Persistencia Políglota
Caso de estudio: MongoDB y Neo4j

Jefferson Santiago
Escuela de Computación
Facultad de Ciencias
Universidad Central de Venezuela
Caracas, Venezuela
Email: jefri.26.4@gmail.com

Jesús Arévalo
Escuela de Computación
Facultad de Ciencias
Universidad Central de Venezuela
Caracas, Venezuela
Email: jaas1710@gmail.com

Luinel Andrade
Escuela de Computación
Facultad de Ciencias
Universidad Central de Venezuela
Caracas, Venezuela
Email: luinel1393@gmail.com

***Resumen*—Con el auge de los sistemas de bases de datos NoSQL los cuales implementan modelos de datos diferentes al relacional como son las bases de datos documentales o de grafos, ha surgido el concepto de Persistencia Políglota. Ésta sostiene que debido a la gran variedad y cantidad de representación de los datos, y los diversos servicios que pueden dar las aplicaciones hoy en día; es necesario el uso de más de un tipo de sistema de almacenamiento para ser capaz de cubrir de forma eficiente todas las necesidades de la aplicación que use dicho sistema.**
En este trabajo se busca dar una idea general de las Aplicaciones de Persistencia Políglota describiendo las posibles arquitecturas que hacen uso de las bases de datos NoSQL y su funcionamiento, se estudian algunos casos de éxito y se lleva a cabo un caso de estudio usando MongoDB y Neo4j.

***Index Terms*—persistencia, políglota, NoSQL, MongoDB, N eo4j.**

I. Introducción

En los últimos años debido al uso de Internet y de nuevoas tecnologías ha aumentado la necesidad de manejar datos en volúmenes mayores a los que tradicionalmente manejan las bases de datos tradicionales, con una alta disponibilidad y escalabilidad. Es por esto que surgen las bases de datos NoSQL cuyo uso suele ser determinado por la necesidad de abarcar un fenomeno existente mediánte un modelo de datos apropiado, generalmente fuera del modelo relacional, sin embargo, la variedad de modelos de datos y las diferentes características técnicas entre ellas han requerido nuevos enfoques que deben establecer criterios que permitan conocer a que tareas se adaptan de forma más adecuada para conseguir mejor desempeño y productividad. La Persistencia Políglota se centra en el hecho de que las aplicaciones se pueden beneficiar del uso de una base de datos para cada dominio o componente de la aplicación, de esta forma aprovechar el enfoque de ”usar la mejor herramienta para el trabajo” en cada uno de los paradigmas que se necesiten contemplar, sin embargo, refiere una dificultad a nivel de integracion de las bases de datos.

II. Modelos de datos semiestructurados

Las bases de datos NoSQL han adoptado modelos de datos de esquema flexible que buscan adaptarse a los distintos problemas a resolver. Estos modelos se han denominado semiestructurados, debido a que aunque tienen una estructura implicita esta no es tan regular que pueda ser gestionada.

Entre los modelos de datos semiestructurados que se han implementado se tienen

- **Clave valor:** Es como un mapa o diccionario, donde a una clave se le asigna un valor que puede ser cualquier estructura de datos.
- **Familia de Columnas:** Modelo propuesto por Goggle para su base de datos BigTable [3] basada en el concepto de columna y la agrupación de columnas relacionadas en familias de columnas.
- **Documentales:** Compuesto de un conjunto de tuplas clave-valor, donde el valor pueden ser datos atómicos, un arreglo u otras tuplas clave-valor. Son representados en el formato Json [5] el cual es un formato de texto para intercambio de datos estructurados entre lenguajes de programamción.
- **Grafos:** Estructura de datos donde la información se representa como nodos con propiedades y sus relaciones como enlaces [9].

Cómo se puede observar, cada uno de estos modelos representa la información de diferentes maneras, lo cual hace que la aplicación que use este tipo de bases de datos, tenga que adaptarse a dicha representaciones y su forma de acceso a los datos. Esto se hace más complejo si la aplicación usa más de una base de datos NoSQL con modelos o formas de acceso distintos entre si.

III. Persistencia Políglota

El término Aplicación de Persistencia Políglota se basa en la idea de que las aplicaciones aprovechen las facilidades que cada modelo de datos aporta a la solución de un problema o ejecución de una tarea, que llevaría al uso de más de un sistemas de almacenamiento al la vez [1]. Lo que se busca al aplicar persistencia políglota es que las aplicaciones primero definan las tareas que van a realizar, y en base a estas, que modelo de datos se adapta mejor a las necesidades de dicha tarea. De esta forma se tendrán varios sistemas de bases de datos para cada aplicación.

