, todos los clientes ven los mismos datos con independencia del nodo al que se conecten.
- Disponibilidad (Availability - A): está vista como una propiedad cualitativa denotando que cada solicitud enviada por un cliente eventualmente (dentro de un tiempo finito) recibe una respuesta exitosa. Sin embargo, esta definición presenta algunas ambigüedades. En primer lugar, algunos sistemas que son altamente disponibles debido a su elevado tiempo de actividad pueden no ser considerados como disponibilidad CAP. Por ejemplo, una base de datos distribuida que utiliza una replicación síncrona basada en quórum no presenta disponibilidad CAP debido a que, durante una partición de la red, las operaciones de lectura y escritura en la

partición más pequeña pueden no ser satisfechas. En segundo lugar, no hay un tiempo de respuesta establecido. Por ejemplo, una operación que se completa después de una semana podría considerarse que cumple disponibilidad CAP.

- Tolerancia a las particiones (Partition tolerance - P): está vista como una propiedad cualitativa denotando que incluso en presencia de una partición de red, el sistema continúa proporcionando garantías CAP de disponibilidad o consistencia. Esta definición es también confusa, ya que las particiones de la red no son el único fallo en un sistema distribuido. En otras palabras, existen otros fallos, como mensajes perdidos y fallos de nodos.

Las bases de datos distribuidas se categorizan de la siguiente manera:

- Consistencia + Disponibilidad -> Sistemas CA. Los algoritmos utilizados por los sistemas CA no soportan particiones en la red. Como consecuencia, lograr esta combinación es prácticamente imposible en sistemas distribuidos, ya que las particiones en la red son inevitables. Por tanto, el problema fundamentalmente recae en prescindir de consistencia o disponibilidad. Este dilema se convirtió en una justificación por apoyar la consistencia débil y justificar las decisiones de diseño de las bases de datos distribuidas, especialmente los almacenes NoSQL en los cuales la consistencia se sacrifica más que la disponibilidad.
- Consistencia + Tolerancia a las particiones -> Sistemas CP. Esta combinación se logra en las bases de datos que preservan la consistencia CAP. Sin embargo, en el caso de un particionamiento de la red, una solicitud de lectura o escritura puede no ser respondida debido a la necesidad de evitar el riesgo de inconsistencia. Por tanto, CP tiene sentido para sistemas diseñados para operar en una red confiable, como un centro de datos único, debido a la poca frecuencia de particiones en la red. Esta combinación se logra en algunos sistemas, como Scalaris, el servicio de bloqueo de Google's Chubby y Spanner.
- Disponibilidad + Tolerancia a las particiones -> Sistemas AP. Esta combinación la logran las bases de datos distribuidas que aplican una noción débil de consistencia. Sin embargo, la ejecución de escrituras conflictivas está permitida, lo que puede provocar divergencia de réplicas y requiere implementar un mecanismo de resolución de conflictos. Estos sistemas se utilizan típicamente en aplicaciones cuyos usuarios se encuentran dispersos geográficamente en un área amplia y requieren un alto nivel de disponibilidad junto con un breve tiempo de respuesta (en vez de consistencia fuerte), como el almacenamiento en caché web. Muchas bases de datos NoSQL, como Amazon Dynamo, Couch DB y Apache Cassandra logran esta combinación [Jiaqing14].

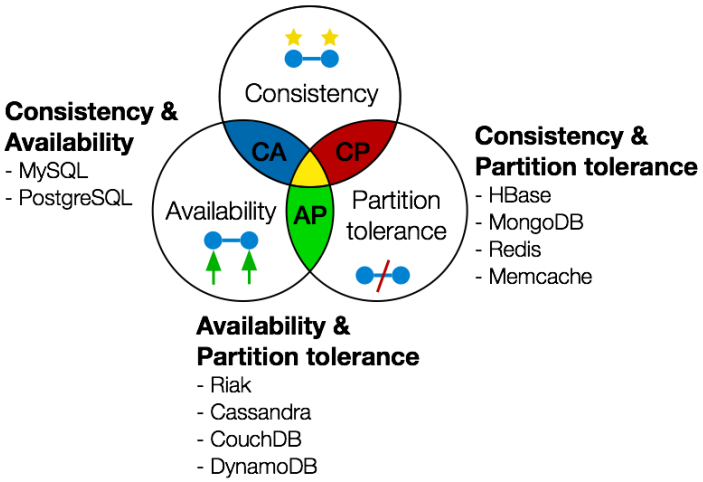

Ilustración 16: Categorías de sistemas gestores de bases de datos según el teorema CAP[7]

Brewer anunció que un sistema debería estar diseñado de manera que proporcionara propiedades CAP continuas [Simon00]. El peso de esas propiedades puede ser modificado entre ellas para optimizarlas según los requerimientos de las aplicaciones. Esto requiere utilizar una estrategia para la detección de particiones en la red y gestión de invariantes del servicio y operaciones. Por ejemplo, en una aplicación de reservas de una aerolínea, mientras que haya sitios libres en el avión, la aplicación puede contar con datos inconsistentes. Sin embargo, a medida que el avión se ocupa gradualmente, el nivel de consistencia de datos debería de ir incrementándose para prevenir la sobreventa. Una aplicación debería de imponer varios niveles de requerimientos CAP en muchas dimensiones:

- Diferentes tipos de datos soportados. Por ejemplo, la información de un producto gestionada por un carrito de compra puede ser inconsistente. Sin embargo, la compra, la facturación y los datos de envío deben ser fuertemente consistentes.
- Diferentes tipos de operaciones. Por ejemplo, Yahoo Pnuts proporciona operaciones de lectura y escritura fuertemente consistentes que acceden a los datos en una réplica primaria, así como operaciones de lectura consistentes cronológicas que acceden a datos en réplicas locales.
- Correlación entre datos, por lo cual los datos que son más propensos a ser accedidos juntos se sitúan en una partición común. Esto incrementa la disponibilidad de operaciones entre datos correlacionados, mientras que lo hace menos vulnerable a la indisponibilidad por particionamiento de la red. Por ejemplo, los usuarios de una red social que pertenecen al mismo grupo de amigos deberían estar situados en la misma partición.

[7] https://www.researchgate.net/

3 Bases de datos en grafo

3.1 Introducción

Las bases de datos en grafo abordan una de las grandes tendencias empresariales actuales: aprovechar las relaciones dinámicas y complejas en los datos fuertemente relacionados para obtener información y ventajas competitivas. Bien queramos entender las relaciones entre consumidores, elementos en un centro de datos, consumidores de entretenimiento, o genes y proteínas, la habilidad de entender y analizar grandes grafos de datos fuertemente relacionados será clave para determinar qué compañías superarán a sus competidores en las próximas décadas [Bryce].

Para datos de cualquier tamaño significante, las bases de datos son la mejor alternativa para representar y consultar datos relacionados. Los datos conectados son aquellos cuya interpretación y valor requieren en primer lugar entender la forma en la que sus elementos están relacionados. A menudo, para facilitar esta comprensión necesitaremos nombrar y calificar las conexiones entre elementos.

Lo más curioso de este renacimiento de los datos en grafo y el pensamiento en grafos es que esta teoría no es nueva. La teoría de grafos fue pionera por Euler en el siglo XVIII [Biggs86], y ha sido investigada y mejorada por matemáticos, sociólogos y antropólogos desde entonces. Sin embargo, sólo ha sido en los últimos años cuando la teoría de grafos y pensamiento en grafos se ha aplicado a la gestión de información. Desde entonces, las bases de datos han ayudado a solucionar problemas importantes en el área de las redes sociales, gestión de datos maestros, información geoespacial, recomendaciones y demás. Este foco en las bases de datos ha sido ocasionado por dos fuerzas: el masivo éxito de compañías como Facebook, Google y Twitter, las cuales han centrado su modelo de negocio alrededor de sus propias tecnologías de grafos; y por la introducción de bases de datos en grafo de propósito general en el ámbito tecnológico.

Para comprender las bases de datos en grafo, no es necesario conocer demasiada teoría. Con conocer qué es un grafo, podremos comprender el funcionamiento de este modelo. Formalmente, un grafo es simplemente una colección de vértices y aristas, en otras palabras: un conjunto de nodos y las relaciones que los conectan. Los grafos representan entidades como nodos y las forma en que se interactúan en la realidad son las relaciones.

Por ejemplo, los datos de Twitter se representan fácilmente con un grafo. En la Ilustración 17 podemos ver una pequeña red de usuarios de Twitter. Cada nodo está etiquetado como “User", indicando su rol en la red. Estos nodos están conectados con relaciones, que ayudan a establecer el contexto semántico: es decir, que Billy sigue a Harry, y que Harry, a su vez, sigue a Billy. Ruth y Harry igualmente se siguen entre ellos, pero, aunque Ruth sigue a Billy, este no la sigue.

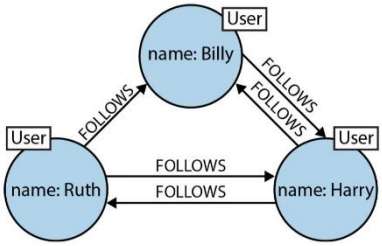

Ilustración 17: Ejemplo de grafo en una red social[8]

Evidentemente, el grafo real de Twitter es cientos de millones de veces más grande que el del ejemplo, pero funciona con los mismos principios.

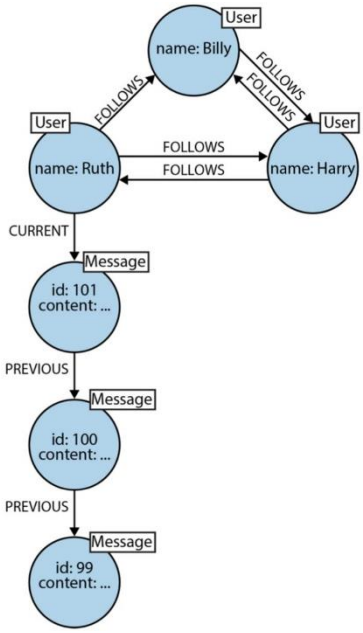

Ilustración 18: Ejemplo de grafo en una red social[9]

La Ilustración 18 muestra de forma sencilla el potencial del modelo en grafo. Es fácil ver que Ruth ha publicado una cadena de mensajes. Su mensaje más reciente se puede encontrar siguiendo una relación marcada como “CURRENT”. Las relaciones “PREVIOUS” forman la línea de tiempo de Ruth.

En el ejemplo anterior hemos introducido también informalmente la forma más popular de los modelos de grafo. El grafo etiquetado tiene las siguientes características:

- Contiene nodos y relaciones.
- Los nodos contienen propiedades (pares clave-valor).
- Los nodos pueden ser etiquetados con una o más etiquetas.
- Las relaciones están nombradas y dirigidas, y siempre tienen un nodo de inicio y otro de fin.
- Las relaciones pueden contener también propiedades.

[8] https://neo4j.com/developer/cypher/

[9] https://neo4j.com/developer/cypher/

3.2 Áreas de aplicación

Como se ha mencionado anteriormente, el modelado de datos en grafo representa de forma óptima las relaciones entre entidades de una base de datos. De esta forma, existen multitud de aplicaciones donde este modelo aventaja al resto. Algunos de estas aplicaciones son las siguientes [Robinson]:

- **Detección de fraudes**

Los bancos y compañías aseguradoras pierden billones de dólares anuales debido al fraude. Los métodos tradicionales de detección de fraudes fracasan al minimizar estas pérdidas debido a que realizan análisis discretos susceptibles a falsos positivos y negativos. Conociendo esto, cada vez más estafadores desarrollan formas de explotar estas debilidades de análisis discretos.

Las bases de datos en grafo ofrecen nuevos métodos para descubrir organizaciones fraudulentas y otras estafas con un alto nivel de precisión a través del análisis de enlaces, permitiendo frenar escenarios de fraude en tiempo real.

Al contrario que muchas otras formas de analizar los datos, los grafos están diseñados para expresar relaciones. Las bases de datos en grafo descubren patrones que son difíciles de detectar utilizando representaciones tradicionales como tablas.

Por ejemplo, si en un fraude de comercio electrónico se considera una transacción online con los siguientes identificadores: ID del usuario, dirección IP, geolocalización, una cookie de seguimiento y un número de tarjeta de crédito, típicamente, las relaciones entre estos identificadores deberían de ser de uno-a-uno. Tan pronto como las relaciones entre estas variables exceden un número razonable, el fraude podría considerarse una posibilidad.

El siguiente grafo representa una serie de transacciones realizadas desde diferentes direcciones IP con un escenario de fraude muy probable ocurriendo desde la IP1, la cual ha realizado múltiples transacciones con cinco tarjetas de crédito diferentes.

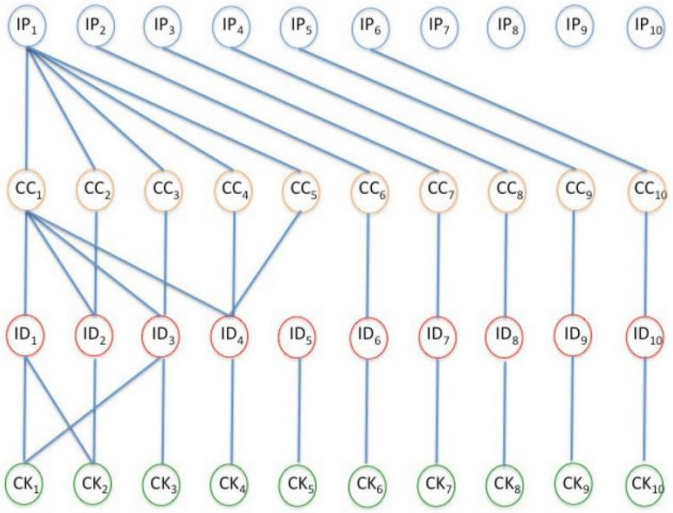

Ilustración 19: Ejemplo de grafo en un caso de fraude [Robinson]

- **Motores de recomendación en tiempo real**

En el caso de empresas que operan en la venta al por menor, sectores sociales, servicios y medios de comunicación, ofrecer recomendaciones en tiempo real y ajustadas a los usuarios es esencial para maximizar la valoración del cliente y permanecer competitivo. Con las bases de datos en grafo, es posible capturar el comportamiento de las búsquedas de un cliente y datos demográficos y combinar estos con su historial de compras para analizar instantáneamente sus elecciones actuales e inmediatamente proporcionar recomendaciones relevantes.

