Comparación Base de Datos Documental - Grafo

Sebastián Zarrillo *Facultad de Ingeniería, Universidad de la República*
Montevideo, Uruguay
sebastian.zarrillo@fing.edu.uy
Jairo Correa *Facultad de Ingeniería, Universidad de la República*
Montevideo, Uruguay
jairo.correa@fing.edu.uy

Resumen

Este documento es el informe correspondiente a la entrega del proyecto final de la edición 2022 del curso de Bases de datos no relacionales de la Facultad de Ingeniería de la Universidad de la República.

I. Introducción

El objetivo de nuestro trabajo es realizar una comparación justa entra una base de datos documental y una base de datos de grafo para una misma realidad. La comparación será realizada en base al diseño, implementación y rendimiento de cada solución. Un punto importante a destacar para que la comparación sea justa, es la realidad planteada a partir de la cual se realizará.

Nuestra idea parte del área de los videojuegos, más específicamente de la plataforma Steam. Esta permite, entre otras cosas realizar compras de videojuegos para computadora y además contiene un sistema de recomendación de videojuegos en base a las compras realizadas por el usuario y las ventas realizadas mundialmente. A partir de esto, decidimos plantear una realidad similar, donde existan usuarios, que puedan explorar y comprar juegos. También que puedan recibir recomendaciones de otros videojuegos en base a las compras ya realizadas por ellos y otros usuarios.

Decidimos implementar una API REST que exponga los juegos del sistema y permita a los usuarios realizar compras, explorar juegos y recibir recomendaciones de juegos que pueden ser de su interés. Esta API va a tener dos sub-dominios que van a ser prácticamente iguales, con la salvedad de que uno de ellos va a consumir la base de datos documental y el otro la base de datos de grafo.

Luego de la implementación de la API, pasaremos a realizar tests de rendimiento para nuestra solución, con el fin de ver si existen grandes diferencias de rendimiento para la creación y consumo de datos de ambos tipos de base de datos. En base a la realidad planteada, creemos que la base de datos documental va a ser una mejor solución en cuanto a dificultad de implementación y *performance*, pero la base de datos de grafo tendrá mejor rendimiento para el módulo de recomendaciones.

En la siguiente sección (sección II) vamos a presentar los conceptos claves que creemos necesarios para comprender este artículo. Luego, en la sección III vamos a explicar a fondo el trabajo realizado. En la sección IV vamos a mostrar las pruebas realizadas y los resultados obtenidos, y por último en la sección V vamos a plantear conclusiones y trabajos a futuro.

II. Marco Teórico

II-A. MongoDB

MongoDB [1] es un sistema de base de datos documental de código abierto.

II-B. Neo4j

Neo4j [2] es un software libre de base de datos orientada a grafos, implementado en Java. Es un sistema de alto desempeño para poder almacenar datos en forma de grafos y consumirlos.

II-C. API Rest

Una API [3] o *Aplication Programing Interface* es un concepto que hace referencia a los procesos, las funciones y los métodos que brinda determinado servicio para que este pueda ser utilizado por otro. El concepto de REST [4] o *Representational State Transfer* hace referencia a un protocolo de comunicación que está basado sobre el protocolo HTTP. Este sirve para obtener y generar datos, retornando los mismo en formatos como XML o JSON. Una API REST es una hace referencia a una interfaz de programación que se comunica mediante el protocolo REST.

[1] https://www.mongodb.com/
[2] https://neo4j.com/
[3] https://es.wikipedia.org/wiki/Interfaz_de_programaci\%C3\%B3n_de_aplicaciones
[4] https://openwebinars.net/blog/que-es-rest-conoce-su-potencia/

II-D. NodeJs

NodeJs es un ambiente de ejecución multiplataforma de código abierto basado en el lenguaje de programación Javascript. Este mismo está enfocado principalmente en la capa de servidor.

II-E. Heroku

Heroku [5] es una solución de computación en la nube que permite a sus clientes hostear aplcaciones en la nube de forma sencilla.

II-F. Algoritmo de recomendación

La recomendación de videojuegos que pueden ser de interés para los usuarios en base a las compras realizadas es uno de los problemas a resolver en nuestro sistema. Por esa razón vamos a especificar en esta sección como vamos a realizarla. Cabe destacar que decidimos hacerlo lo más simplificado posible, ya que realizar un buen algoritmo de recomendación no es el foco principal de este trabajo.