El principal problema a la hora de desarrollar este tipo de aplicación es manejar las distintas representaciones de los datos, dependiendo del sistema de almacenamiento, mantener las conexiones a cada base de datos y mantener la integridad

entre la data. Por lo que en principio ésto podría hacer que la aplicación fuera muy compleja. En la siguiente sección se describirán varios diseños de arquitectura de persistencia políglota y como manejan esta complejidad.

IV. Arquitecturas de Persistencia Políglota

Una arquitectura de persistencia políglota tiene como objetivo mostrar como se interrelacionan los diferentes elementos de persistencia políglota en un sistema o aplicación

Uno de los puntos más interesantes en las Aplicaciones de Persistencia Políglota - además de los sistemas de almacenamiento en si - es la forma en la que se debería estructurar la aplicación. La presencia de varios sistemas de almacenamiento trae consigo una serie de problemas y necesidades que no existen en las aplicaciones con un único sistema de almacenamiento.

En una aplicación simple se tendría un único sistema al cual se accedería para realizar las diferentes operaciones, independientemente de qué tarea se trate o qué datos se quieran consultar. Por tanto, la aplicación realiza operaciones que necesitan hacer diferentes consultas, pero todas ellas se hacen sobre el mismo sistema de almacenamiento como se puede observar en la figura 1

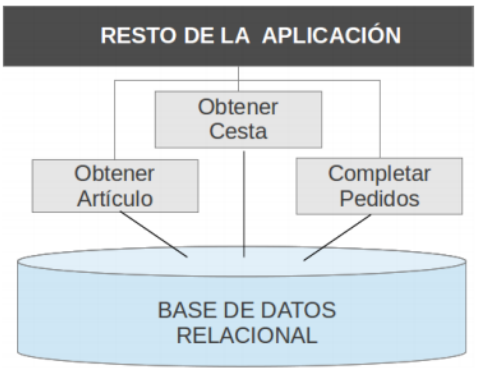

Figura 1. Ejemplo de una arquitectura con un único sistema de almacenamiento.

Al introducir otros sistemas de almacenamiento, como se muestra en la figura 2, la aplicación deberá realizar las consultas de los datos sobre distintos sistemas, con lo que puede ocurrir lo siguiente:

- No sería transparente para la aplicación el hecho de que hay más de un sistema manejador actuando por debajo, por tener diferentes modelos de datos.
- Se deben tomar en cuenta las implicaciones que tendría agregar nuevos manejadores o realizar algún cambio en uno existente.

Un primer diseño sería el más directo. En este caso, cada parte de la aplicación se comunica directamente con el sistema de almacenamiento que contiene los datos a los que se quiere acceder. Tal y como se puede ver en la figura 2 la aplicación realiza operaciones sobre cada uno de los sistemas de almacenamiento según lo vea necesario.

En principio este diseño genera varios problemas, pero uno de los principales es que la presencia de las tres bases de datos no es transparente para la aplicación, por ende, es indispensable tener conocimientos sólidos sobre como gestionar los datos en todos los SMBD. La problematica se presenta ya que generalmente para hacer persistencia políglota se utilizan diferentes SMBD, como en este caso, para gestionar los datos de una manera específica para cada tarea.

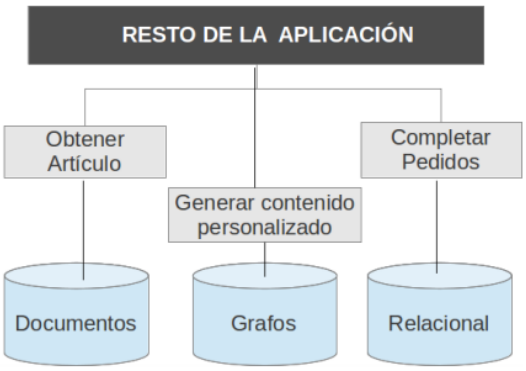

Figura 2. Arquitectura para una Aplicación de Persistencia Políglota: Diseño I.

El segundo diseño intenta eliminar los problemas anteriores. La idea es envolver las bases de datos en servicios, de forma que la aplicación no se comunique directamente con los sistemas de almacenamiento [2]. Ésta se comunicará con los servicios. Estos servicios serán los encargados de realizar las consultas sobre las bases de datos.

Con esto, por un lado se consigue que:

- La distribución de los datos entre diferentes sistemas de almacenamientos sea transparente para la aplicación.
- Los servicios puedan ser reutilizados por otras aplicaciones que necesiten acceder a esos datos.

Por tanto, una arquitectura con este diseño tendría forma similar al de la figura 3.

Siguiendo la idea anterior del uso de servicios se puede considerar que hay dos posibles opciones de diseño. Por un lado, tener un servicio por cada base de datos (Figura 3) y por el otro tener servicios en función de qué datos se guardan (Figura 4).