Hacer efectivas las recomendaciones en tiempo real depende de una base de datos que entienda las relaciones entre entidades, al igual que la calidad y fortaleza de esas conexiones. Solo una base de datos en grafo consigue enlazar esas relaciones de acuerdo con las compras, interacciones y reseñas del usuario para dar la percepción más relevante de las necesidades del usuario y tendencias de productos.

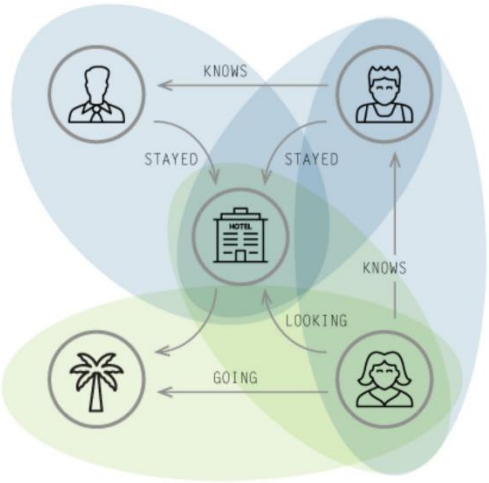

Ilustración 20: Ejemplo de grafo en un motor de recomendación [Robinson]

- **Gestión de datos maestros**

Los datos maestros son el alma de una compañía, incluyendo datos como:

- Usuarios
- Clientes
- Productos
- Cuentas
- Socios
- Lugares
- Unidades de negocio

Muchas aplicaciones de negocios utilizan datos maestros y éstos son a menudo almacenados en diferentes lugares, con redundancia, en diferentes formatos, y con grados diferentes de calidad y formas de acceso. La gestión de datos maestros (MDM) es la práctica de identificar, limpiar, almacenar y gestionar estos datos.

Debido a que los datos maestros están fuertemente conectados y distribuidos, los sistemas MDM construidos de forma deficiente suponen pérdidas en la agilidad de negocio. Muchos de estos sistemas dependen de una base de datos relacional que no está optimizada para recorrer relaciones o reaccionar a las consultas rápidamente.

Con las bases de datos en grafo, los datos maestros son más fáciles de modelar, suponiendo un coste menor de recursos (modeladores, arquitectos, DBAs y desarrolladores) que diseñando una solución relacional. Además, con una base de datos en grafo no es necesario migrar todos los datos en una única ubicación. Las relaciones conectan fácilmente los datos aislados entre sistemas CRM (tecnología empleada para gestionar todas las relaciones de la una compañía y las interacciones con clientes y potenciales clientes), sistemas de inventariado, contabilidad y sistemas de punto de venta para proporcionar una visión consistente de los datos de la empresa.

Un ejemplo práctico sería la estructura jerárquica de los empleados de una empresa. Una jerarquía sencilla como la de la figura 21 será fácil de mantener y modelar en una base de datos relacional. Sin embargo, la mayoría de las empresas cuentan con una estructura más compleja, donde las consultas y la mantenibilidad de los datos es más costoso. Por ejemplo, si un empleado consigue una promoción, cada relación debe ser reestructurada para cada jerarquía en la que el empleado trabaje.

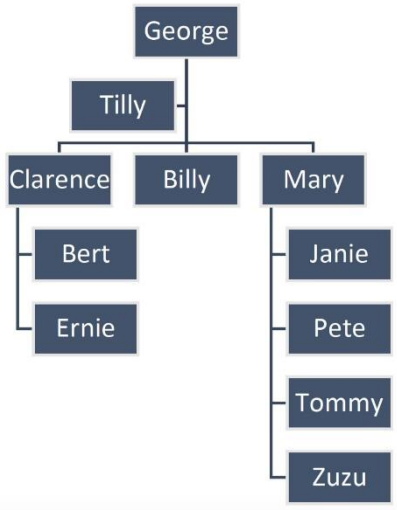

Ilustración 21: Estructura jerárquica de una empresa [Robinson]

- **Redes y operaciones IT**

Por su naturaleza, las redes son grafos. Las bases de datos en grafo son, por tanto, un modelo excelente para almacenar, modelar y consultar datos de redes y operaciones IT.

Actualmente, las bases de datos en grafo se están empleando de forma exitosa en las áreas de las telecomunicaciones, gestión de redes, análisis de impacto, gestión de plataformas en la nube y gestión de activos IT y centros de datos.

En todos estos dominios, las bases de datos almacenan información de configuración para alertar a los operadores en tiempo real de fallos potenciales en la infraestructura y para reducir problemas de análisis y tiempos de resolución de horas a segundos.

Al igual que con los datos maestros, las bases de datos en grafo se utilizan para acoplar información de diferentes sistemas, proporcionando una visión conjunta de la red y sus consumidores – desde el elemento más pequeño de la red a las aplicaciones, servicios y usuarios que la utilizan.

3.3 Importancia y trascendencia

El hecho de que las bases de datos en grafo proporcionen una técnica potente pero novedosa no supone una justificación suficiente para reemplazar una plataforma de datos tan establecida y comprendida. En el caso de las bases de datos en grafo, esta motivación existe en la forma de una serie de casos de uso y patrones de datos cuyo rendimiento mejora por uno o más órdenes de magnitud cuando se implementa en un grafo, y cuya latencia es mucho menor en comparación con el procesamiento de agregados por conjuntos. En el primer lugar de esta mejora de rendimiento, las bases de datos en grafo ofrecen un modelo de datos extremadamente flexible y un modelo de presentación alineado con las prácticas de presentación de software ágiles actuales [Pokorný15]. Por tanto, los puntos fuertes de este modelado de datos son los siguientes:

- **Rendimiento**

Una razón convincente para elegir una base de datos en grafo es el incremento de rendimiento total al trabajar con datos conectados frente a bases de datos relacionales y almacenes NoSQL. En contraste con las bases de datos relacionales, donde el rendimiento de las consultas dependientes de *joins* se ve perjudicado a medida que el conjunto de datos aumenta, con una base de datos en grafo el rendimiento permanece relativamente constante, incluso con el crecimiento del volumen de datos. Esto es debido a que las consultas se localizan en una porción del grafo. Como resultado, el tiempo de ejecución para cada consulta es proporcional únicamente con el tamaño de la parte del grafo atravesado para satisfacer esa consulta, en vez del tamaño del grafo completo.

- **Flexibilidad**

Los grafos son aditivos naturales, de manera que podemos añadir nuevos tipos de relaciones, nuevos nodos, nuevas etiquetas, y nuevos subgrafos a la estructura existente sin perturbar las consultas existentes y la funcionalidad de la aplicación. Esto tiene implicaciones generales positivas para la productividad del desarrollador y riesgo del proyecto. Debido a la flexibilidad del modelo en grafos, no tenemos que modelar nuestro dominio con detalles exhaustivos antes de tiempo (una práctica temeraria debido a los cambios en los requerimientos de negocio). La adición natural de los grafos también supone la tendencia a realizar menos migraciones reduciendo un mantenimiento y riesgos elevados.

- **Agilidad**

Las bases de datos en grafo modernas nos equipan para realizar un desarrollo sin fricciones y un mantenimiento del sistema sencillo. En particular, la naturaleza del esquema libre del modelado en grafo, en conjunto con la naturaleza testeable de la interfaz de programación de aplicaciones de una base de datos en grafo (API) y lenguaje de consultas, nos permite evolucionar una aplicación de forma controlada.

Al mismo tiempo, precisamente porque son libres de esquema, estas bases de datos no tienen esos mecanismos de gestión de datos orientados al esquema típicos del mundo relacional. Pero esto no supone un riesgo, conduce a un tipo de gobierno sobre los datos mucho más visible y procesable. La gestión de los datos está aplicada típicamente en una moda

programática, utilizando tests para expulsar el modelo de datos y consultas, así como imponer las normas de negocio sobre el grafo.

3.4 Componentes

Una base de datos en grafo es cualquier sistema de almacenamiento que utiliza estructuras de grafo con nodos y aristas, para representar y almacenar datos. El modelo de grafos más común utilizado en el contexto de bases de datos en grafo es el conocido como grafo etiquetado con propiedades. Éste contiene entidades (los nodos) conectadas que pueden tener asignadas cualquier cantidad de propiedades (atributos) expresadas como pares clave-valor. Los nodos y aristas se pueden marcar con etiquetas representando sus diferentes roles en la aplicación de dominio. Estas etiquetas pueden utilizarse también para agregar metadatos (información o restricciones) a ciertos nodos.

Las relaciones proporcionan conexiones (aristas) directas y semánticamente relevantes entre dos nodos. Una relación siempre tiene una dirección, un nodo de inicio, y un nodo de fin. Al igual que los nodos, las relaciones pueden tener propiedades. A menudo, las relaciones tienen propiedades cuantitativas, como peso, coste, distancia, clasificaciones o intervalos de tiempo. Tanto los nodos como las aristas tienen asociado un identificador único.

Debido a que las relaciones se almacenan de forma eficiente, dos nodos pueden compartir cualquier número de relaciones de tipos diferentes sin sacrificar rendimiento. Cabe destacar que, aunque sean dirigidas, las relaciones siempre podrán ser navegadas independientemente de la dirección.

En ocasiones, podremos trabajar con hípergrafos en el software de bases de datos en grafo. Un hípergrafo es una generalización del concepto de grafo, en el cuál las aristas son sustituidas por híperaristas. Si una arista regular conecta dos nodos de un grafo, entonces una híperarista conecta un número cualquiera de nodos.

Podemos distinguir un número de componentes básicos de la tecnología de bases de datos en grafo. Estos incluyen el almacenamiento de grafos, las consultas en grafos, escalabilidad y procesamiento de las transacciones. Se desarrollarán estos puntos en las siguientes secciones.

3.4.1 Almacenamiento de grafos

Un rasgo importante de las bases de datos en grafo es que proporcionan capacidades de procesamiento nativo, una propiedad conocida como adyacencia libre de índices, significando que cada nodo está enlazado directamente con su nodo vecino. Un sistema de almacenamiento que utilice adyacencia sin índice es aquél en el cual cada nodo inicio mantiene referencias directas a sus nodos adyacentes; cada nodo, por tanto, actúa como un índice para otros nodos cercanos, lo cual es mucho menos costoso que utilizar índices globales. Esto es apropiado para consultas locales del grafo donde necesitamos un índice de referencia para el nodo de inicio, y después atravesaremos las relaciones mediante punteros físicos directamente. En los sistemas

de bases de datos relacionales, probablemente necesitaríamos concatenar más tablas mediante claves ajenas y, posiblemente, índices de búsqueda adicionales.

Algunas bases de datos en grafo ofrecen una interfaz de grafo sobre almacenes de grafos no-nativos, como una base de datos columnar en el Virtuoso Universal Server en aplicación para datos RDF (un modelo de datos intercambiado en la web). Por ejemplo, la base de datos en grafos FlockDB almacena datos en grafos, pero no está optimizada para operaciones de recorrer grafos. En cambio, está optimizada para listas de adyacencia muy extensas. FlockDB utiliza MySQL como el sistema de almacenamiento de bases de datos básico sólo para almacenar listas de adyacencia.

3.4.2 Consultas sobre grafos

En la práctica, las consultas básicas son las más frecuentes. Éstas incluyen una vista al nodo, a los vecinos (de un salto), explorar aristas en múltiples saltos (capas), recuperar los valores de un atributo, etc. Consultar un nodo basándose en sus propiedades o a través de su identificador se llama punto de consulta.

Recuperar un nodo por su *id*, puede no ser una operación constante. Algunas consultas complejas frecuentes son aquellas sobre subgrafos y supergrafos. Otras consultas típicas incluyen búsquedas en profundidad y anchura, búsqueda de caminos y camino más corto, encontrar subgrafos densos, encontrar componentes fuertemente conectados, etc. Los algoritmos utilizados para consultas tan complejas necesitan frecuentemente computación iterativa.

Inspirado por el lenguaje SQL, las bases de datos en grafo están equipadas a menudo con un lenguaje de consulta declarativo. Actualmente, el lenguaje de consulta declarativo sobre grafos más conocido es Cypher trabajando con bases de datos Neo4j, cuyos comandos son similares a la sintaxis de SQL. Un lenguaje sobre grafos más bien procedimental es el lenguaje transversal Gremlin.

La respuesta más distintiva para una consulta sobre grafos es otro grafo, lo cual es ordinariamente una transformación, una selección o proyección del grafo original almacenado en la base de datos. Esto implica que la visualización de un grafo está fuertemente enlazada a su consulta.

3.4.3 Escalabilidad

La fragmentación es crucial para hacer que los grafos escalen. Escalar los datos de un grafo distribuyéndolos entre múltiples máquinas es más complejo que escalar datos más simples en otras bases de datos NoSQL, pero es posible. La razón es la naturalidad con la que los datos del grafo están conectados. Cuando distribuimos un grafo, queremos evitar tener relaciones que abarquen las máquinas en la medida de lo posible; esto se llama el "*minimum point-cut problem*". Pero lo que parece una buena distribución en cierto momento puede no ser óptimo unos segundos más tarde. Típicamente, los problemas de la fragmentación de los grafos se

incluyen en la categoría de problemas NP-hard. La escalabilidad está conectada con tres ámbitos:

- Escalabilidad para grandes conjuntos de datos.
- Escalabilidad para rendimiento de lectura
- Escalabilidad para rendimiento de escritura

En la práctica, el primero es el más discutido. Actualmente, no es un problema en el área de las bases de datos en grafo. Por ejemplo, Neo4j tiene un límite superior de tamaño del grafo del orden de 10^10. Esto es suficiente para soportar la mayoría de los grafos del mundo real, incluyendo un despliegue en Neo4j que contenga más de la mitad de los grafos sociales de Facebook en un clúster de Neo4j.

La escalabilidad para lecturas normalmente no presenta problemas. Por ejemplo, Neo4j se ha enfocado históricamente en el rendimiento de lecturas. En régimen maestro-esclavo las operaciones de lectura se pueden realizar localmente en cada esclavo. Para mejorar la escalabilidad en cargas de trabajo altamente concurrentes, Neo4j utiliza dos niveles de caché.