La forma más simple de explicar nuestro algoritmo es con un pseudocódigo:

Listado 1 Recomendación

```
var juegosRecomendados = [] // Diccionario de clave (id juego) valor (cant veces comprado)

# para cada compra del usuario
for juego in comprasUsuario {
   # busco todos los usuarios que compraron el mismo juego
   var usuariosQueCompraron = buscoUsuariosQueCompraron(juego)
  for usuario in usuariosQueCompraron {
     # marco todas las otras compras de ese usuario como posible juego recomendado
     for juegoComprado in (usuario.compras - juego) {
        juegosRecomendados[juegoComprado] += 1
     }
  }
}

# ordeno el mapa por cantidad de veces comprado y me quedo con los top 10
var juegosOrdenados = ordenarJuegosPorCantidad(juegosRecomendados)
return juegosRecomendados.top(10)
```

Lo que se hace es contar la cantidad de veces que cada juego se compró en conjunto con alguno de los juegos comprados por el usuario y se recomendarán los 10 juegos que se compraron más veces en conjunto con alguno de los juegos comprados por el usuario. Que dos juegos se compraron en conjunto se refiere a que un usuario compró ambos juegos.

Como se menciona anteriormente, el foco del trabajo no pasa por implementar el mejor algoritmo de recomendaciones posible. Claramente resultaría interesante tener en cuenta distintos aspectos de los videojuegos como el año en el que se realizó, el genero del mismo o la valoración que el usuario hizo a la compra que nos lleva a recomendar determinado juego. Pero estas modificaciones no tendrían una impacto significativo en la forma en la que interactuamos con ambas bases de datos.

III. Parte central

En esta sección se presentará el sistema implementado. Para ello vamos a presentar las herramientas utilizadas, las fuentes de datos, la carga de datos y discutiremos algunos detalles sobre la implementación.

III-A. Herramientas utilizadas

En primer lugar, la implementación de nuestro proyecto fue realizada en *NodeJs*[6] utilizando *ExpressJs*[7] como framework para nuestra API REST. Para realizar las pruebas manuales de nuestra API se utilizó como herramienta *Postman*[8], y para realizar el hosting de nuestra aplicación para poder realizar nuestras pruebas en un servidor utilizamos *Heroku*[9]. Por último, como herramienta de control de versionado utilizamos el servicio de *GitLab* de Facultad de Ingeniería[10].

Como sistema de base de datos documental se utilizó *MongoDB*[11] y *MongoDB Cloud*[12] para el almacenamiento de los datos en la nube. Para consumir y modificar nuestra base de datos *MongoDB* utilizamos la librería *Mongoose*[13].

[5] https://www.heroku.com/
[6] https://nodejs.org/es/
[7] https://expressjs.com/es/
[8] https://www.postman.com/
[9] https://www.heroku.com/
[10] https://gitlab.fing.edu.uy/
[11] https://www.mongodb.com
[12] https://cloud.mongodb.com/
[13] https://mongoosejs.com/

Por otro lado, para la base de datos de grafos se utilizó *Neo4j*[14] y *Neo4j Aura*[15] para el hosteo en cloud de la misma. Para conectarnos y realizar consultas desde nuestro proyecto utilizamos el driver que proporciona Neo4j para Javascript [16].

III-B. Requerimientos

Como se mencionó anteriormente, el principal objetivo de nuestro sistema el la venta de videojuegos para computadora. Para ello necesitamos Usuarios y Juegos. A continuación se listan las funcionalidades requeridas para los usuarios:

- Como potencial usuario debo poder registrarme en el sistema ingresando un nombre de usuario.
- Como usuario del sistema (sabiendo mi identificador de usuario) debo poder editar mi nombre de usuario.
- Como usuario del sistema (sabiendo mi identificador de usuario) debo poder borrar mi usuario.
- Como usuario del sistema (sabiendo mi identificador de usuario) debo poder comprar un videojuego que aún no haya comprado.
- Como usuario del sistema (sabiendo mi identificador de usuario) debo poder visualizar mis compras realizadas.
- Como usuario del sistema (sabiendo mi identificador de usuario) debo poder explorar los juegos recomendados en base a mis compras.
- Todo usuario del sistema debe poder explorar los juegos en el sistema, pudiendo además realizar búsquedas por texto.
- Todo usuario del sistema debe obtener los detalles de un juego sabiendo su identificador.

Además, para poder realizar mejores pruebas sobre nuestro sistema, también se deben satisfacer los siguientes requisitos:

- Se debe poder crear un juego.
- Se debe poder editar un juego.
- Se debe poder eliminar un juego.
- Se deben poder listar registrados en el sistema.

III-C. Fuente de Datos

Para que nuestro sistema sea lo más similar posible a una plataforma de venta de videojuegos real debemos tener juegos, usuarios y compras realizadas por los usuarios. Luego de investigar, encontramos en el sitio *kaggle* dos fuentes de datos que decidimos utilizar en conjunto.

- En primer lugar, encontramos un catálogo [17], de juegos para varias plataformas. Este catálogo contiene el nombre, año de lanzamiento, género, desarrollador y publicador del juego, además de información de cantidad de ventas y plataforma del juego. A nosotros solamente nos interesan los juegos para computadora.
- Por otro lado, encontramos también un listado de ventas [18] de la plataforma Steam, el cual contiene nombre del juego comprado e identificador o número de usuario que realizó la compra. Vale destacar que, como Steam es un servicio disponible solamente para computadora, están ventas son solamente de juegos para computadora.