Cualquiera de los diseños anteriores es perfectamente válido, todo depende de si se tiene pensado reutilizar las consultas o se van a añadir o modificar los sistemas de bases de datos en el futuro.

V. Casos de Exito: Wanderu y Zephyr Health

V-A. *Wanderu*

Wanderu es una aplicación web cuya finalidad es que los consumidores puedan realizar búsquedas de tickets de tren y/o autobús, con la combinación de múltiples compañías de transporte, las rutas son almacenadas en formato JSON haciendo que la solución por excelencia trabaje con MongoDB. Sin embargo, también tienen la opción de buscar rutas óptimas

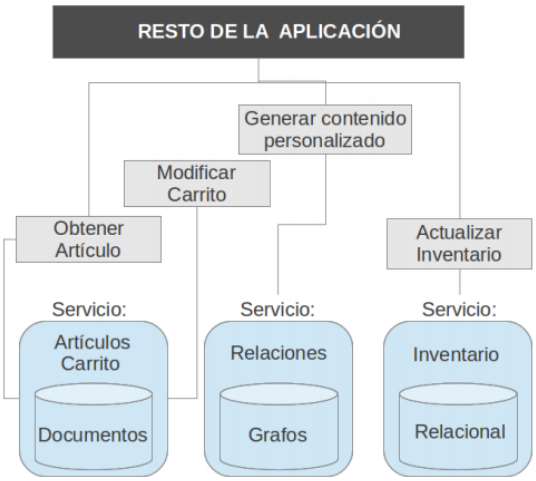

Figura 3. Arquitectura para una Aplicación de Persistencia Políglota: Diseño II.

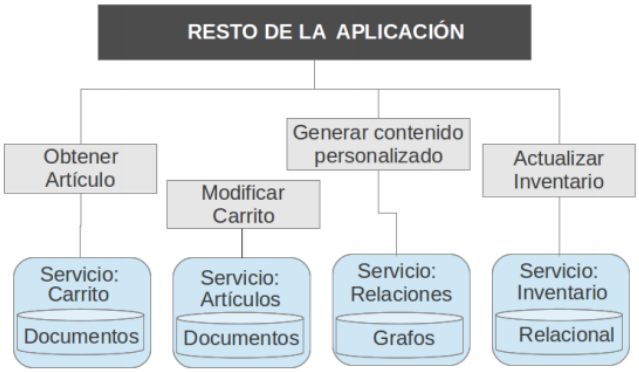

Figura 4. Arquitectura para una Aplicación de Persistencia Políglota: Diseño III.

desde un origen a un destino, tarea que es perfectamente implementable con una base de datos orientada a grafos como es el caso de Neo4J.[6]

Wanderu no quiso incrementar la complejidad haciendo el manejo de relaciones directamente desde MongoDB, lo que implicaría mucho procesamiento y terminaria siendo ineficiente, por lo que decidieron utilizar persistencia políglota para aprovechar las bondades de ambos SMBD (MongoDB y Neo4J).

V-B. Zephyr Health

Zephyr Health busca integrar diversos datos de cuidado de la salud mediante el uso de MongoDB y Neo4J, su plataforma realiza complejas tareas como lo son la conexión de los pacientes a los tratamientos de salud y terapias que los mismo necesitan.[7]

Fundada en 2011, Zephyr Health toma data de las compañías farmacéuticas y las transforma en información. Permitiendo realizar análisis en tiempo real que ayuda a los pacientes a conectar con diferentes terapias y tratamientos,

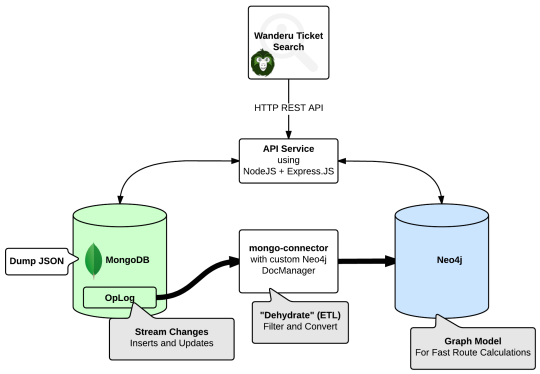

Figura 5. La búsqueda de Wanderu utiliza MongoDB para un manejo sencillo de JSON y Neo4J para un eficiente cálculo de rutas

y a las compañías farmacéuticas a conectarse con diferentes proveedores de cuidados de la salud.