La escalabilidad en lecturas se puede lograr mediante escalabilidad vertical, pero en algún punto, para cargas extremadamente pesadas, se requiere la habilidad de distribuir los datos a través de múltiples máquinas. Este es el verdadero reto. Por ejemplo, Titan es un sistema de bases de datos en grafo OLTP altamente escalable que está optimizado para soportar cientos de usuarios accediendo concurrentemente y actualizando un gran grafo.

3.4.4 Procesamiento de transacciones

Al igual que en cualquier otro sistema gestor de base de datos, existen tres casos de uso genéricos para grafos:

- Aplicaciones CRUD (*create, read, update, delete*)
- Procesamiento de consultas – informando, almacenando datos, y analíticas en tiempo real
- Analíticas por conjuntos o descubrimiento de datos

En particular, los dos primeros usos se focalizan en procesamiento de transacciones. Por ejemplo, bases de datos OLTP. Al tratar con muchas transacciones concurrentes, la naturaleza de la estructura de datos del grafo ayuda a distribuir la carga transaccional a través de los grafos. A medida que el grafo crece, los conflictos transaccionales típicamente se reducen. Pero no todas las bases de datos son completamente ACID. Sin embargo, la variante basada en las propiedades BASE a menudo considerado en el contexto de las bases de datos NoSQL no es demasiado apropiado para grafos.

En general, el procesamiento de grafos distribuidos requiere de la aplicación de estrategias de fragmentación y replicación apropiadas para maximizar la localidad del procesamiento. Por ejemplo, minimizar la necesidad de enviar datos entre diferentes nodos de la red.

Por ejemplo, Neo4j utiliza replicación maestro-esclavo, donde una máquina se designa como el maestro y el resto como esclavos. En Neo4j, todas las escrituras dirigidas hacia cualquier

máquina pasan a través del maestro, el cual a su vez envía actualizaciones a los esclavos cuando lo obtiene. Si el maestro falla, el clúster elige automáticamente otro maestro.

Neo4j necesita un quórum para realizar escrituras locales. Esto significa que una mayoría estricta de los servidores del clúster tienen que estar online para aceptar operaciones de escritura. De no ser así, el clúster solo permitirá operaciones de lectura hasta que el quórum se establezca. Por ello, cabe destacar que las bases de datos en grafo actuales no tienen el mismo nivel de rendimiento de escritura que otros tipos de bases de datos NoSQL. Esto es una consecuencia del *clustering* maestro-esclavo y transacciones adecuadas ACID.

Algunas arquitecturas más complejas tienen lugar en el mundo de las bases de datos en grafo. Típicamente, una base de datos simple se utiliza para absorber datos, y después volcar los datos en una base de datos en grafo para refinamiento y análisis. La arquitectura Neo4j contiene incluso un cargador que opera a un ritmo de millones de registros por segundo.

3.5 Categorías de sistemas de bases de datos en grafo

Podemos distinguir entre sistemas gestores de bases de datos generales, como Neo4j, InfiniteGraph, Sparksee, Titan, GraphBase y Trinity, y especiales, como la base de datos Web InfoGrid y FlockDB, o bases de datos multimodelo como bases de datos orientadas a documentos permitiendo recorridos entre documentos. Por ejemplo, OrientDB presenta en conjunto la potencia de los grafos y la flexibilidad de los documentos en una base de datos escalable incluso con una capa SQL. HyperGraphDB no sólo almacena grafos, sino también estructuras de hípergrafos. Toda la información de los grafos se almacena en pares clave-valor [Angles].

Una pregunta interesante es qué base de datos en grafos es la más popular hoy en día. En junio de 2015, la página Web DB-Engines Ranking of GDBMS estableció a Neo4j, OrientDB y Titan como las más populares.

3.5.1 Sistemas de Gestión de Bases de datos en grafo de propósito general

- **Neo4j**

Neo4j es la plataforma de gestión de bases de datos en grafo más popular globalmente. Se trata de un sistema de base de datos en grafo nativo robusto (completamente ACID), altamente escalable y de código abierto.

Neo4j almacena los datos como nodos y relaciones. Ambos, nodos y relaciones, pueden contener propiedades en forma de clave-valor. Los valores pueden ser de tipo primitivo o un array de un tipo primitivo. Los nodos se utilizan a menudo para representar entidades, pero dependiendo del dominio las relaciones se pueden utilizar para ese propósito también. Los

nodos y aristas tienen identificadores internos únicos que pueden utilizarse para la búsqueda de datos. Los nodos no se pueden referenciar a sí mismos directamente. Las semánticas pueden expresarse añadiendo relaciones directas entre nodos.

El procesamiento de grafos en Neo4j implica principalmente el acceso a datos aleatorios lo que puede ser inadecuado para Big Graphs (grafos de gran tamaño). Los grafos que no caben en memoria principal podrían requerir más accesos de disco, lo que influye significativamente el procesamiento del grafo. De forma similar a otras colecciones de Big Data, los Big Graphs tienen que ser fragmentados en múltiples máquinas para lograr un procesamiento escalable.

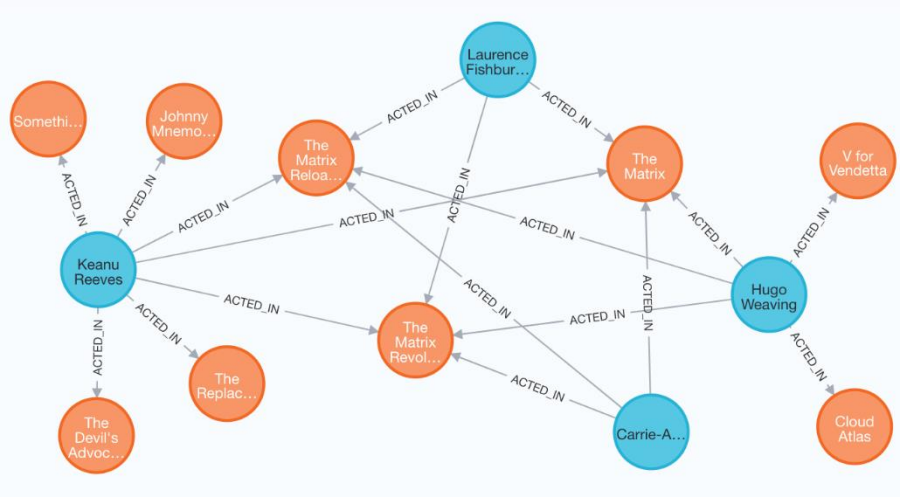

Ilustración 22: Ejemplo de grafo en Neo4j[10]

- **Sparksee**

Además del modelo básico de grafo, Sparksee también introduce la noción de una arista virtual que conecta nodos con el mismo valor de un atributo. Estas aristas no están materializadas. Un grafo Sparksee se almacena en un único archivo; los valores e identificadores se mapean mapeando funciones en árboles B+. Los mapas de bits se utilizan para almacenar nodos y aristas de un cierto tipo[11].

La arquitectura de Sparksee incluye el núcleo, que gestiona y consulta las estructuras del grafo, después una capa API para proporcionar una interfaz de programación de la aplicación, y la capa más alta de aplicaciones, para extender las capacidades del núcleo y visualizar y buscar los resultados. Para acelerar las diferentes consultas del grafo y otras operaciones, Sparksee ofrece estos tipos de índice:

- Atributos
- Atributos únicos
- Relaciones para indexar sus nodos vecinos
- Índices en nodos vecinos

[10] https://neo4j.com/

[11] https://dbdb.io/db/sparksee

Sparksee implementa un número considerable de algoritmos sobre grafos. Por ejemplo, búsqueda de camino más corto, búsqueda en profundidad y búsqueda de componentes fuertemente conectados.

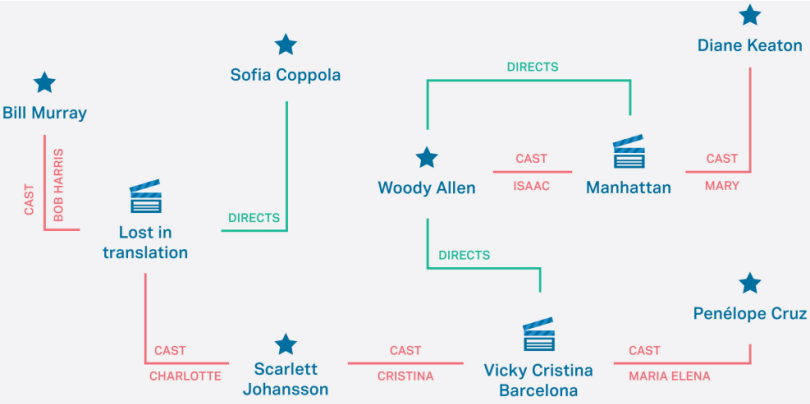

Ilustración 23: Ejemplo de multigrafo en Sparksee[12]

3.5.2 Sistemas *triplestores*

Algunos productos orientados a grafos están orientados a satisfacer aplicaciones especiales sobre grafos, mayormente datos RDF (un modelo conceptual que sirve para proporcionar información descriptiva sobre los recursos que se encuentran en la Web) en forma de sujeto (S) - predicado (P) - objeto (O). Un ejemplo de dato RDF o triple sería <<Pedro es persona>> o <<Pedro conoce a Miguel>> [Angles]. Estos *triplestores* emplean soluciones de gestión inteligente de los datos que combinan la búsqueda de texto total con analíticas de los grafos y razonamiento lógico para producir resultados más profundos.

El formato que emplea de SUJETO->PREDICADO->OBJETO permite tomar cualquier sujeto y conectarlo a cualquier otro objeto empleando el predicado (verbo) para mostrar el tipo de relación existente entre el sujeto y el objeto.

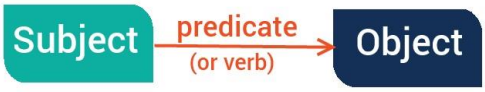

Ilustración 24: Estructura de dato RDF o triple[13]

Por ejemplo, “María vende libros” puede ser almacenado como una sentencia RDF en un *triplestore* y describe la relación entre el sujeto, “María”, y el otro objeto, “libros”. El predicado “vende” muestra cómo el sujeto y el objeto están conectados.

Actualmente, GraphDB es el *triplestore* puntero a nivel global que puede llevar a cabo inferencia semántica a escala permitiendo a los usuarios crear nuevos elementos semánticos a

[12] https://www.sparsity-technologies.com/doc-sparksee/StartingGuide/StartingGuide.pdf

[13] https://dbdb.io/db/sparksee

partir de los a existentes. GraphDB está construido sobre OWL (*Ontology Web Language*). Éste utiliza ontologías que permiten al repositorio razonar automáticamente sobre los datos.

La tecnología *triplestore* existente, sin embargo, no está suficientemente desarrollada todavía para almacenar eficientemente grandes conjuntos de datos.

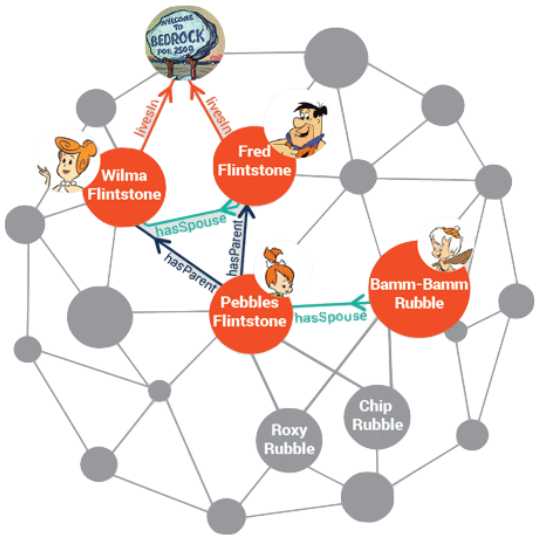

Ilustración 25: Ejemplo de grafo en un sistema triplestore[14]

3.5.3 Pregel y Giraph[15]

Pregel y Giraph son sistemas para procesamiento de grafos de gran escala. Éstos proporcionan un *framework* tolerante a fallos para la ejecución de algoritmos sobre grafos en paralelo sobre varias máquinas. Giraph utiliza la implementación del *framework* Apache MR para procesar grafos.

Una aproximación considerable sobre el diseño, análisis e implementación de algoritmos paralelos, hardware y software en Pregel es el modelo Bulk Synchronous Processing (BSP). BSP ofrece independencia de arquitectura y un rendimiento muy alto de algoritmos paralelos sobre múltiples ordenadores conectados por una red de comunicaciones. Está considerado como un candidato para ser el modelo de programación para computación paralela y Big Data en los próximos años. Por ejemplo, Google está migrando su infraestructura interna de MR a BSP/Pregel.

3.6 Limitaciones

En este apartado, vamos a enumerar las principales restricciones a tener en cuenta a la hora de emplear bases de datos orientadas a grafos [Pokorný15]:

[14] https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-triplestore/

[15] https://www.oreilly.com/library/view/yarn-essentials/9781784391737/ch07s05.html

Restricciones de funcionalidad

- **Consultas declarativas:** La mayoría de las bases de datos en grafo comerciales no pueden ser consultadas utilizando un leguaje declarativo. Sólo unos pocos proveedores ofrecen una interfaz de consultas declarativas. Esto también implica una falta de habilidades de optimización de las consultas.
- **Fraccionamiento de los datos:** La mayoría de las bases de datos en grafo no incluyen la funcionalidad de particionar y distribuir los datos en una red de ordenadores. Esto es esencial para soportar escalabilidad horizontal también. Es difícil fraccionar un grafo de forma que no tenga como resultado muchas consultas teniendo que acceder a múltiples particiones.
- **Operaciones vectorizadas:** Éstas soportan un procedimiento el cual secuencialmente escribe datos de múltiples buffers a un solo flujo de datos o lee datos de un flujo de datos a múltiples *buffers*. Las bases de datos NoSQL horizontalmente escaladas soportan este tipo de acceso a datos. Parece ser que no es el caso en las bases de datos en grafo actuales.
- **Restricciones de modelado:** Las posibilidades del esquema de datos y definición de restricciones están restringidas en las bases de datos en grafo. Por tanto, las inconsistencias de datos pueden reducir rápidamente su utilidad. A menudo, el modelo en grafo por sí mismo está restringido. Por ejemplo, los nodos en Neo4j no pueden referenciarse a sí mismos directamente. Existen casos en los que la referenciación de una entidad a sí misma es necesaria.
- **Restricciones de consultas:** Por ejemplo, FlockDB no permite saltos múltiples para recorrer el grafo, por lo que no puede realizar una "clausura transitiva" completa. Sin embargo, FlockDB permite el procesamiento de consultas de un salto muy rápido y escalable.