Combinando ambas fuentes de datos conseguiremos una rica base para nuestro sistema.

III-D. Diseño

Ya encontrada nuestra fuente de datos, podemos pasar al diseño de nuestras bases de datos. Vamos a mostrar el diseño de cada una por separado.

III-D1. Documental: En la Figura 1 se muestra el diseño de nuestra base de datos documental.

Nuestro diseño de la base de datos documental es relativamente simple. Contamos con solamente dos entidades grandes, una para los Juegos y otra para los Usuarios. Los usuarios cuentan con un listado de compras, el cual es el identificador del juego comprado. Por otro lado, los juegos cuentan con un listado de juegos relacionados, el cual, al igual que las compras de los usuarios, es un listado de identificadores de juegos.

Como se puede observar, el diseño de la base de datos documental tiene parcialmente resuelto el problema de los juegos relacionados para luego realizar las recomendaciones a los usuarios. Esta va a ser una de las grandes diferencias entre la base de datos documental y de grafo. Decidimos resolver este problema de esta manera, ya que realizar el calculo de los juegos recomendados para un usuario en tiempo de ejecución iba a ser muy costoso. En la sección III-E vamos a explicar más a fondo como resolvimos este problema y que posibles complicaciones puede traer.

[14] https://neo4j.com
[15] https://neo4j.com/cloud/platform/aura-graph-database/
[16] https://neo4j.com/developer/javascript/
[17] https://www.kaggle.com/datasets/gregorut/videogamesales
[18] https://www.kaggle.com/datasets/tamber/steam-video-games

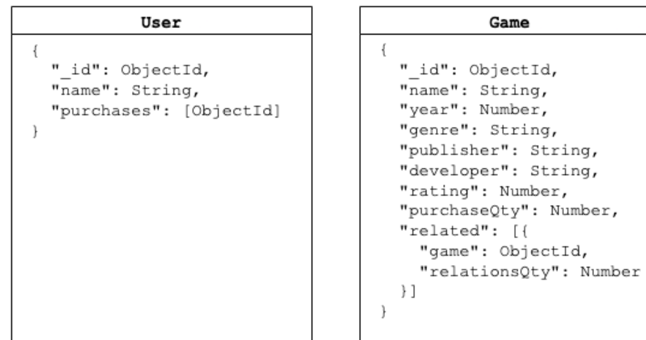

Figura 1: Diseño base de datos documental.

III-D2. Grafo: se tienen tres tipos de nodos, cada uno con una sola etiqueta.

- User: los nodos que tienen esta etiqueta representan un usuario de nuestro sistema. Cuentan con solo dos propiedades además del id auto generado por Neo4j: un identificador de Steam que llamamos *Steam_Id* y el nombre de usuario (*Name*) proporcionado al registrarse. En un contexto mas cercano a la realidad, la cantidad de propiedades crecería notoriamente.
- Game: los nodos etiquetados con *Game* representan a los juegos de nuestro sistema y cuentan con las siguientes propiedades:
  1. *id*: auto generado por Neo4j.
  2. *Name*: de tipo String.
  3. *Publisher*: de tipo String.
  4. *Year*: el año de publicación, de tipo Integer.
  5. *User_Count*: de tipo Float.
  6. *User_Score*: de tipo Integer.
- Category: utilizado para representar una categoría o genero al cual un juego puede pertenecer.

En cuanto a los tipos de relación podemos encontrar dos:

- *CATEGORIZED_AS*: relaciona un nodo *Game* con un nodo *Category*. Se asume que cada juego pertenece como máximo a un genero o categoría, por lo cual se controla que cada juego tenga a lo sumo una relación de este tipo. Se decide modelar el genero como un nodo a parte del juego y relacionarlo con el mismo en lugar de que sea una propiedad del propio nodo *Game*. Esto ultimo se decide así ya que se considera que el dominio de los géneros es mas reducido que el de los juegos, muchos juegos pertenecen a la misma categoría.
- *PURCHASED*: relaciona un nodo *User* con un nodo *Game* y representa una compra realizada por dicho usuario. Si bien esta relación no cuenta con propiedades, la fecha de compra o el monto de la misma surgen como candidatos en una realidad mas compleja.

La imagen 2 muestra parte de la base de datos que permite visualizar lo explicado anteriormente.

III-E. Carga de Datos

Partiendo de las dos fuentes de datos previamente presentadas, el proceso de carga de datos se basa en combinar de ambas fuentes de datos de la mejor forma posible. Ambas fuentes de datos son archivos */.csv* y la lectura y procesamiento de los mismos se realizó en *nodeJs*.