MongoDB es su almacén de documentos, que contiene toda la informacién del perfil holístico para cada médico, información de tratamientos , etc. Sin embargo el manejo de las relaciones puede ser bastante complejo al usar MongoDB, por lo que Zephyr Health decide utilizar Neo4J para solventar ciertas limitaciones de MongoDB, Neo4J es un SMBD de índice pesado, el rendimiento es muy coherente si se trabaja con unos pocos miles de nodos o unos pocos millones.

VI. Tecnologías Usadas

VI-A. MongoDB

MongoDB es una base de datos documental y ágil de código abierto que permite a los esquemas cambiar rápidamente cuando las aplicaciones evolucionan, proporcionando siempre la funcionalidad que los desarrolladores esperan de las bases de datos tradicionales, tales como índices secundarios, un lenguaje completo de búsquedas, consistencia estricta y un alto rendimiento.[8]

VI-B. Neo4j

Neo4j es una base de datos gráfica nativa altamente escalable, diseñada específicamente para aprovechar no sólo los datos, sino también sus relaciones.

El motor de procesamiento y procesamiento de grafos nativo de Neo4j proporciona un rendimiento constante y en tiempo real, ayudando a las empresas a crear aplicaciones inteligentes para satisfacer los actuales desafíos de datos en constante evolución.[9]

VI-C. Neo4j Doc Manager

Es una herramienta que permite migrar documentos desde MongoDB a una estructura de grafos de propiedades de Neo4j. Simplemente se ejecuta en segundo plano y la información que se encuentra en MongoDB se importa a un grafo visible desde Neo4j, para permitir a los desarrolladores de MongoDB almacenar datos JSON en Mongo mientras consulta las relaciones entre los datos usando Neo4j.

MongoDB almacena datos como documentos similares a JSON, mientras que Neo4j almacena datos como grafos de propiedades. Para permitir la consulta basada en grafos de datos MongoDB, necesitamos determinar cómo mapear entre estas dos estructuras de datos diferentes. Neo4j Doc Manager provee un plan de mapeo por defecto. Siguiendo la convención en lugar de requerir configuración.[10]

VII. Caso de Estudio: MongoDB y Neo4j

Al crear aplicaciones lo desarrolladores tienen una gran variedad de tecnologías para escoger, incluyendo a lo que la elección de base de datos concierne, sin embargo, al añadir nuevas tecnologás a menudo termina significando mayor complejidad en el manejo de los sistemas en lugar de obtener un beneficio significante. La idea de la persistencia políglota es permitir tomar ventaja de los puntos fuertes de las distintas capas de persistencia para mejorar la funcionalidad de las aplicaciones. A continuación se presenta un caso de estudio tomado de la página oficial de Neo4j [11] basado en una aplicación de comercio electrónico en el que se utiliza MongoDB y Neo4j juntos en base a los puntos fuertes de cada base de datos y también se utiliza Neo4j Doc Manager para Mongo Connector, que permite la sincronización en tiempo real de MongoDB a Neo4j.

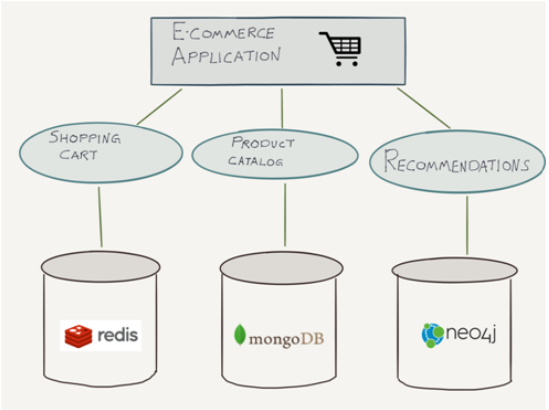

Figura 6. Se utiliza una BD claves-valor para alimentar el carrito de compras del usuario, una BD orientada a documentos para la búsqueda y la navegación por el catálogo de productos y una BD orientada a grafos para obtener recomendaciones personalizadas en tiempo real.

Un caso de uso básico en aplicaciones de comercio electrónico para una base de datos documental es la de manejar un catálogo de productos y poder realizar búsquedas sobre él. Entre los requisitos funcionales está el soportar una diversa cartera de productos con consultas complejas y filtrado a través de los atributos de muchos productos. Para entender cómo el uso de una base de datos de grafos junto con una base de datos de documentos puede mejorar una aplicación, se tomará como ejemplo una aplicación web que ofrece un catálogo de cursos en línea.