Grandes requisitos de análisis

- **Extracción a grafos:** Una cuestión es cómo extraer eficientemente un grafo o una colección de grafos de un almacén de datos NoSQL. La mayoría de sistemas de análisis de grafos asumen que el grafo se proporciona explícitamente. Sin embargo, en muchos casos, el grafo debe ser construido concatenando y combinando información de diferentes recursos que no son necesariamente en grafo. Incluso si los datos están almacenados en una base de datos en grafo, a menudo sólo necesitamos cargar un conjunto de subgrafos de esa base de datos para un mayor análisis.
- **Alto coste de algunas consultas:** La mayoría de los grafos del mundo real son altamente dinámicos y a menudo generan grandes volúmenes de datos a un ritmo muy elevado. Un reto aquí es cómo almacenar el rastro histórico de forma compacta mientras que se realizan ejecuciones eficientes de consultas puntuales y tareas de análisis globales o centradas en los nodos propios y cercanos.
- **Procesamiento en tiempo real:** El descubrimiento de datos del grafo tiene lugar esencialmente en "*batch environments*" (consiste en un programador de trabajos, aplicaciones, trabajos, interfaces para funciones de administración y tablas de bases de datos) como Giraph. Algunos productos tienen como objetivo el descubrimiento de datos y analíticas complejas que operen en tiempo real.
- **Algoritmos sobre grafos:** Algoritmos sobre grafos más complejos son necesarios en la práctica. La base de datos ideal debería comprender consultas analíticas que van

más allá de saltos pequeños entre nodos. Las bases de datos actuales (como las relacionales) tienden a priorizar la baja latencia en la ejecución de consultas sobre el análisis de datos de alto rendimiento.

- **Paralelización:** En el contexto de grandes grafos, hay una necesidad de paralelizar los algoritmos de procesamiento de los datos del grafo cuando los datos son demasiado grandes para manejar en un solo servidor. Existe la necesidad de entender el impacto sobre el rendimiento en los algoritmos de procesamiento de los grafos del dato cuando los datos no caben en la memoria disponible y para diseñar algoritmos explícitamente para esos escenarios.
- **Datos del grafo heterogéneos e inciertos:** Existe la necesidad de encontrar métodos automatizados para manejar la heterogeneidad, incompletitud e inconsistencia entre diferentes conjuntos de datos de grandes grafos que necesitan estar semánticamente integrados para ser consultados efectivamente o analizados.

Otros retos

Otros retos presentes en el desarrollo de bases de datos en grafo incluyen:

- Metodología de diseño de bases de datos en grafo.
- Necesidad de un *benchmark.*
- Desarrollo de heurísticas para algunos problemas complejos sobre el grafo.
- *Matching* de patrones del grafo.
- Compresión de grafos.
- Integración de los datos del grafo.
- Visualización.
- Procesamiento de los flujos del grafo, desarrollo de algoritmos para procesar flujos de datos de grandes grafos con el objetivo de estimar las propiedades del grafo sin almacenar el grafo entero.

En comparación con los sistemas de gestión de bases de datos relacionales tradicionales, existe una dificultad para identificar los tipos particulares de casos de uso para los que cada producto es más apropiado. El rendimiento varía considerablemente entre los diferentes sistemas de gestión de bases de datos en grafo dependiendo del tamaño del grafo y cómo de bien optimizada está una herramienta para una tarea en particular. Parece ser que especialmente para grandes grafos y grandes analíticas una gran cantidad de resultados previos y diseños tendrán que ser reconsiderados y replanteados en los próximos estudios y desarrollos.

4 Neo4j

Neo4j se trata de un proyecto *open-source*, respaldado por Neo Technology que tuvo sus inicios en 2003, con la aplicación siendo disponible de forma pública desde 2007. Esta plataforma está disponible para Windows, OS X, o Linux, por lo que puede ser instalada en cualquier dispositivo siempre que cumpla los requisitos mínimos de funcionamiento [Webber12].

Ilustración 26: Logo de Neo4j[16]

El código se encuentra disponible en GitHub, de manera que la comunidad de desarrolladores puede aportar en el desarrollo del producto. Existe una opción de empresa que ofrece prestaciones adicionales, soporte, y se trata esencialmente de un producto diferente. Esta versión es *closed-source*, por lo que sólo se encuentra disponible al pagar por la licencia. Esta versión solo es necesaria para grandes bases de datos en Neo4j y que requieren de soporte técnico en caso de incidencias.

Neo4j siempre se ha considerado como la base de datos en grafo líder a nivel mundial (de hecho, esta frase en inglés se encuentra en el título de su página web). Está implementada de forma nativa, es decir, el almacenamiento de los datos presentes en el grafo se realiza sobre una base de datos en grafo, a diferencia de las no nativas, donde el almacenamiento proviene de una fuente externa que puede ser relacional u otro tipo NoSQL. Neo4j es el ejemplo por excelencia de este tipo de base de datos en grafo, donde cada capa de su arquitectura, módulos y lenguaje de consulta están optimizados y pensados para almacenar todos los datos de un grafo sin utilizar tablas u otros métodos de almacenamiento.

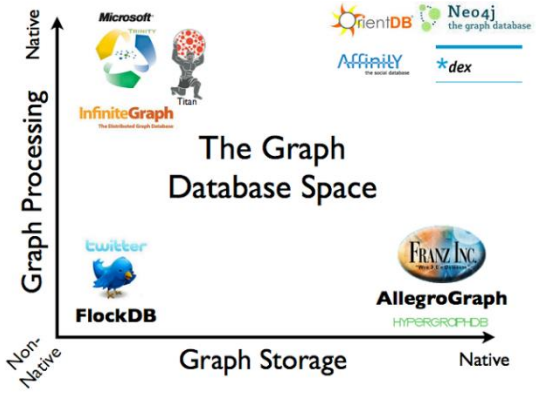

Ilustración 27: Categorización de sistemas gestores de bases de datos en grafo según su implementación[17]

[16] https://neo4j.com/

[17] https://www.oreilly.com/library/view/graph-databases-2nd/9781491930885/ch01.html

Existen multitud de motivos por los que utilizar Neo4j. En primer lugar, tiene una gran comunidad debido a sus más de 1.000.000 de descargas [Kemper15]. Se organizan más de 500 eventos de Neo4j al año, reuniendo a más de 20.000 miembros, permitiendo compartir nuevas técnicas, casos de uso, ejemplos e ideas. Desde su desarrollo, Neo4j ha ido evolucionando hasta convertirse en un producto altamente confiable ofreciendo escalabilidad, óptimas velocidades de lectura y escritura, y completo cumplimiento de ACID. Uno de los grandes beneficios de Neo4j es la facilidad con la que se puede aprender. El lenguaje de consulta, Cypher, está diseñado para ser descriptivo con el objetivo de hacerlo más fácil de comprender, y también de aprender. El equipo de Neo4j también ha publicado múltiples artículos útiles con diferentes casos de uso, con guías paso a paso. También, al construir una aplicación sobre una tecnología en particular, es necesario tener seguridad de que será capaz de hacer frente a la cantidad de tráfico que tendrá la aplicación. Neo4j está diseñado para soportar grandes volúmenes de tráfico. Por último, ofrece soporte a particiones y alta disponibilidad [Sun15] (HA) lo que significa que, gracias a su diseño maestro/esclavo y su habilidad de propagar cambios entre las otras instancias, la aplicación no sólo podrá seguir funcionando bajo presión, sino que lo hará también de forma rápida.

4.1 Cypher

Cypher fue diseñado con el objetivo de ser sencillo de leer, describiéndose como un lenguaje de consulta textual y declarativo como SQL. Se desarrolló con la finalidad de facilitar a los usuarios procedentes de SQL el uso de una base de datos NoSQL [Hodler19].

La sintaxis básica de Cypher consta de los siguientes elementos:

- () Nodo
- {} Propiedades
- [] Relaciones

Estos elementos se pueden combinar de múltiples formas para lograr diferentes consultas. A la hora de buscar entre los datos, añadir una propiedad puede reducir los datos a filtrar, basándose en el valor de esta propiedad. Lo mismo ocurre con las relaciones, añadir una restricción según las relaciones puede suponer una mejora a la hora de realizar una consulta, obteniendo unos resultados más condensados y relevantes.

Una consulta básica que es necesario conocer de este lenguaje es:

```
MATCH (n) RETURN n;
```

Esta consulta devuelve todos los nodos de la base de datos. Es aconsejable ejecutar esta sentencia únicamente en el entorno local, y no en producción. Si se lleva a cabo en una base de datos con millones de nodos, podría llevar mucho tiempo y bloquear algunas transacciones relevantes. En ese caso, sería aconsejable añadir alguna restricción a la consulta de manera que devuelva un conjunto más pequeño de datos. En ocasiones simplemente se desea consultar todos los nodos del grafo, por lo que se hace inevitable realizar esta consulta.

Las relaciones son uno de los puntos que en Cypher se trató de simplificar. También era importante lograr hacerlo un lenguaje descriptivo, lo que se puede observar en la siguiente consulta.

- Obtener todos los nodos enlazados por la relación "CONNECTED":

```
MATCH(a)-[:CONNECTED]->(b) RETURN a,b;
```

- Obtener los nodos que tienen el valor "Pepe" en la propiedad name:

```
MATCH (n {name:'PEPE'}) RETURN n;
```

- Obtener todos los nodos etiquetados como "Persona":

```
MATCH (n:Persona) RETURN n;
```

- Obtener todos los nodos Persona de nombre Pepe y más de 20 años:

```
MATCH(n:Person)
WHERE n.name='Pepe' AND n.age<20
RETURN n
```

- Obtener el nombre, edad y email de la persona que conoce María desde antes de 2010:

```
MATCH (n:Person {name:'María'})-[k:KNOWS]->(x:Person)
WHERE k.since<2010
RETURN x.name,x.age,x.email
```

También se pueden utilizar cláusulas como ORDER BY, SKIP, UNWIND, LIMIT, FOREACH y más para aumentar la complejidad de la consulta y obtener resultados más concretos y complicados de obtener.

Es necesario comprender también la forma en la que se crean, actualizan y eliminan los elementos de la base de datos.

- Crear un nodo Persona de nombre Pepe:

```
CREATE (n: Person {name: 'Pepe'})
```

- Conectar como Familia a dos poersonas de nombres A y B (crea la relación):

```
MATCH (a: Person), (b: Person)
WHERE a.name = 'A' AND b.name = 'B'
CREATE (a) - [ f: FAMILIA ] -> (b)
```

- Crear dos nodos personas de nombres Pepe y Sara y conectarlos como familia:

```
CREATE (sara:Persona{name:'Sara'})-[f:FAMILIA]->
                (pepe:Persona{name:'Pepe'})
```

- Empleamos SET para actualizar etiquetas de nodos y propiedades en nodos y relaciones. Ponerle a la persona Pedro, López como apellido:

```
MATCH (n{nombre:'Pedro'})
SET n.apellido='López'
```

- Eliminar todas las propiedades del nodo:

```
MATCH (p {nombre: 'Pedro'})
SET p = {}
```

En cuanto a la posibilidad de eliminar elementos, podemos ejecutar DELETE sobre nodos y/o relaciones.

- Eliminar los nodos de nombre Pedro:

```
MATCH (n: Persona {nombre: 'Pedro'})
DELETE n
```

- Eliminar las relaciones CONOCE que enlazan a Pedro con otros nodos:

```
MATCH (n {nombre: 'Pedro'}) - [r: CONOCE] -> ()
DELETE r
```

- Por último, es posible eliminar todos los nodos y relaciones de la base datos:

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

Es posible crear índices de los datos que nos interesen. Un índice es una copia redundante de la información que está siendo indexada, para hacer que las consultas sobre dicha información sean más rápidas. La parte negativa es que almacenar índices ocupa mucho espacio, y también ralentiza la velocidad de escritura. Esto es debido a que los índices necesitan ser actualizados cuando la información se almacena en la base de datos, suponiendo un coste en el rendimiento [Angles16].

Neo4j permite crear índices sobre propiedades de nodos que comparten la misma etiqueta. Si existe una propiedad particular que se consulta muy frecuentemente, entonces se debe plantear la posibilidad de añadir un índice para ésta. En algunos casos, un índice puede ser asignado automáticamente por el sistema, si considera que es una propiedad consultada con mucha frecuencia.

Al crear un índice en Neo4j, éste se actualiza automáticamente. Esto incluye cualquier actualización sobre el nodo que tiene propiedades en un índice y también cuando nuevos nodos se crean siguiendo el criterio establecido.

- Creación de un índice sobre la propiedad *nombre* de los nodos *Persona*:

```
CREATE INDEX ON: Persona (nombre)
```

- Eliminación de un índice sobre la propiedad *nombre* de los nodos *Persona*:

```
DROP INDEX ON :Persona (nombre)
```

Por último, cabe señalar que también se permite establecer restricciones. Por ejemplo:

- Creación de una restricción de unicidad sobre la propiedad *mail* de los nodos *Persona:*

```
CREATE CONSTRAINT ON (p: Persona) ASSERT p.email IS UNIQUE
```

Si se intenta crear un nodo que viola la restricción, la sentencia CREATE producirá un error y el nodo no se creará.

5 El Metro de Valencia

Con el objetivo de reflejar la utilidad y funcionalidad de las bases de datos en grafo, se va a presentar un caso práctico donde el uso de un modelo en grafo puede aplicarse de forma sencilla. Sobre la colección de datos presentados se realizarán una serie de consultas de utilidad real, así como modificaciones del grafo que podrían producirse debido a incidentes o averías de la red.