A gran escala la carga de datos se puede resumir en los siguientes pasos:

- Cargamos todos los juegos del catálogo de juegos filtrando solo aquellos que sean para la plataforma de computadora.
- Cargamos todos los juegos dentro de nuestro listado de ventas buscando que el mismo no exista aún. Para verificar la existencia debemos buscar por nombre. Luego de ver nuestro datos, encontramos que existen varios juegos con el mismo nombre que difieren en caracteres especiales, como por ejemplo en nuestro catálogo de juegos contamos con el juego *The Witcher 3: Wild Hunt*, mientras que en nuestro listado de ventas aparece como *The Witcher 3 Wild Hunt*. Para ello decidimos crear un id provisorio limpiando el nombre de todos los caracteres especiales y los espacios. Si el juego a ingresar desde una venta no existe, se agrega solamente con el nombre, ya que el resto de la información no está disponible.
- Cargamos todos los usuarios que realizaron compras en nuestro listado de compras con nombre igual al identificador del listado también cargando sus compras correspondientes.

A continuación se presentan salvedades que se realizaron en la carga de datos para cada una de las bases de datos.

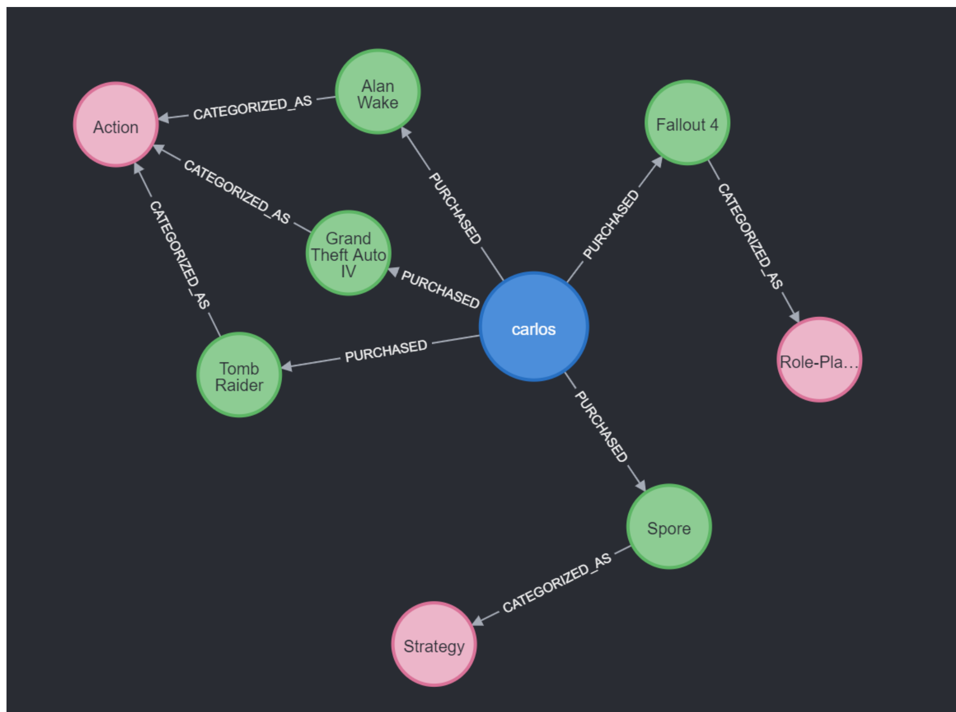

Figura 2: Base de datos de grafos

III-E1. Documental: Como se menciono en la sub-sección III-D1, el problema de los juegos recomendados se decidió resolver en la carga de datos. Para ello se implementó un *script* que precalcula en base a las compras de todos los usuarios los juegos relacionados para cada uno de los juegos en el sistema. Con ese procedimiento ya precalculado, para realizar la recomendación de juegos para un usuario hay que simplemente unir todos los juegos relacionados para la compras del usuario y retornar el *top* 10 de los mismos.

Lamentablemente, al ser calculados mediante un *script*, los juegos recomendados para un usuario no se actualizarán luego de que el usuario realice una compra. Esta es una gran desventaja de nuestra solución documental sobre la de grafo. En un sistema real, el *script* mencionado, o una alternativa tal vez un poco más performante del mismo podría ejecutarse una vez por día en la medianoche para mantener el listado de juegos recomendados para un usuario lo más actualizado posible. Otra alternativa podría ser tener un proceso de fondo que realice estás actualizaciones luego de que se ejecute una compra actualizando los juegos recomendados de forma asíncrona. De todas formas, estas u otras alternativas que puedan surgir quedan por fuera del alcance de este proyecto.

III-E2. Grafo: A diferencia de la base de datos documental, la recomendación de videojuegos es calculada en el momento que se ejecuta el método correspondiente, y no en la carga de datos. Mas detalles sobre esto se detallan en la sección III-F1.

La carga de datos se realiza a nivel de código, generando un arreglo de datos, ya sean juegos o compras, que es pasado como parámetro de una consulta. Y con el uso de la clausula `UNWIND` se crean los nodos y relaciones correspondientes para cada linea del dataset.