La vista mostrada (Figura 7) permite al usuario ver una lista de todos los cursos disponibles para ellos, buscar por palabra clave y filtrar por fecha, idioma o categoría. La vista

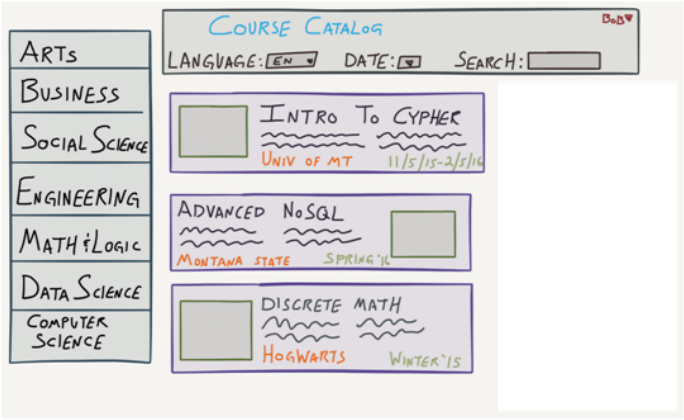

Figura 7. Un catálogo de cursos en línea es un buen caso de uso de una base de datos de documentos.

tiene que ser alimentada de una sóla consulta en BD por lo que se está volcando una gran cantidad de información ahí. Las consultas tienen que utilizar diferentes tipos de índices (de texto completo, filtro por categorías, filtros por intervalo de tiempo) y devolver toda la información necesaria para renderizar la vista, sin embargo, falta mostrar información que esté acorde al contexto del usuario. La aplicación debe saber que cursos el usuario ha tomado anteriormente y cómo el usuario ha interactuado con otros usuarios en la plataforma para ser capaces de ofrecer algún contenido personalizado basado en estas preferencias del usuario, por lo que entra en juego lo que sería los cursos recomendados al usuario en cuestión.

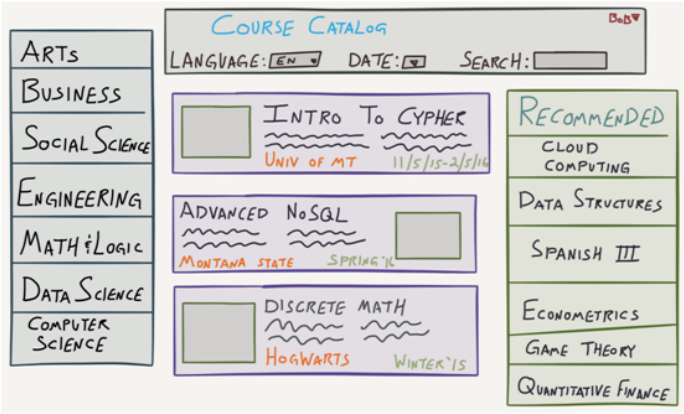

Figura 8. Catálogo de cursos con contenido personalizado, tales como las recomendaciones de cursos en base a la información que se sabe acerca de las preferencias del usuario actual.

Por sus características Neo4j es muy bueno para la generación de recomendaciones en tiempo real. MongoDB podría ser la más idónea en servir a un catálogo de productos, pero si lo que se quiere es generar contenido centrado en el usuario como por ejemplo las recomendaciones personalizadas de productos se sabe que es fácil en Neo4j. Ahora le pregunta es ¿Cómo?, normalmente, esto implicaría la sincronización de datos en

la capa de aplicación, es decir, interactuar directamente con MongoDB, e interactuar directamente con Neo4j pero surgen otras interrogantes como: ¿Ambas operaciones terminaron con éxito? ¿Qué hacer si una de las transacciones falla? ¿En qué punto se sincronizan los datos? Esto puede convertirse rápidamente en un componente muy complicado. Los beneficios de la persistencia políglota se producen a expensas de la complejidad. Ahora tenemos que aplicar la lógica a nivel de aplicación para escribir datos tanto MongoDB y Neo4j.

Neo4j Doc Manager: Habilitación de Persistencia Políglota en MongoDB y Neo4j

El proyecto Neo4j Doc Manager de los creadores de Neo4j es una implementación del proyecto Mongo Connector, proporcionado por la gente de MongoDB. Mongo Connector proporciona un mecanismo para notificar las solicitudes y actualizaciones en MongoDB y escribir los cambios a un sistema de destino. Neo4j Doc Manager funciona mediante el uso del Oplog (un registro de todas las operaciones en MongoDB). Siempre que hay una actualización de un documento (por ejemplo, una inserción, actualización o eliminación), Neo4j Doc Manager es notificado de la actualización y éste contiene la lógica para convertir ese documento en un modelo de grafos con sus respectivas propiedades y, a continuación, escribe inmediatamente esa actualización para Neo4j.