El sistema gestor de bases de datos empleado es Neo4j, debido a que es el sistema más popular actual y que presenta mejores prestaciones en comparación con el resto. Por tanto, las consultas sobre los datos almacenados se realizarán en Cypher.

Teniendo en cuenta las características de las bases de datos en grafo presentadas a lo largo del documento, podemos concluir que el modelo está diseñado especialmente para casos en los que las relaciones entre entidades son muy relevantes en la mayoría de las consultas que se van a realizar. El caso planteado es el modelado de la red de metro de Valencia. Con el fin de ajustarnos a la realidad al máximo, los datos de las estaciones, líneas y duración del trayecto entre paradas son reales y además está actualizado al momento en el que se está redactando el documento. La red representada es la siguiente:

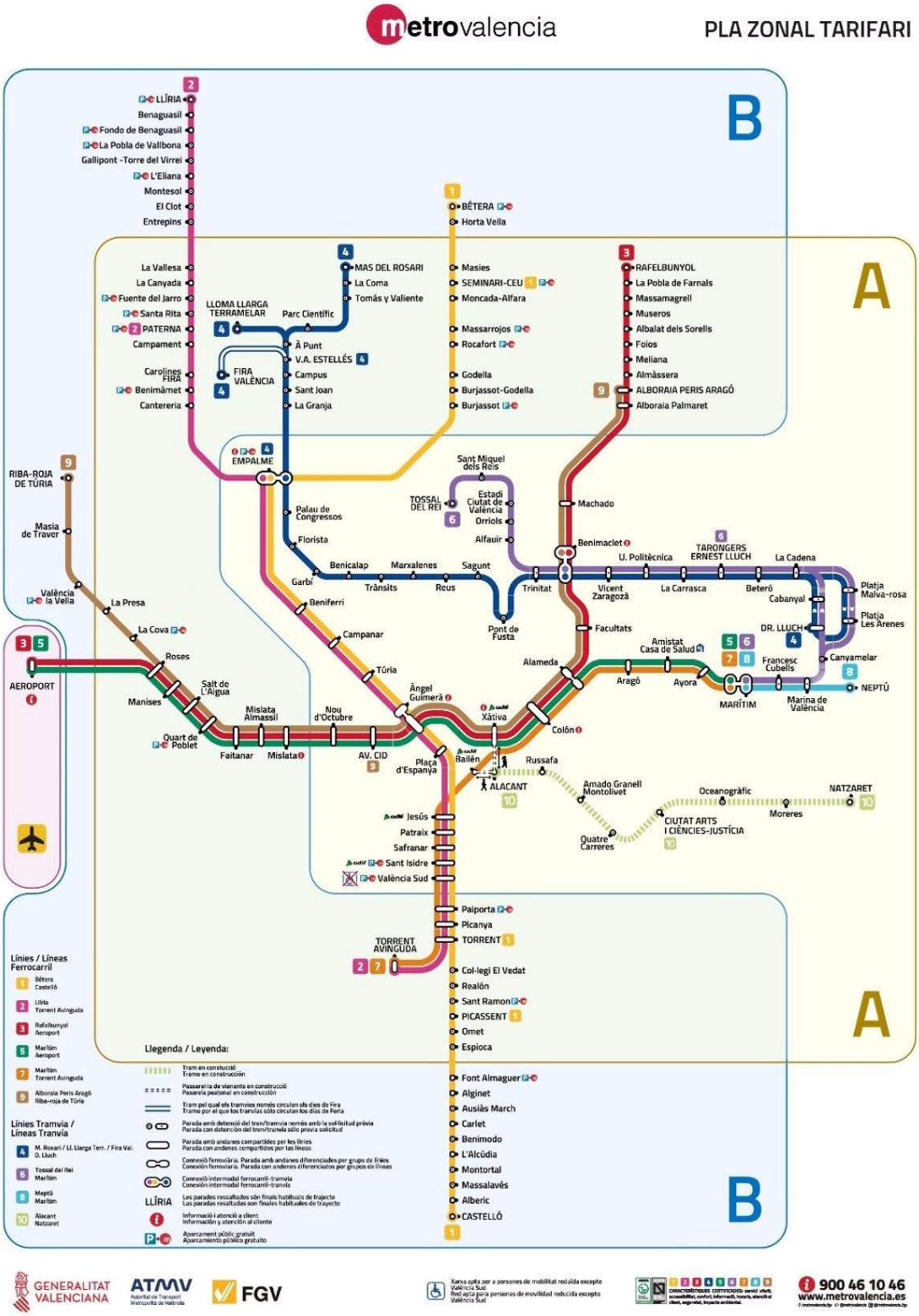

Ilustración 28: Red de metro de Valencia[18]

[18] https://es.m.wikipedia.org/wiki/Archivo:Mapa_metrovalencia.svg

5.1 Diseño y desarrollo de la solución

- **Diseño conceptual:**

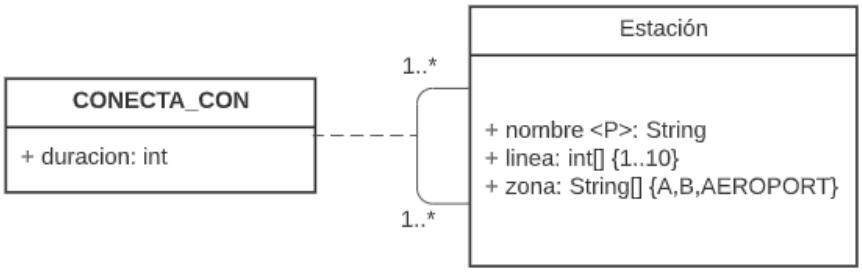

Ilustración 29: Diseño conceptual de la solución

- **Diseño lógico:**
  - Va a existir un único tipo de nodo *Estación* con las siguientes propiedades:
    - *nombre*: de tipo carácter.
    - *zona:* de tipo carácter. Determina el valor de los billetes.
    - *línea*: vector de enteros que indica a qué líneas pertenece la estación.
  - Entre los nodos de este tipo, hay una relación de nombre *Conecta_con* con la propiedad *duración* de tipo entero que permitirá saber los minutos necesarios para ir de una estación a otra.

- **Diseño físico:**

  En cuanto al diseño físico de la solución, se delega esa responsabilidad sobre el sistema gestor de bases de datos Neo4j. Cabe destacar que se genera un índice sobre el atributo *nombre* debido a la frecuencia con el que es consultado. De esta forma, las consultas sobre este campo serán más rápidas.

5.2 Implementación de la solución

Cada nodo de la base de datos representa una parada de metro, con propiedades definiendo la línea o líneas a las que pertenece, el nombre de dicha estación y la zona a la que pertenece (A, B o Aeroport). Estas entidades están conectadas por relaciones con una propiedad que refleja la duración del trayecto entre una estación y otra. De esta forma, podemos realizar consultas en base a esta información incluida en la base de datos.

Para la creación de todos los elementos necesarios para modelar la red, ha sido necesario introducir a mano cada una de las estaciones y relaciones que la componen. Esto es debido a que no existe un documento de dominio público en el que se reflejen la información necesaria para construir la base de datos (duración entre estaciones, líneas de cada estación, etc.).

Un ejemplo de línea que genera un nodo (estación) con los datos previstos es la siguiente:

```
CREATE (E1:Estacion {nombre:"Horta vella", línea: 1, zona:"B"})
```

Para simplificar el proceso de poblado de la base de datos, se ha procedido a crear los nodos y relaciones entre los mismos en un solo comando CREATE. Esto es posible de realizar mediante la creación de un nodo y la referencia al mismo a la hora de crear relaciones. Por ejemplo:

```
CREATE (E1:Estacion {nombre:"Bètera", línea: 1, zona:"B"})-[:CONECTA_CON {duracion: 1}]
        ->(E2:Estacion {nombre:"Horta vella",línea: 1, zona:"B"})
        (E2)-[:CONECTA_CON {time:3}]->(E3:Estacion {nombre:"Masies", línea: 1, zona:"A"})
```

La ejecución de la línea anterior genera las estaciones (nodos) de Bétera, Horta vella y Masies, con su información correspondiente sobre las líneas y zonas a las que pertenecen. Adicionalmente, se establecen los enlaces entre nodos reflejando la duración del trayecto entre estaciones. De esta forma, se logra poblar la base de datos de forma eficiente.

Una vez completado el script, se procede a ejecutarse en el input habilitado en la aplicación. Así, la red generada tiene el siguiente aspecto:

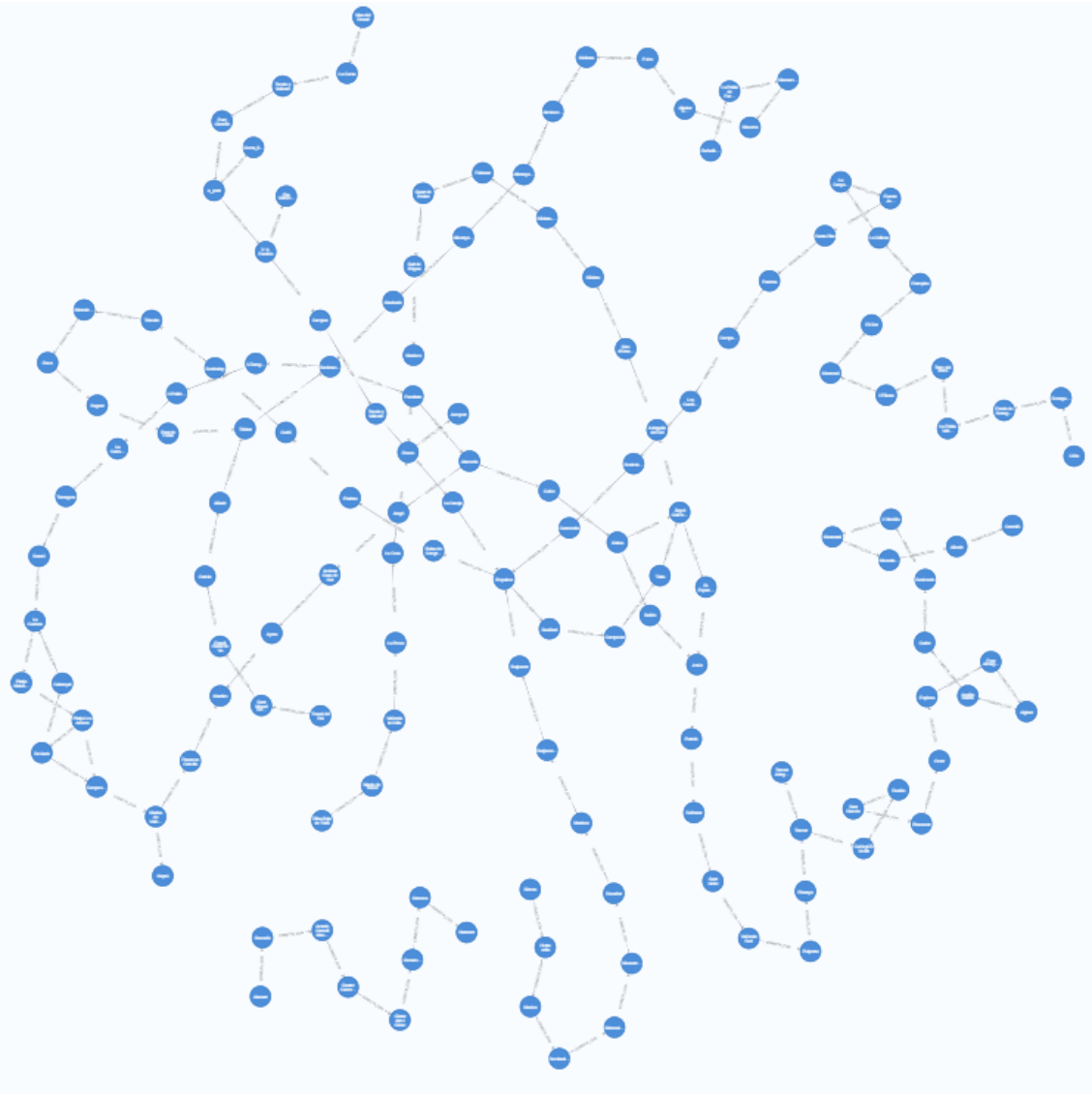

Ilustración 30: Red de metro de Valencia en Neo4j

El total de nodos generados es de 143, enlazados por 146 relaciones.