Un primer intento fue el uso de la clausula `LOAD CSV FROM` o `LOAD CSV WITH HEADERS FROM` sin éxito. Se necesita tener acceso al archivo desde la base de datos, además de que se encuentran distintos errores como por ejemplo: parsear caracteres especiales. Se decide implementar usando `UNWIND` y dejar el uso de `LOAD CSV FROM` como trabajo futuro.

III-F. Implementación

A continuación vamos a realizar una breve exposición de la implementación de nuestro sistema. Como ya se mencionó previamente, la implementación del sistema fue realizada en *NodeJs* utilizando *ExpressJs* como framework para la implementación de nuestra API REST. Para conectarnos con la base de datos *MongoDB* se utilizó la librería *mongoose* y para la de grafos se utilizo el propio driver para JavaScript que Neo4j provee.

En la Figura 3 se puede visualizar un pequeño diagrama exponiendo a gran escala la arquitectura de nuestro sistema. Allí se puede ver que contamos con dos sub-dominios dentro de nuestra API, uno que expone los datos de la base de datos documental (*/documental*) y otro que expone los datos de la base de datos de grafo (*/graph*). Ambos sub-dominios exponen los mismos endpoints REST. Estos son los siguientes:

- *POST /users* - Registrar usuario
- *GET /users* - Obtener todos los usuarios

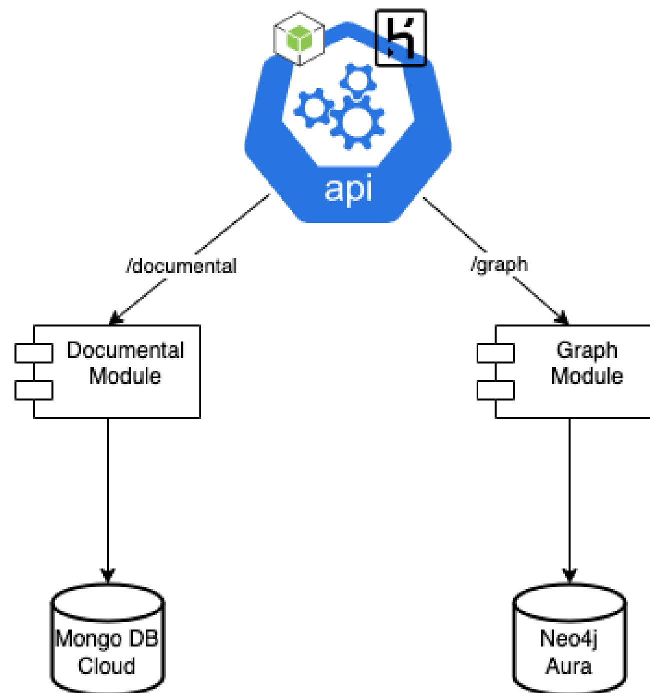

Figura 3: Arquitectura

- *GET /users/:id* - Obtener usuario por identificador
- *PATCH /users/:id* - Actualizar usuario
- *DELETE /users/:id* - Borrar usuario
- *POST /users/:id/purchases* - Comprar juego para un usuario
- *GET /users/:id/purchases* - Obtener compras de un usuario
- *GET /users/:id/recommended* - Obtener juegos recomendados para un usuario
- *POST /games* - Crear juego
- *GET /games* - Listar juegos
- *GET /games/:id* - Obtener juego por identificador
- *PATCH /games/:id* - Actualizar juego
- *DELETE /games/:id* - Borrar juego

No se va a entrar en detalle acerca de la implementación de cada uno de los *endpoints*.

III-F1. Recomendación de videojuegos en la base de datos de grafos: Para el caso de esta base de datos, se implementa el calculo de los juegos recomendados para un usuario en una consulta y no en un *script*. En términos de grafos, el pseudocódigo mostrado en II-F puede verse como:

1. Buscar todos los caminos de largo 3 que sigan el patrón que se puede apreciar en 4:
2. Para cada juego de los alcanzados por los caminos anteriores, contar de cuantas formas distintas llegue a ellos.
3. Recomendar los que son alcanzados por mas caminos distintos.

La operación de buscar caminos en un grafo, esta bien implementada y optimizada en las bases de datos de este estilo. Es por esto que se decide utilizar una consulta sobre la base para resolver este problema.

IV. Experimentación

En esta sección vamos a exponer los resultados de nuestra experimentación. En primer lugar vamos a exponer a modo de comparación los tiempos de ejecución de la carga de datos y de las pruebas de rendimiento implementadas. Luego vamos a exponer los resultados de nuestro sistema de recomendaciones, y por último vamos a realizar una breve comparación acerca de como respondería nuestro sistema ante posibles cambios de requerimientos.

Antes que nada, a continuación vamos a dejar el link al nuestro proyecto de GitLab: https://gitlab.fing.edu.uy/bndr10/proyecto-final. En el mismo se va a encontrar una carpeta *data* que contienen los archivos de datos utilizados en formato *csv*.