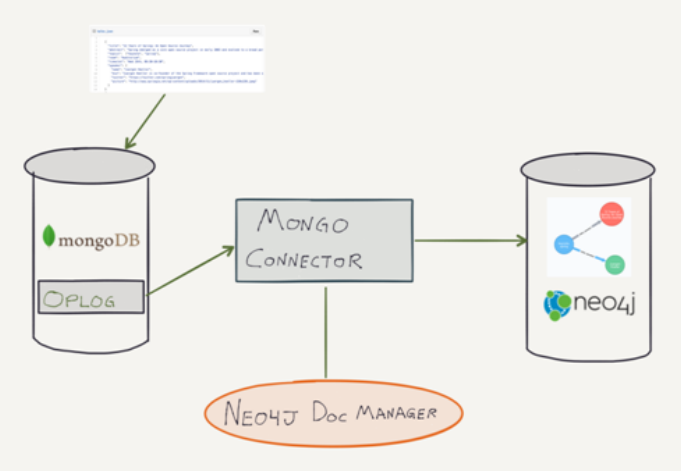

Figura 9. Neo4j Doc Manager es notificado de todas las operaciones en MongoDB, convierte esos cambios a un modelo de grafos y de inmediato escribe esos cambios a Neo4j.

El Neo4j Doc Manager permite al desarrollador de aplicaciones alcanzar el objetivo de construir una aplicación que aprovecha la persistencia políglota con MongoDB y Neo4j. En el contexto del catálogo de cursos, perfectamente se puede proporcionar búsqueda y navegación entre los productos con MongoDB, y proporcionar recomendaciones en tiempo real en base a lo que los cursos que el usuario ha tomado y cómo han interactuado con el sistema desde Neo4j.

Neo4j Doc Manager: Instalación e Inicio del servicio

Ya el equipo en dónde se ejecutará el servicio debe tener instalado y configurado MongoDB y Neo4j.

Se instala Neo4j Doc Manager clonando el repositorio y estableciendo el PYTHONPATH al directorio local correspondiente:

```
git clone https://github.com/neo4j-contrib/
        neo4j_doc_manager.git
cd neo4j_doc_manager
export PYTHONPATH=.
```

Si se tiene la autenticación habilidada para Neo4j, se debe asegurar que la variable de ambiente correspondiente contenga el nombre de usuario y la contraseña para conectarse al servidor:

```
export NEO4J_AUTH=<user>:<password>
```

Si la autenticación está deshabilitada en Neo4j esa acción no es requerida. Para deshabilitar la autenticación, se tiene que ir hasta el directorio de instalación de Neo4j y editar el archivo 'conf/neo4j-server.properties'de la siguiente manera:

```
dbms.security.auth_enabled=false
```

Ahora, de forma adicional se debe asegurar que mongo esté corriendo en un conjunto de réplicas. Para iniciar y establecer una réplica en Mongo basta con:

```
mongod --replSet myDevReplSet
```

Luego, ya con el servicio iniciado, abrir una cónsola de Mongo y correr:

```
rs.initiate()
```

Por último, una vez realizado los pasos anteriores se inicia el servicio de Neo4j Doc Manager con el siguiente comando en una sóla línea:

```
mongo-connector -m localhost:27017 -t
http://localhost:7474/db/data -d
neo4j_doc_manager
```

-m específica el servidor de MongoDb, -t el servidor de Neo4j (incluyendo el protocolo) y -d específica el Neo4j Doc Manager.

Neo4j Doc Manager: Primeros pasos

Los documentos se convierten en grafos en base a la estructura del documento. Las llaves del documento se convertirán en nodos. Los valores anidados entre cada llave se convertirán en propiedades.

A continuación (Figura 10) se muestra un documento que se inserta en MongoDB para ejemplificar como trabaja el Neo4j Doc Manager.

El Documento insertado en MongoDB es transformado a un modelo de grafos en Neo4j (Figura 11 y Figura 12).

Nodos creados:

- producto: producto es el nodo raíz, viene del nombre de la colección a la que pertenece el documento con un "_id" que también viene de MongoDB. Documentos

```
db.producto.insert([
{
  _id:"pro0001",
  nombre:"Playstation 4",
  descripcion:"Playstation 4 500 GB",
  caracteristicas:["Playstation 4","500 GB","2 Controles"],
  sku:"a123c001",
  categoria_id:"cat0003",
  envio:{
    peso:5.5,
    ancho: 70,
    alto: 55,
    profundidad: 20,
  },
  precio:{
    total:450.00,
    sin_iva:396.00,
    iva:54.00,
    pct_iva:12
  }
},
{
```

Figura 10. Documento insertado en MongoDB que se convierte a grafos en Neo4j en base a su estrcutura