Neo4j, además de generar el grafo para que sea posible visualizar de forma gráfica los datos introducidos, proporciona cada uno de los nodos de la base de datos en formato JSON:

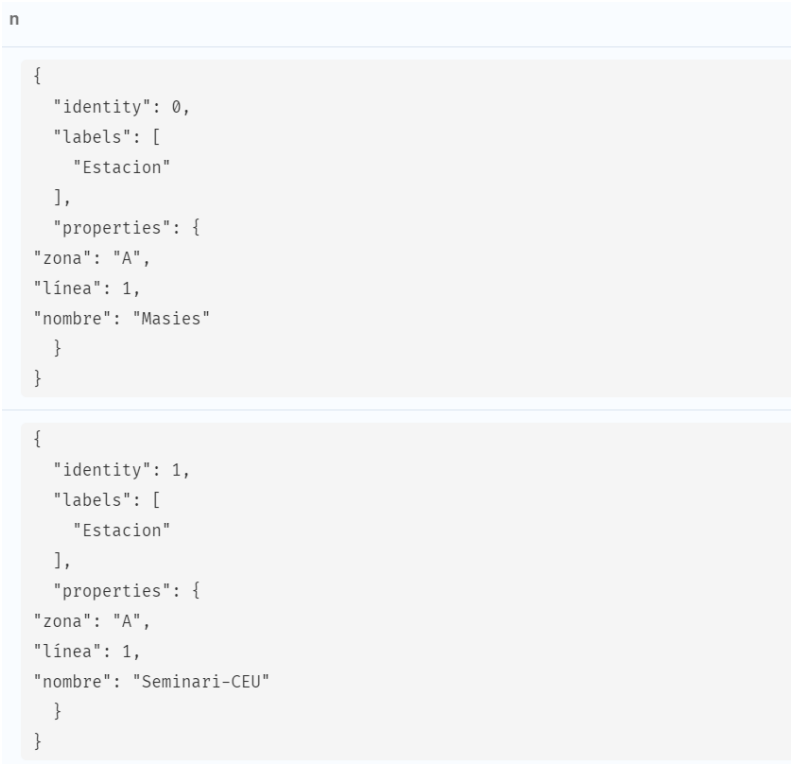

Ilustración 31: Datos de la red de metro en formato JSON

Por último, también es posible obtener información de las consultas ejecutadas. Por ejemplo, al ejecutar "MATCH (n) RETURN n" para obtener todo el grafo, Neo4j genera la siguiente información en base al comando ejecutado y la respuesta obtenida:

Summary:


```
{
  "query": {
    "text": "match (n) return n",
    "parameters": {}
  },
  "queryType": "r",
  "counters": {
    "_stats": {
      "nodesCreated": 0,
      "nodesDeleted": 0,
      "relationshipsCreated": 0,
      "relationshipsDeleted": 0,
      "propertiesSet": 0,
      "labelsAdded": 0,
      "labelsRemoved": 0,
      "indexesAdded": 0,
      "indexesRemoved": 0,
      "constraintsAdded": 0,
      "constraintsRemoved": 0
```

```
  },
  "_systemUpdates": 0
 },
 "updateStatistics": {
  "_stats": {
   "nodesCreated": 0,
   "nodesDeleted": 0,
   "relationshipsCreated": 0,
   "relationshipsDeleted": 0,
   "propertiesSet": 0,
   "labelsAdded": 0,
   "labelsRemoved": 0,
   "indexesAdded": 0,
   "indexesRemoved": 0,
   "constraintsAdded": 0,
   "constraintsRemoved": 0
  },
  "_systemUpdates": 0
 },
 "plan": false,
 "profile": false,
 "notifications": [],
 "server": {
  "address": "localhost:7687",
  "version": "Neo4j/4.4.3",
  "agent": "Neo4j/4.4.3",
  "protocolVersion": 4.4
 },
 "resultConsumedAfter": {
  "low": 7,
  "high": 0
 },
 "resultAvailableAfter": {
  "low": 24,
  "high": 0
 },
 "database": {
  "name": "neo4j"
 }
}
```


Response:


```
[
 {
  "keys": [
   "n"
  ],
```

```
 "length": 1,
 "_fields": [
  {
   "identity": {
    "low": 0,
    "high": 0
   },
   "labels": [
    "Estacion"
   ],
   "properties": {
    "zona": "B",
    "línea": {
     "low": 1,
     "high": 0
    },
    "nombre": "Horta vella"
   }
  }
 ],
 "_fieldLookup": {
  "n": 0
 }
},
{
 "keys": [
  "n"
 ],
 "length": 1,
 "_fields": [
  {
   "identity": {
    "low": 1,
    "high": 0
   },
   "labels": [
    "Estacion"
   ],
   "properties": {
    "zona": "A",
    "línea": {
     "low": 1,
     "high": 0
    },
    "nombre": "Masies"
   }
  }
```

```
  ],
  "_fieldLookup": {
    "n": 0
  }
 },
 {
  "keys": [
    "n"
  ],
  "length": 1,
  "_fields": [
    {
      "identity": {
        "low": 2,
        "high": 0
      },
      "labels": [
        "Estacion"
      ],
      "properties": {
        "zona": "A",
        "línea": {
          "low": 1,
          "high": 0
        },
        "nombre": "Seminari-CEU"
      }
    }
  ],
  "_fieldLookup": {
    "n": 0
  }
 },
```


…

Las consultas más frecuentes sobre una base de datos de este tipo estarían relacionadas con el camino más corto para llegar de una estación a otra. Para ello, Neo4j dispone de una serie de *pugins* que posibilitan la ejecución de consultas basadas en algoritmos disponibles. Dos de estas librerías de especial utilidad son:

- ***APOC***: compuesta por unas 450 funciones aplicables en diferentes tareas en áreas como la integración de datos, algoritmos sobre grafos y conversión de datos.
- ***Graph Data Science Library***: proporciona capacidades analíticas extendidas centradas en los algoritmos sobre grafos. Esta librería incluye algoritmos para centralidad, similitud de nodos, búsqueda de caminos, y predicción de enlaces, así como procedimientos de catálogos de grafo diseñados para soportar flujos de datos

de ciencias de datos y *machine learning* sobre los grafos. Todas las operaciones están diseñadas para gran escala y paralelización.

Para tratar de encontrar el camino más corto entre dos grafos (estaciones) como "Parc Científic" y "Cantereria" podríamos emplear la siguiente consulta:

```
MATCH (E1:Estacion {nombre:'Parc Científic'})
MATCH(E2:Estacion {nombre:'Cantereria'})
MATCH path = shortestPath((E1)-[*]-(E2))
RETURN path
```

El resultado de ejecutar la misma sería:

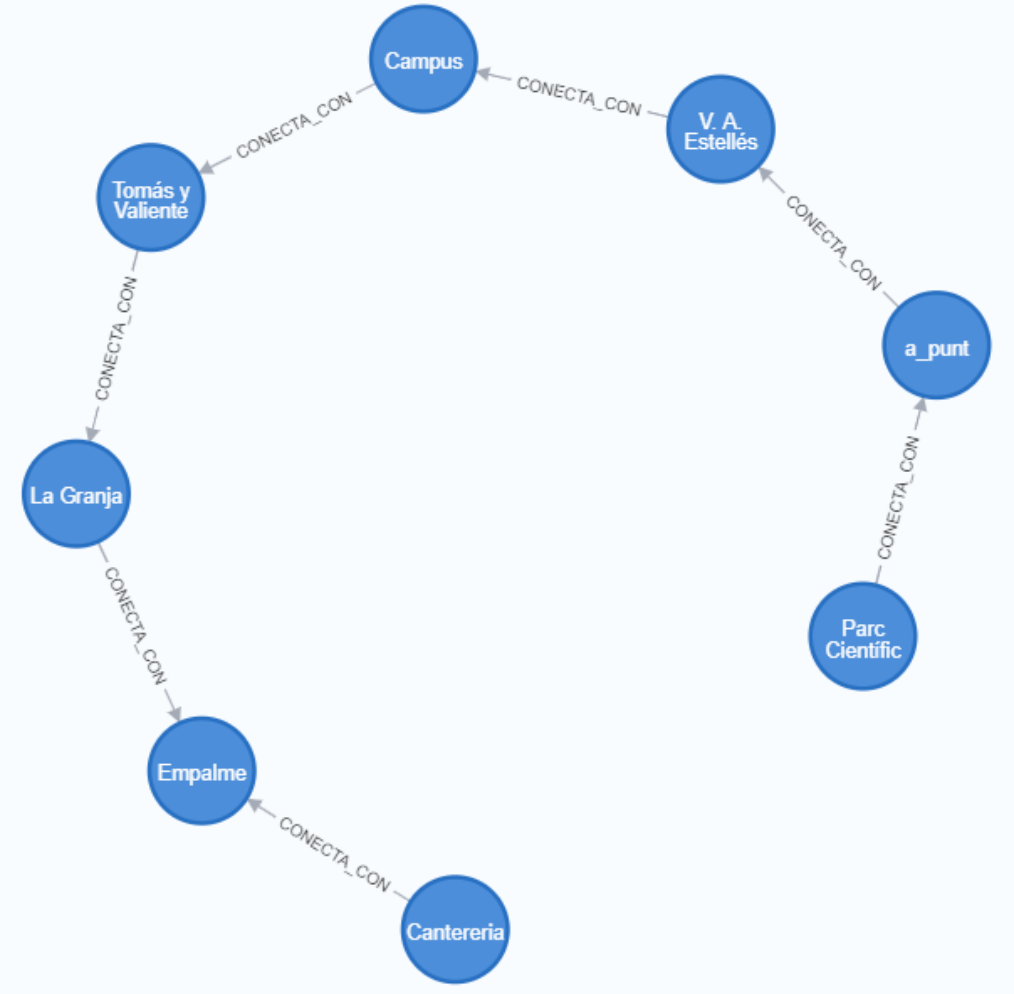

Ilustración 32: Camino más corto entre dos estaciones de la red

Cabe destacar que la dirección de las relaciones en este caso es irrelevante, ya que las líneas circulan en ambos sentidos, por lo que no afecta al resultado de la consulta.

Ejecutar la anterior consulta tiene un inconveniente en el caso de uso presentado, ya que el resultado no tendrá en cuenta los pesos establecidos en las relaciones, por lo que devolverá el camino con menor número de nodos sin tener en cuenta los valores introducidos en dichos pesos. Por tanto, el algoritmo adecuado para nuestro caso particular se trata de Dijkstra. Afortunadamente, se encuentra disponible en la librería de Graph Data Science anteriormente presentada.

El algoritmo de **Dijkstra**, desarrollado por Edsger Dijkstra en el año 1959, tiene como objetivo determinar el camino más corto dado un vértice (nodo) origen al resto de vértices en un grafo con pesos en cada arista (relación). Por tanto, se ajusta perfectamente a la consulta que queremos realizar.

Existen múltiples variantes para este algoritmo. La más común fija un único nodo origen y encuentra los caminos más cortos desde dicho nodo al resto de nodos en el grafo, generando un árbol de caminos más cortos. También se puede utilizar para encontrar el camino más corto desde un solo nodo a un único nodo destino deteniendo el algoritmo una vez que el camino más corto al destino se determina.

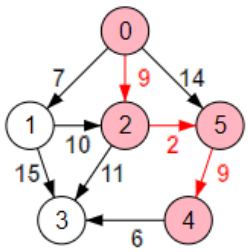

Ilustración 33: Ejemplo de implementación del algoritmo de Dijkstra[19]

El procedimiento empleado por el algoritmo es el siguiente:

1. Seleccionar vértice de partida, es decir un origen.
2. Marcar el punto de partida como el punto de inicio.
3. Determinar los caminos especiales (camino que sólo puede trazarse a través de los nodos o vértices ya marcados) desde el nodo de partida, es decir, el de inicio.
4. Para cada nodo no marcado, se debe determinar si es mejor usar el camino especial antes calculado o si es mejor usar el nuevo camino especial que resulte al marcar este nuevo nodo.
5. Para seleccionar un nuevo nodo no marcado como referencia, deberá tomarse aquel cuyo camino especial para llegar a él es el mínimo. Por ejemplo, si anteriormente se ha marcado el nodo o vértice 2, el cual tiene dos nodos adyacentes 3 y 4 cuyo peso en la arista corresponde a 10 y 5 respectivamente, se tomará como nuevo nodo de partida el 4, ya que el peso de la arista o camino es menor.
6. Cada camino mínimo corresponde a la suma de los pesos de las aristas que forman el camino para ir del nodo principal al resto de nodos, pasando únicamente por caminos especiales, es decir nodos marcados.

En código, sería de la siguiente manera [Hodler19]

[19] https://en.wikipedia.org/

```
//Declarando variables
#define MAX_NODOS 1024 /* número máximo de nodos */
#define INFINITO 1000000000 /* un número mayor que cualquier ruta máxima */
#include<iostream>
using namespace std;
int n,i,k,minimo, dist[MAX_NODOS][MAX_NODOS]; /* dist[i][j] es la distancia de i a j */

struct nodo { /* Indica eL estado del nodo,la ruta y quien lo precede a dicho nodo */
    int predecesor; /* nodo previo */
    int longitud; /* longitud del origen a este nodo */
    bool etiqueta;  /*verdadero para un nodo permanente y falso para nodo tentativo*/
} nodo[MAX_NODOS];

void inicializacion(){
    for (auto p = &nodo[0]; p < &nodo[n]; p++) { /* estado de inicialización*/
        p->predecesor = -1;
        p->longitud = INFINITO;
        p->etiqueta = false;
    }
}
void relajar(){
    for (int i = 0; i <n; i++){ /* este grafo tiene n nodos */
        if (dist[k][i] != 0 && nodo[i].etiqueta == false) {
            if (nodo[k].longitud + dist[k][i] < nodo[i].longitud) {
                nodo[i].predecesor = k;
                nodo[i].longitud = nodo[k].longitud + dist[k][i];
            }
        }
    }
}
void extraer_minimo(){  /* Encuentra los nodo etiquetados tentativamente y determina el menor entre estos nodos tentativos. */
    k = 0;
    minimo = INFINITO;
    for (i = 0; i < n; i++){
        if (nodo[i].etiqueta == false && nodo[i].longitud < minimo) {
            minimo = nodo[i].longitud;
            k = i;
        }
    }
void camino_corto(int s, int t, int camino[]) {
    inicializacion();
    nodo[t].longitud = 0; nodo[t].etiqueta = true;
    k = t; /* k es el nodo de trabajo inicial */
    do{ /* ¿hay una ruta mejor desde k? */
        relajar();
        extraer_minimo();
        nodo[k].etiqueta = true;
    } while (k != s);
    /* Copia la ruta en el arreglo de salida y procede a ir imprimiendolo. */
    i = 0; k = s;
    cout<< "La ruta es: ";
    do {
        cout<< k<< " ";
        camino[i] = k;
        k = nodo[k].predecesor;
        i++;
    } while (k >= 0);
        cout <<"La ruta minima es: "<<minimo<<endl ;
}
int main(){
    int nodo_final,nodo_inicial,arista;
    //solicita o ingresa directamente los valores de nodo_final,nodo_inicial
    //Llenar de 0 la matriz
    for (int i=0; i<n; i++){
        for( int j=0; j<n; j++){
            dist[i][j]=0;
        }
    }
    // Llenar la matriz dist[i][j]
    /*.........................
    .........................*/
    //Por ultimo llamar a la función camino corto
    //camino_corto(nodo_final,nodo_inicial,arista);

    return 0;
}
```

Ilustración 34: Algoritmo de Dijkstra en código

Empleando Dijkstra, la consulta a ejecutar sería la siguiente:

```
MATCH (E1:Estacion {nombre:'Parc Científic'})
MATCH (E2:Estacion {nombre:'Cantereria'})
CALL apoc.algo.dijkstra(E1, E2, 'CONECTA_CON, 'duracion') YIELD path, weight
RETURN path, weight
```

Al ejecutarla, el resultado obtenido es el siguiente:

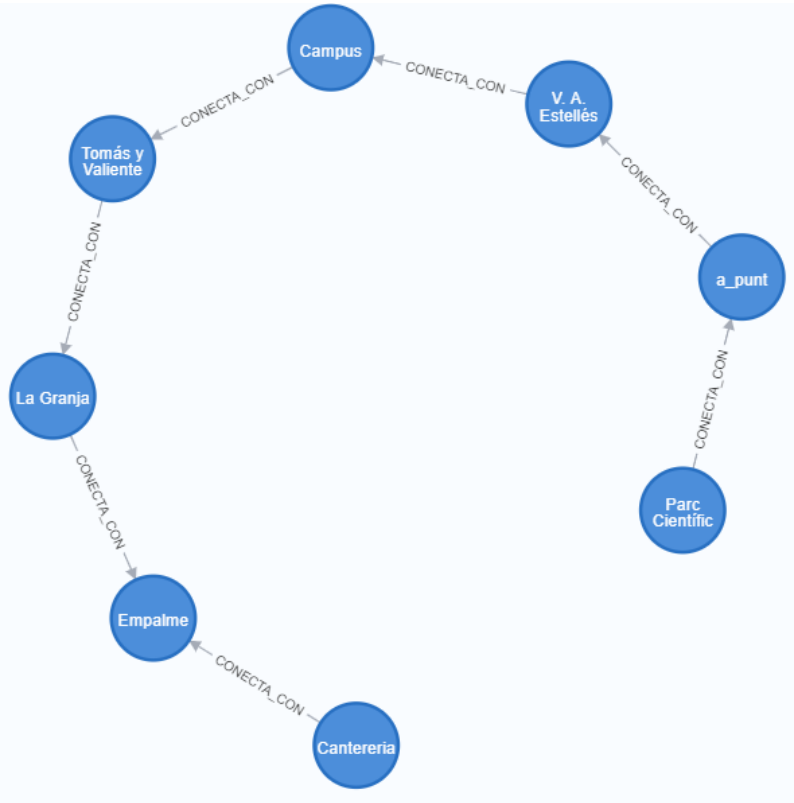

Ilustración 35: Resultado de una consulta empleando Dijkstra en el grafo

Casualmente, el camino devuelto es el mismo que en la consulta anterior. Sin embargo, en este caso se ha tenido en cuenta la duración del trayecto entre cada estación. De hecho, Neo4j presenta el peso total, es decir, la duración, del trayecto completo desde la estación origen al destino:

"path"	"weight"
[{"zona":"A","nombre":"Parc Científic","línea":4},{"duracion":1},{"zona":"A","nombre":"a_punt","línea":4},{"zona":"A","nombre":"a_punt","línea":4},{"duracion":2},{"zona":"A","nombre":"V. A. Estellés","línea":4},{"zona":"A","nombre":"V. A. Estellés","línea":4},{"duracion":1},{"zona":"A","nombre":"Campus","línea":4},{"zona":"A","nombre":"Campus","línea":4},{"duracion":1},{"zona":"A","nombre":"Tomás y Valiente","línea":4},{"zona":"A","nombre":"Tomás y Valiente","línea":4},{"duracion":1},{"zona":"A","nombre":"La Granja","línea":4},{"zona":"A","nombre":"La Granja","línea":4},{"duracion":2},{"zona":"A","nombre":"Empalme","línea":[1,2,4]},{"zona":"A","nombre":"Empalme","línea":[1,2,4]},{"duracion":3},{"zona":"A","nombre":"Cantereria","línea":2}]	11.0

Ilustración 36: Datos obtenidos tras emplear Dijkstra

También es posible realizar consultas sobre el número de nodos (máximo, mínimo o cantidad exacta) que se desear recorrer. Por ejemplo, si queremos conocer las estaciones que desde Alameda se encuentran a 3 paradas de distancia como máximo, ejecutamos:

```
MATCH p = (E1:Estacion {nombre:'Alameda'})-[:CONECTA_CON *0..3]-(E2:Estacion)
CALL apoc.algo.dijkstra(E1, E2, 'CONECTA_CON, 'duracion') YIELD path
RETURN path
```

Lo que tendría como resultado:

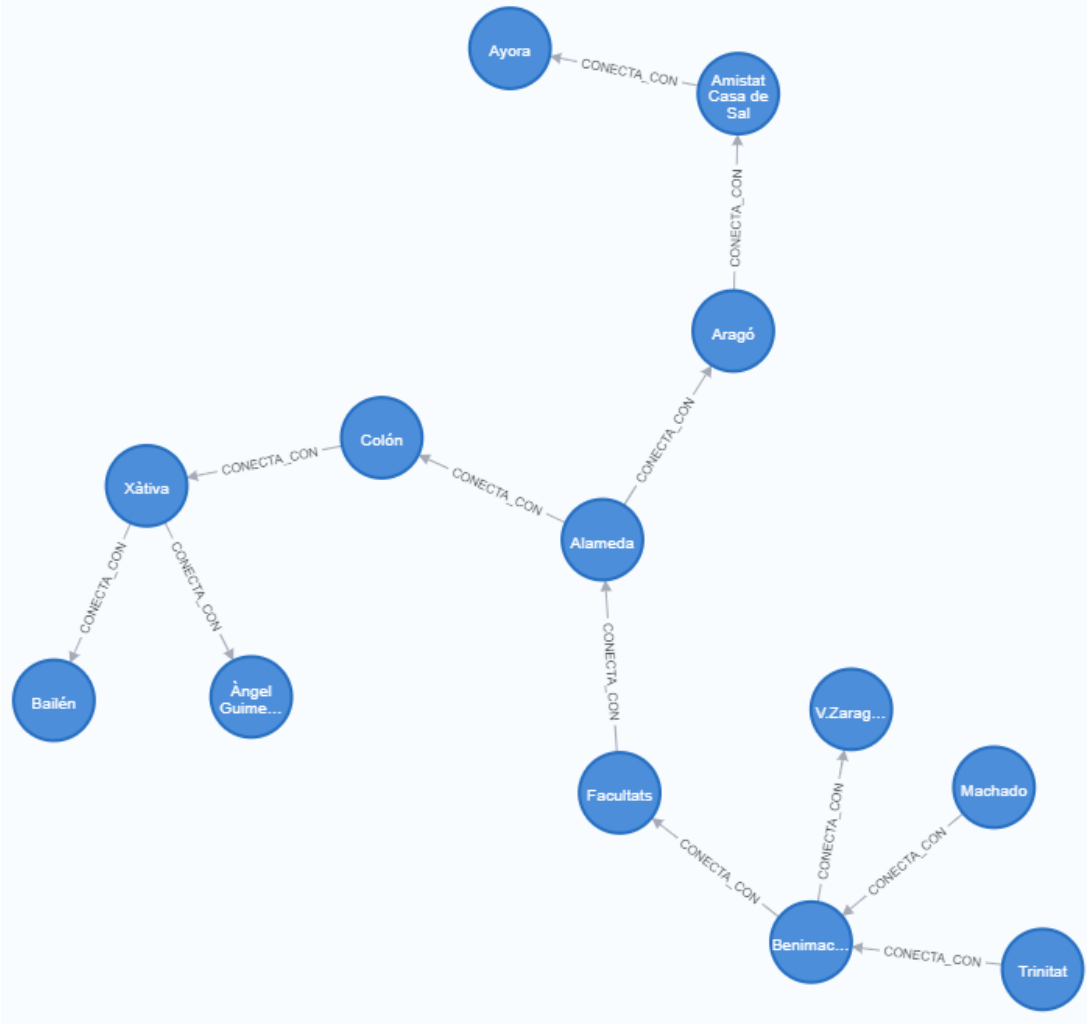

Ilustración 37: Resultado de consultar las estaciones a una determinada distancia de Alameda

Así mismo, es posible añadir condiciones a la consulta en base a otras propiedades que se deben tener en cuenta. Por ejemplo, es frecuente querer encontrar la ruta entre estaciones que cumplan una determinada propiedad como la zona a la que pertenecen. Por ejemplo, para obtener los caminos mediante estaciones únicamente en zona "A" que conectan "Palau de Congressos" y "Garbí" ejecutamos:

```
MATCH path = (E1 {nombre:'Palau de Congressos'})-[*]-(E2 {nombre:'Garbí'})
WITH nodes(path) AS ns
WHERE ALL(n IN ns where n.zona='A')
RETURN ns
```

El resultado sería:

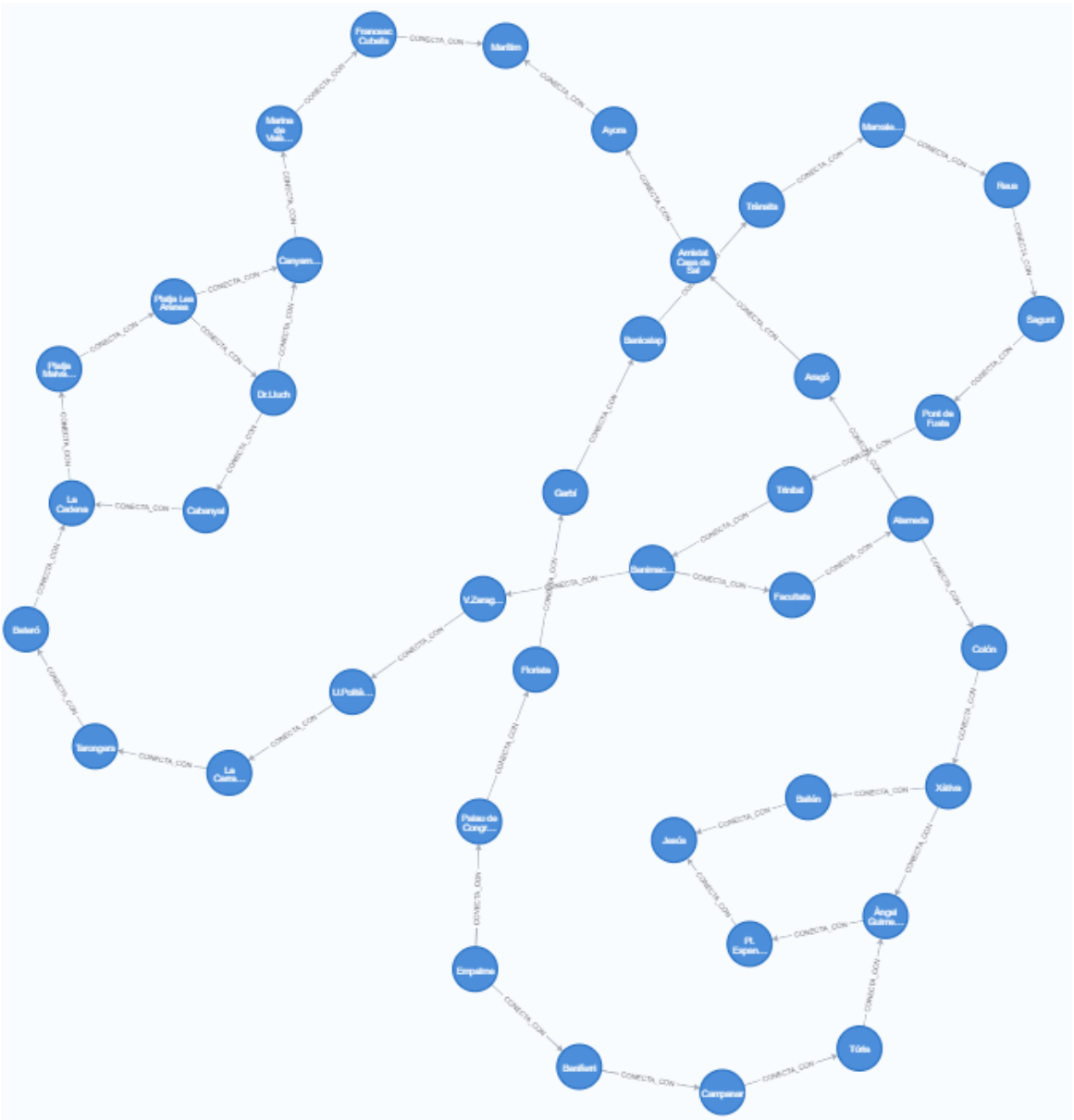

Ilustración 38: Resultado de consultar las estaciones en zona A que conectan Palau de Congressos y Garbí

Es recomendable combinar esta consulta con el algoritmo de Dijkstra para obtener el camino más corto de una estación a otra donde los nodos de este cumplan una determinada condición. El ejemplo anterior combinado con Dijkstra resultaría:

```
MATCH (E1:Estacion { nombre: 'Palau de Congressos' })
MATCH (E2:Estacion { nombre: 'Garbí' })
CALL apoc.algo.dijkstra(E1, E2, 'CONECTA_CON', 'duracion') YIELD path
WITH nodes(path) AS ns
WHERE
  ALL(n IN ns where n.zona='A')
RETURN ns
```

La ejecución de la consulta tiene como resultado:

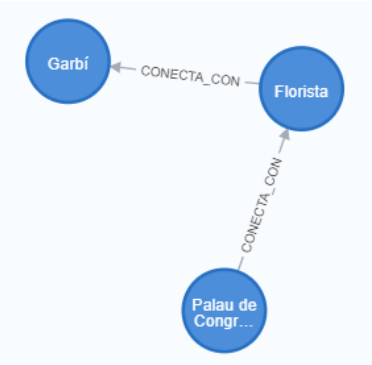

Ilustración 39: Resultado de consulta con Dijkstra

Existen dos opciones a elegir a la hora de analizar una consulta teniendo en cuenta su plan de ejecución:

- *EXPLAIN*: En el caso de que se quiera consultar el plan de ejecución, pero no ejecutar la consulta, se debe incorporar el comando EXPLAIN al principio. La sentencia siempre devolverá un resultado vacío y no realizará cambios en la base de datos.
- *PROFILE*: Si el objetivo es ejecutar la sentencia y ver qué operadores están realizando la mayor parte del trabajo. Este comando ejecuta la sentencia y analiza cuántas filas pasan a través de cada operador, y cuánto necesita cada operador interaccionar con la capa de almacenamiento para recuperar los datos necesarios. Hay que tener en cuenta que utilizar PROFILE en una consulta aumenta el uso de recursos.

A continuación, vamos a comprobar la relevancia del uso de comandos adecuados en una consulta en cuanto a rendimiento. Para ello, se va a emplear una base de datos con un mayor número de elementos como es la base de datos de películas disponible en la web oficial de Neo4j. Esta base de datos está formada por 171 nodos de diferentes clases y 253 relaciones de diversos tipos.