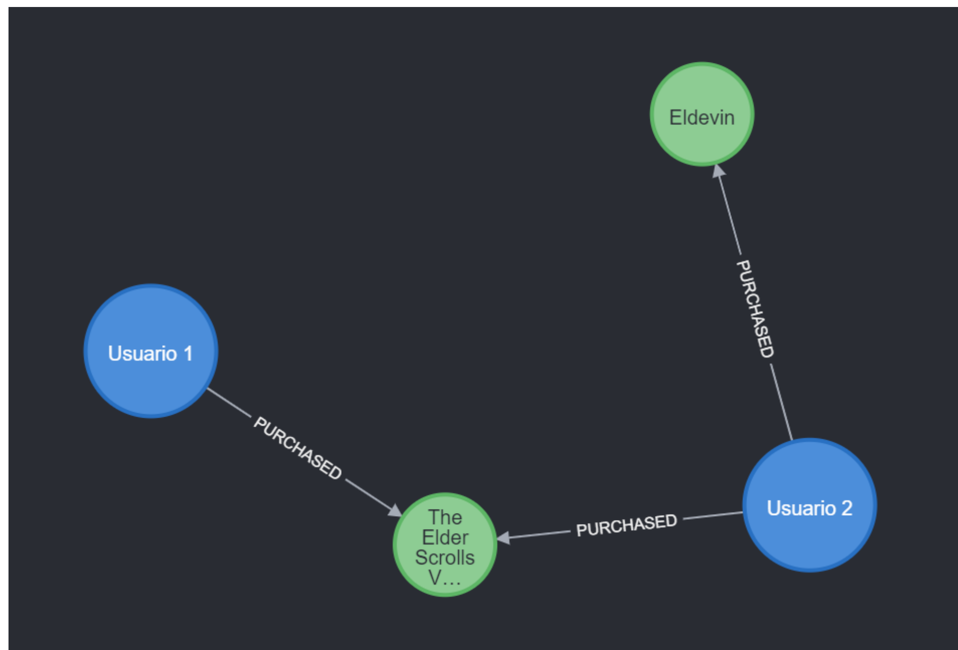

Figura 4: Ejemplo de camino para recomendación

Además existen dos archivos *graph/loadData.js* y *documental/loadData.js* que se encargan de realizar la carga de datos para la base de datos de grafo y documental respectivamente. También podrán encontrar una directorio *tests* con dos sub-directrios *graph* y *documental* donde se encuentran los archivos de test. Por último podrán encontrar un archivo *postman_collection.json* que contiene un colección de *Postman* para poder realizar pruebas manuales contra nuestra API, la cual está hosteada en https://bndr-final.herokuapp.com/.

IV-A. Carga de Datos

En la Tabla I se presentan los tiempos de carga de datos para ambas base de datos.

Como se puede ver en la Tabla I, hay una diferencia relativamente grande entre los tiempos de carga de datos para la base documental y la base de grafo.

IV-B. Pruebas de rendimiento

Decidimos implementar 3 pruebas que rendimiento que en nuestra opinión cubren la mayor parte de nuestro sistema. Estas son las siguientes:

- 1. Prueba de juegos: En esta prueba se testea un flujo de creación, edición, búsqueda y borrado de juegos.
- 2. Prueba de usuarios: En esta prueba se testea un flujo de registro de varios usuarios, realizando compras, obtener el perfil, listar compras, listar juegos, editar y borrar el usuario. Todo esto en paralelo.
- 3. Prueba genérica: En esta prueba se testea un flujo genérico de navegación en el sistema para muchos usuarios. Para ello se obtienen 100 usuarios y en paralelo se obtiene el usuario, sus compras y todos sus juegos recomendados.

En la Tabla II se presentan los tiempos que demoran nuestras pruebas de rendimientos para ambas bases de datos.

Como se puede ver en la Tabla II hay una leve diferencia de rendimiento favorable a la base de datos documental. En particular, como se puede ver en la prueba genérica, la base de datos comienza a tener cierta disminución de rendimiento cuando aumentan la cantidad de usuarios en paralelo. Pero hay que considerar también que al obtener los juegos recomendados dentro del test genérico la base de datos documental ya tiene esa operación precalculada, por lo que el tiempo de ejecución disminuye. Nuestro procedimiento de pre-calcular los juegos relacionados para nuestra solución documental demora unos *88.26* segundos en ejecutar.