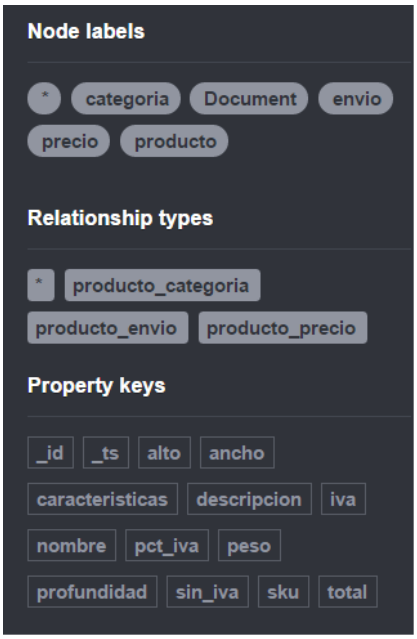

Figura 11. Información de la base de datos en Neo4j

no anidados son convertidos a propiedades del nodo, como por ejemplo: ”nombre”,”descripcion”, ”caracteristicas”,”sku’.

- envio: envio es un documnto embebido. Pares clave-valor que estén dentro son convertidos a propiedades del nodo como por ejemplo: ”peso”, ”ancho”, ”alto”, ”profundidad”. También tiene una propiedad ”_id” que viene del nodo raíz y es la mismo.
- precio: precio es un documento embebido por lo que es análogo al nodo session.
- categoria: en la colección producto hay una clave ”categoria_id” que Neo4j Doc Manager toma como una referencia a un nodo con etiqueta categoria que contenga ese id y crea la relación entre ellos. En caso de no existir, crea el nodo categoria y le asigna el id que tiene como valor en el documento.

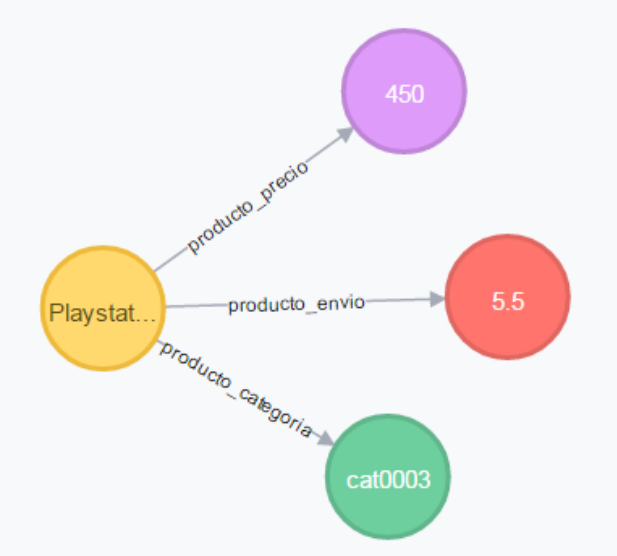

Figura 12. Grafo en Neo4j que se generó según el documento insertado en MongoDB

También es importante mencionar que a los nodos se les agrega automáticamente una propieda ”_ts” que representa su timestamp de la creación en MongoDB.

Relaciones creadas:

- producto_precio: Una relación que conecta a los nodos producto y precio.
- producto_envio: Una relación que conecta a los nodos producto y envio.
- producto_categoria: Una relación que conecta a los nodos producto y categoria.

Ya visto como se hace la conversión de un documento en MongoDB a grafos en Neo4j podría decirse que las posibilidades son muchas, ya que desde MongoDB se puede trabajar directamente con las colecciones y los datos se verán reflejados en Neo4j para después procesar consultas relacionas a estructuras de grafos. Desde Mongo se puede hacer actualizaciones y eliminaciones según criterios de filtrado que se reflejarán en Neo4j y que permitirán hacer cosas como por ejemplo: agregar o eliminar propiedades en los nodos, cambiar la estructura de los mismos, agregar o eliminar relaciones, entre otras cosas. Para ello, se invita al lector a profundizar más en el tutorial de Neo4j Doc Manager que se encuentra en la página oficial de Neo4j y en donde se explica de manera detallada los comandos a ingresar en MongoDb y los resultados obtenidos. Tutorial disponible en el siguiente enlace:
https://neo4j.com/developer/neo4j-doc-manager/

Neo4j Doc Manager: Aplicación de Persistencia Políglota

Siguiendo el esquema propuesto (Figura 6) para una aplicación de persistencia políglota los autores del presente documento desarrollarón una sencilla aplicación de comercio electrónico (Figura 13) que maneja datos provenientes de diferentes sistemas manejadores de base de datos para su funcionamiento.