En primer lugar, para obtener una película la consulta más sencilla y costosa es la siguiente:

```
MATCH (M {title:'The Birdcage'}) RETURN M
```

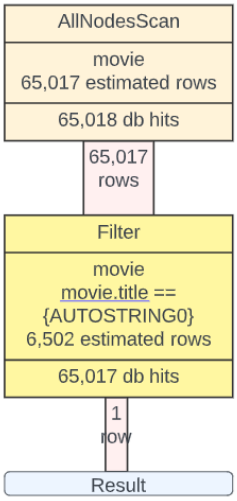

Ilustración 40: Coste de consultar una determinada película

Como muestra la figura, los accesos a la base de datos superan la cantidad de 65.000, una cifra extremadamente alta. Para facilitar la búsqueda de la película en concreto, es aconsejable determinar la clase del modo buscado. En este caso, un nodo de tipo “Movie”:

```
MATCH (M:Movie {title:'The Birdcage'})
RETURN M
```

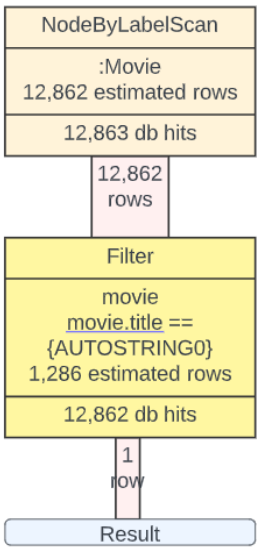

Ilustración 41: Coste de consultar una determinada película filtrando por la clase del nodo buscado

De esta forma, la cantidad de accesos a la base de datos se reduce considerablemente, disminuyendo el tiempo total que supone la ejecución de la consulta. Además del uso de etiquetas como en el caso anterior, la creación de índices sobre el campo que más va a ser buscado facilita también la búsqueda del nodo. Por ejemplo:

```
CREATE INDEX ON :Movie(title)
```

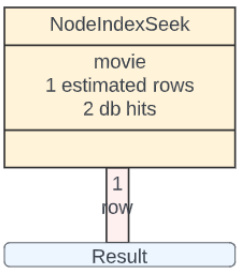

Ilustración 42: Coste de consultar una determinada película empleando índices

Como se puede observar, la cantidad de accesos a la base de datos se ha reducido a un número ínfimo con el uso de buenas prácticas de búsqueda.

Por último, es recomendable devolver únicamente los parámetros que se vayan a necesitar. Es decir, es menos recomendable utilizar:

```
MATCH (a:Actor) RETURN a
```

Que emplear:

```
MATCH (a:Actor) RETURN a.name, a.birthdate, a.height
```

6 Conclusiones

Tras el desarrollo del tema expuesto anteriormente, vamos a concluir el proyecto con una valoración de los objetivos establecidos al comienzo de su elaboración y su consecución.

En primer lugar, se ha logrado realizar una recogida de documentación bastante completa relacionada con las bases de datos NoSQL. Para ello, ha sido necesaria la lectura y estudio de múltiples libros y artículos de divulgación de autores expertos en el tema.

La revisión bibliográfica ha supuesto la mayor parte del tiempo dedicado al desarrollo del proyecto debido a la necesidad de adquirir una base teórica sólida sobre el campo desarrollado en la memoria. Cabe destacar que la mayor parte de la bibliografía existente sobre el tema está redactada en inglés, por lo que han sido necesarias herramientas para poder interpretar algunos fragmentos más técnicos de los textos.

En cuanto al desarrollo concreto de las bases en grafo, la documentación disponible es notablemente más escasa debido a que se trata de un campo más específico y novedoso. Sin embargo, considero que se ha logrado plasmar una explicación completa sobre las bases de esta tecnología, mostrando sus puntos fuertes y características.

En lo referente a la aplicación práctica presentada, el hecho de no disponer de una base de datos pública sobre la que obtener los datos ha aumentado el tiempo de trabajo, especialmente en lo referente a la carga de dichos datos. No obstante, la fase de explotación he supuesto un tiempo de trabajo más corto gracias, en parte, a la comunidad de desarrolladores implicados con la plataforma de Neo4j y Cypher.

Una vez finalizada la memoria, podemos afirmar que mediante la elaboración del proyecto se ha logrado adquirir conocimientos sobre el campo de las bases de datos NoSQL. Además, el desarrollo del caso práctico ha permitido conocer las diferentes fases de la implementación de una base de datos en grafo, así como la aplicación sobre un caso real.

Desde mi punto de vista, considero que he podido adquirir de forma autodidacta los conocimientos necesarios para redactar una memoria sobre el tema a desarrollar y aplicar esos conocimientos sobre un caso práctico de forma exitosa. Sobre las bases de datos en grafo, me gustaría destacar que, a pesar de presentar múltiples ventajas sobre el modelo relacional, es un campo sobre el que la mayoría de los profesionales informáticos no tienen conocimiento. A mi parecer, se trata de una tecnología más que interesante que, además, cuenta con una gran comunidad de desarrolladores a sus espaldas.

7 Bibliografía

[Abadi12] Abadi, D. J., Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. Computer 2. 2012

[Angles12] Angles, R. A comparison of current graph database models. 28th International Conference on Data Engineering Workshops. 2012

[Angles] Angles, R. A Comparison of Current Graph Database Models.

[Avrilia12] Avrilia Floratou, N. T. Can the Elephants Handle the NoSQL Onslaught? 2012

[Beeri90] Beeri, C. A formal approach to objet-oriented databases. Data & Knowledge Engineering. 1990

[Berge73] Berge, C. Graphs and hypergraphs. Amsterdam: North-Holland Publishing Company. 1973

[Brewer12] Brewer, E. A. CAP twelve years later: How the "rules" have changed. Computer. 2012

[Bryce] Bryce Merkl Sasaki, J. C. Graph Databases for Begginers. neo4j.

[Brzezinski04] Brzezinski, J. From session causality to causal consistency. PDP. 2004

[Casamayor21] Casamayor, J. C. Tema 4: Diseño de Almacenes de Datos. En Sistemas de Información Estratégicos Parte I: Almacenes de Datos (4 ed.). 2021

[Casamayor21] Casamayor, J. C. Tema 5: Mantenimiento de Almacenes de Datos. Herramientas ETL. En Sistemas de Información Estratégicos Parte I: Almacenes de Datos (5 ed.). 2021

[Cattell11] Cattell, R. Scalable SQL and NoSQL data stores. ACM SIGMOD. 2011

[Chang08] Fay Chang, J. D. Bigtable: A distributed storage system for structured data. ACM. 2008

[Gutierrez08] Gutierrez, R. A. Survey of graph database models. ACM Computing Surveys. 2008

[Hodler19] Hodler, M. N. Graph Algorithms: Practical examples in Apache Spark & Neo4j. O'REILLY. 2019

[Robinson15] Ian Robinson, J. W. Databases: new opportunities for connected data. O'REILLY. 2015

[Jiaqing14] Jiaqing Du, C. I. Gentletrain: Vheap and scalable causal consistency with physical clocks. Proceedings of the ACM Symposium on Cloud Computing. 2014

[Kapali76] Kapali P. Eswaran, J. N. notions of consistency and predicate locks in a database system. ACM. 1976

[Grolinger13] Katarina Grolinger, W. A. Data management in cloud environments: NoSQL and NewSQL data stores. Sipringer. 2013

[Kemper15] Kemper, C. Beginning Neo4j. Apress. 2015

[Laudon96] Laudon, K., & Laudon, J. Administracion de los sistemas de informacion. México: Prentice-Hall Hispanoamericana. 1996

[Mota21] Mota, L., & Casamayor, J. C. 1: Los Sistemas de Información Estratégicos. En Sistemas de Información Estratégicos. Parte I: Almacenes de Datos (1 ed.). 2021

[Mota21] Mota, L., & Casamayor, J. C. Tema 2: Introducción a los Almacenes de Datos. En Sistemas de Información Estratégicos Parte I: Almacenes de Datos (2 ed.). 2021

[Mota21] Mota, L., & Casamayor, J. C. Tema 3: Explotación de Almacenes de Datos: herramientas OLAP. En Sistemas de Información Estratégicos Parte I: Almacenes de Datos (3 ed.). 2021

[Mota19] Mota, L., & Vicent, M. Tema 2: Procesamineto de transacciones y mantenimiento de la integridad. En Diseño y Gestión de bases de datos (2 ed.). 2019

[Mota19] Mota, L., & Vicent, M. Tema 7: Normalización. En Diseño y gestión de bases de datos (7 ed.). 2019

[Naeem20] Naeem, T. Astera. Obtenido de https://www.astera.com/es/type/blog/data-warehouse-concepts. 2020

[Naeem20] Naeem, T. Conceptos de Data Warehouse: enforque de Kimball vs Imon. Obtenido de Astera: https://www.astera.com/es/type/blog/data-warehouse-concepts/ . 2020

[Nazemi20] Nazemi, E. CONST: Continuous online NoSQL schema tuning. WILEY. 2020

[Atzeni17] Paolo Atzeni, F. B. Data Modeling in the NoSQL World. HAL. 2017

[Pokorný15] Pokorný, J. Graph Databases: Their Power and Limitations. IFIP. 2015

[Angles16] Renzo Angles, M. A. Foundations of modern graph query languages. arXiv preprint arXiv. 2016

[Robinson] Robinson, J. W. The Top 5 Use Cases of Graph Databases. neo4j.

[Rumbaugh98] Rumbaugh, J., Jacobson, I., & Booch, G. The Unified Modeling Language Reference Manual. Massachusetts: Addison Wesley Logman. 1998

[Talend22] Talend. What is Database Integration? Obtenido de https://www.talend.com/resources/what-is-database-integration. 2022

[Berners-Lee01] Tim Berners-Lee, J. H. The semantic web. Scientific American. 2001

[Webber12] Webber, J. A programmatic introduction to Neo4j. 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity. ACM. 2012

[Sun15] Wen Sun, A. F. SQLGraph: An efficient relational-databased property graph store. ACM SIGMOD International Conference on Management of Data. ACM. 2015

[Chen13] Zhikun Chen, S. Y. Hybrid range consistent hash partitioning strategy. 2013

[Biggs86] Norman Biggs, E. Keith Lloyd, Robin J. Wilson. Graph Theory, 1736-1936. 1986

8 Objetivos de desarrollo sostenible

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
1. **Fin de la pobreza.**				X
2. **Hambre cero.**				X
3. **Salud y bienestar.**		X		
4. **Educación de calidad.**				X
5. **Igualdad de género.**				X
6. **Agua limpia y saneamiento.**				X
7. **Energía asequible y no contaminante.**		X		
8. **Trabajo decente y crecimiento económico.**				X
9. **Industria, innovación e infraestructuras.**	X			
10. **Reducción de las desigualdades**.		X		
11. **Ciudades y comunidades sostenibles**.	X			
12. **Producción y consumo responsables.**		X		
13. **Acción por el clima.**				X
14. **Vida submarina.**				X
15. **Vida de ecosistemas terrestres.**				X
16. **Paz, justicia e instituciones sólidas.**				X
17. **Alianzas para lograr objetivos.**				X

Tabla 2: Objetivos de desarrollo sostenible sobre el proyecto

Los Objetivos de Desarrollo Sostenible que se pueden relacionar con este proyecto son especialmente el de tecnología, innovación e infraestructuras, y ciudades y comunidades sostenibles en la medida en que permite identificar aquellas fortalezas y escaseces que, en estos ámbitos, presenta cada país.

Al desarrollar un modelo de gestión de datos, se permite optimizar el consumo energético y mejorar la red de infraestructuras de las ciudades. Por ejemplo, en el caso práctico presentado de Metrovalencia, es posible obtener datos sobre las rutas más cortas entre estaciones, reduciendo así el consumo de combustible, implicando a la eficiencia energética.

En este sentido, aunque la implementación de bases de datos basadas en el modelo descrito en la memoria no implica la mejora directa de la sostenibilidad de las ciudades, se podría emplear en ciertos ámbitos donde la duración de las consultas es crítica.

Al estar directamente relacionado con la mejora de las infraestructuras presentes en las ciudades, contribuye al logro de una mayor eficiencia energética y comunidades sostenibles. Además, la mejora de los medios de transporte puede suponer un abaratamiento de sus costes pudiendo estar así al alcance de poblaciones con menos recursos.

En relación con el objetivo de producción y consumo responsables, el modelado de los datos en grafo permite observar de forma detallada las relaciones entre la población y sus intereses. De esta forma, se podrían establecer unas conductas comunes observadas en la ciudadanía, determinando las fuentes de producción y consumo menos sostenibles y prescindibles.

En ese sentido, prácticamente cualquier ámbito de la sociedad puede verse mejorado con una gestión eficiente de la información, analizando grandes volúmenes de datos para obtener estadísticas sobre hábitos, conductas y procedimientos poco sostenibles.

En el sentido del almacenamiento de los datos, antiguamente se realizaba un registro sobre papel de la información que se quería persistir. Con el avance de las técnicas de almacenamiento de datos, es posible guardar esa información en dispositivos de tamaño cada vez más reducido. Mediante las técnicas de almacenamiento descritas en la memoria, se permite disponer de los datos mediante su persistencia en archivos con un tamaño reducido en comparación con los existentes hace unas décadas. De esta forma, se logra una optimización de los recursos necesarios para el almacenamiento de los datos, tales como materias primas y dispositivos electrónicos.

En relación con el objetivo de salud y bienestar, la disposición de una base de datos con información relevante sobre los hábitos de la población permite el análisis de las conductas presentes en la población, pudiendo favorecer los hábitos saludables para lograr un mayor bienestar y salud en la ciudadanía. En ese sentido, el modelado de los datos puede facilitar la tarea del análisis de esa información, así como reducir los tiempos de carga y explotación de los datos en una base de datos.

En la sociedad actual, la gestión de la información supone un punto de inflexión sobre el desarrollo de una comunidad. Un uso eficiente de la información puede conllevar a mejoras en infraestructura, sostenibilidad y bienestar, logrando así múltiples objetivos de desarrollo sostenible.