Cuadro I: Tiempo carga de datos

Base de datos	**Tiempo**
Documental	*5.96 segundos*
Grafo	*106.24 segundos*

IV-C. Recomendaciones

En base a la experimentación en nuestra solución para el algoritmo de recomendaciones, notamos algunos puntos en el mismo que valen la pena mencionar. Tal como se explicó en la sección II-F, nuestro algoritmo de recomendación se basa en las compras realizadas por los usuarios del sistema de forma que, en base a los juegos que un usuario compró, se van a recomendar los juegos más comprados por otros usuarios que compraron los mismos juegos que él. Esto implica que los juegos más comprados van a ser más propensos a aparecer como recomendados por el simple hecho de que son mundialmente comprados, por más que no estén necesariamente relacionados con los juegos que el usuario compró. Esto no es necesariamente un punto negativo, ya que si un juego es muy comprado hay una gran posibilidad de que sea del interés del usuario en cuestión.

De todas formas, nuestro objetivo inicial es que las recomendaciones sean algo basado en las compras del usuario, pero si por lo general los juegos que se recomiendan son los más comprados se pierden un poco la recomendaciones personalizadas. Una posible solución a ello podría ser tomar en cuenta el género de los videojuegos comprados y el del videojuego a recomendar. Otra podría ser normalizar las relaciones del usuario con el videojuego a recomendar con la cantidad de ventas totales, así intentando recomendar también videojuegos que no son tan vendidos pero igual podrían ser de interés.

IV-D. Respuesta ante cambios de requisitos

Se plantean distintos cambios en los requisitos que puedan afectar el diseño de la base de datos. Para cada uno se plantea una alternativa que contemple dicho cambio.

- *Permitir que un usuario valore un juego y pueda ver la valoración general del mismo*: como usuario luego de haber comprado un juego debo poder dar una valoración con un valor entero entre 1 y 10 incluidos. Además como usuario debo poder ver la valoración general de cualquier juego independientemente de si lo compré o no.
- *Permitir que un usuario deje un comentario en el juego*: como usuario debo poder dejar un comentario para un juego. Debo poder repetir la acción las veces que quiera.
- *Permitir que un usuario acceda a los datos de facturación de sus compras*: como usuario debo poder ver una factura de cada compra realizada. Además debo poder acceder a un resumen de los gastos realizados.
- *Permitir que un juego exista para una o mas plataformas*: cada juego debe de estar disponible para al menos una plataforma. En caso de estar disponible para mas de una plataforma, debo permitir que un usuario compre el mismo juego una vez por cada una.

Las soluciones propuestas para la base de datos documental son las siguientes:

- *Permitir que un usuario valore un juego y pueda ver la valoración general del mismo*: Para soportar una valoración dentro de nuestro diseño debemos primero cambiar nuestro sub-documento de compras dentro del usuario para que ahora también soporte una valoración. Este cambio es relativamente sencillo de implementar, pero de la forma que están diseñadas las compras de los usuarios, va a ser necesario migrar las compras de un arreglo de identificadores de juegos a un arreglo de documentos con dos propiedades: *game* haciendo referencia al identificador del juego y *rating*. Luego debemos agregar dos atributos en el documento del juego que sean *avg_rating* y *rating_count*. No será necesario realizar ninguna migración, pero si cada vez que se agregue una nueva valoración va a ser necesario guardarlo en la compra del usuario e impactarlo dentro del juego objetivo aumentando la cantidad de valoraciones y recalculando el promedio.
- *Permitir que un usuario deje un comentario en el juego*: Para implementar este cambio basta con agregar una sub-colección dentro del juego que almacene el identificador del usuario que realizó el comentario y el comentario en sí.
- *Permitir que un usuario acceda a los datos de facturación de sus compras*: Para implementar este cambio va a ser necesario cambiar nuestro sub-documento de compras dentro del usuario para que ahora también soporte almacenar la información de pago. Al igual que en el primer punto, va a ser necesario migrar las compras de un arreglo de identificadores de juegos a un arreglo de documentos. Luego basta con simplemente agregar los nuevos campos a soportar.
- *Permitir que un juego exista para una o mas plataformas*: Para implementar este cambio tenemos dos opciones, agregar un campo en el juego que indique de que plataforma es ese juego, o agregar una sub-colección de plataformas para la cual está disponible ese juego. La más sensata sería la primera, ya que futuros cambios como valoraciones, comentarios o mismo las compras del juego siempre se relacionan a una plataforma en específico. En ese caso, si queremos tener un juego para tres plataformas, tendríamos tres entradas de juegos que difieren en la plataforma.

Cuadro II: Tiempo de pruebas de rendimiento

Test	**Tiempo**	
	Documental	**Grafo**
Juegos	*9.60 segundos*	*13.36 segundos*
Usuarios	*7.06 segundos*	*9.66 segundos*
Genérica	*13.18 segundos*	*32.94 segundos*

Para la base de datos de grafos se proponen las siguientes soluciones para representar las variaciones de requerimientos antes detalladas.