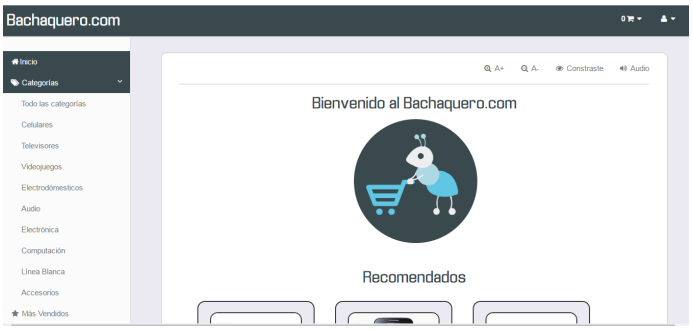

Figura 13. Vista general de la aplicación de comercio electrónico

Las secciones de categorías y de catálogo de productos (Figura 14) se alimentan de datos almacenados en MongoDB. Se muestra contenido adecuado al contexto del usuario mediante recomendaciones (Figura 15) que se hacen a través de consultas hechas en Neo4j y se cuenta con un carrito de compras (Figura 16) que se puede implementar mediante una base de datos clave-valor.

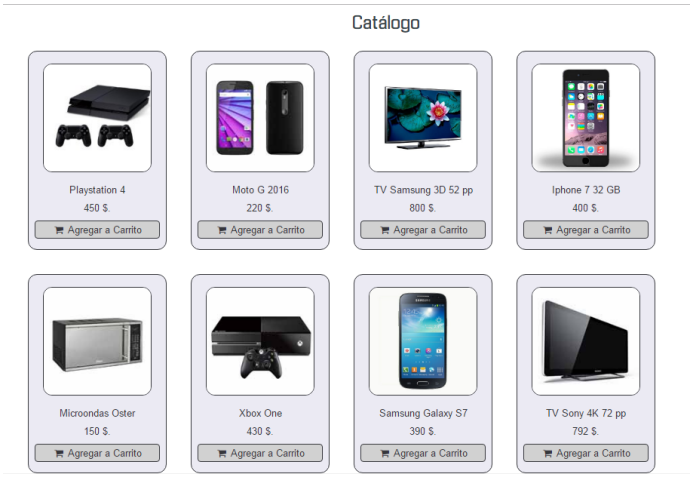

Figura 14. Catálogo de productos de la aplicación de comercio electrónico con datos obtenidos de MongoDB

VIII. Conclusiones

Cuando se trata de construir aplicaciones escalables, los desarrolladores poseen un millar de tecnologias de donde escoger, especialmente cuando se trata de la elección de SMBD. Lo que se busca es escoger las tecnologías adecuadas que ofrezcan las funcionalidades necesarias bajo un desempeño ideal.

La idea de la persistencia políglota permite tomar ventajas sobre las diferentes cosas que nos ofrecen los SMBD al mismo tiempo para tener aplicaciones escalables.

Figura 15. Productos recomendados al usuario en la aplicación de comercio electrónico con datos obtenidos de Neo4j

Figura 16. Carrito de compras de la aplicación de comercio electrónico

Teniendo todo esto en cuenta la pregunta que hay que hacerse es ¿merece la pena hacer una Aplicación de Persistencia Políglota? La respuesta es que depende. En una aplicación simple y con poca carga de datos, la realidad es que un único sistema es más que suficiente. Sin embargo, en una aplicación con muchos datos de diferente tipo y con diferentes necesidades se puede sacar provecho del uso de varios sistemas frente a uno solo donde el rendimiento baje o la complejidad del esquema de datos desborde al desarrollador y a los administradores.

Referencias

[1] Gutiérrez E. Aplicación de Persistencia Políglota para Almacenamientos NoSQL.Universidad del Pais Vasco, 2016.

[2] Pramod J. Sadalage Martin Fowler. NoSQL Distilled: A brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley, 2013.

[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI=http://dx.doi.org/10.1145/1365815.1365816

[4] Vaish, Gaurav. Getting Started with NoSQL. Packt Publishing, 2013.

[5] ECMA-404. Introducción a JSON. 2013. [online] Disponible en: http://www.json.org/json-es.html

[6] Boyd R. Polyglot Persistence Case Study: Wanderu + Neo4j + MongoDB Neo4j Blog, 2015.

[7] Chaudhari M. Integrating Diverse Healthcare Data using MongoDB and Neo4j.Neo4j Blog, 2016.

[8] MongoDB, Inc. Reinventando la gestión de datos. [online] Disponible en: https://www.mongodb.com/es

[9] Neo Technology, Inc. Neo4j: The World's Leading Graph Database. [online] Disponible en: https://neo4j.com/product/
[10] Neo Technology, Inc. Neo4j: The World's Leading Graph Database. [online] Disponible en: https://neo4j.com/developer/neo4j-doc-manager/
[11] Lyion W. Neo4j Doc Manager: Polyglot Persistence for MongoDB and Neo4j. Neo4j Blog, 2015.