- *Permitir que un usuario valore un juego y pueda ver la valoración general del mismo*: en el diseño actual de la base de datos, la compra de un juego está representada por la relación del tipo *PURCHASED* que realciona un nodo *User* con uno *Game*. Agregar una propiedad de tipo *Integer* que sea cargada cuando el usuario valora el juego es una solución que modela el cambio planteado. Como la valoración general del juego se pretende sea visualizada por cualquier usuario, es necesario que sea almacenada en el nodo que representa al juego y actualizada cada vez que un usuario lo valora, o bien ser calculada cada vez que se obtienen los datos del juego.
- *Permitir que un usuario deje un comentario en el juego*: a diferencia de la valoración, se permite que un usuario deje varias comentarios en un mismo juego. Además no se presenta la restricción de ser un juego que haya comprado. Como solución se propone agregar una relación *COMMENTED* que relacione un nodo *User* con uno *Game*. Cada relación *COMMENTED* debe tener una propiedad de tipo *String* con el contenido del comentario.
- *Permitir que un usuario acceda a los datos de facturación de sus compras*: como cada compra tiene una facturación, la información deseada puede ser almacenada en una propiedad en las relaciones *PURCHASED*.
- *Permitir que un juego exista para una o mas plataformas*: en el diseño planteado se asume que solo se cuenta con juegos para PC. Para contemplar el cambio planteado se propone agregar nodos etiquetados con *Platform* para cada plataforma existente. Cada juego tiene relaciones *AVAILABLE_IN* con cada nodo que represente una plataforma en la que se encuentra disponible. Para modelar la compra del juego para cada plataforma, se propone agregar una propiedad en la relación *PURCHASED* que indique en cual se realiza. Otra opción sería agregar al nodo *Game* una propiedad *Platform* que determine para que plataforma es ese mismo juego y un juego está disponible para varias plataformas se crean nodos distintos para cada plataforma. Esto puede ser así si se quieren mantener las compras por plataforma.

Como se puede ver, ambas soluciones son resistentes de cierta manera a cambios con algunos cambios siendo más amigables para un modelo de base de datos que para otro. La diferencia más grande se puede ver en los comentarios, donde impactar ese cambio en una base de datos documental es relativamente más simple que en una de grafo. A su vez, también impactar el cambio de las valoraciones en una base de datos de grafo resulta un poco más simple que en la documental, ya que en la documental hay que agregar nuevos atributos en el juego, pero en la de grafo ese calculo se puede hacer relativamente simple recorriendo las relaciones.

V. Conclusiones y trabajo futuro

Esta sección busca darle un cierre a nuestro trabajo, comparando ambas soluciones y proponiendo mejoras para cada una. Por ultimo se plantea una posible solución que permita sacar el mejor provecho de ambas base de datos.

En el contexto general de las recomendaciones, un modelado con grafos presenta ventajas respecto al resto: facilidad de visualización y entendimiento, rápido desarrollo o algoritmos sobre grafos que permiten resolver el problema eficientemente. Por esto, la base de datos de grafos utilizada resulta mejor que la documental en cuanto a este punto. Obteniendo mejores tiempos y pudiendo resolverlo con una sola consulta.

Si bien se indica que el calculo de las recomendaciones es un punto a destacar de la base de datos de grafos, se plantea como trabajo futuro sacarle más provecho a lo que Neo4j provee: algoritmos de *clustering* sobre grafos o de similitud entre nodos. Estos algoritmos pueden encontrar patrones o grupos de nodos que compartan características y de forma eficiente poder determinar los juegos a recomendar.

Una diferencia entre la implementación sobre *MongoDB* y la implementación sobre *Neo4j* es el nivel de abstracción que se maneja para acceder a la base de datos. Por un lado, con *Mongoose*[19] se tienen validaciones, construcción de consultas, middleware y otras características que dotan de robustez a la implementación. En cambio, para consumir los datos de la base de grafos se usa el driver que provee Neo4j, escribiendo consultas parametrizadas, lo que carece de seguridad y controles varios.

Una mejora propuesta referido a esto es el uso de un driver o librería que permita un acceso mas robusto y seguro, como por ejemplo *Drivine*[20]. Otra opción es agregar controles e implementar abstracciones sobre el driver utilizado.

A lo largo de este trabajo se exponen puntos donde cada una de las soluciones logra mejor desempeño respecto de la otra. En términos generales MongoDB brinda una implementación mas robusta y mucho mas mantenible que Neo4j. Por otro lado, la de grafos permite visualizar la información de una forma mas amigable, y ejecutar algoritmos de grafos de forma mas eficiente.

Para la realidad planteada, se podría mantener toda la información en la base de datos de MongoDB y además mantener parte de la información, mas específicamente lo relacionado a las recomendaciones en una base de datos de grafos, manteniendo la consistencia entre ambas. Herramientas como *apoc.mongodb*[21] permiten realizar esto fácilmente.

[19]https://mongoosejs.com/
[20]https://neo4j.com/blog/introducing-drivine-graph-database-client-for-node-js-and-typescript/
[21]https://neo4j.com/labs/apoc/4.3/overview/apoc.mongodb/