Evaluación del patrón *Polyglot Persistence* en una aplicación de mediano porte

Grupo 5: Santiago Acevedo, Donato Aguirre *Instituto de Computación*
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

Resumen

Este documento describe el proceso de implementación del proyecto final de la materia bases de datos no relacionales, donde se aborda la descripción de la solución propuesta, así como los principales desafíos encontrados.

I. Introducción

Desde la tercera revolución de las bases de datos que comenzó en el 2005, se reconoce que una sola arquitectura de datos no es suficiente para cubrir la variedad desafíos que trajo consigo la era digital moderna, dando nacimiento a diversas alternativas al modelo relacional.

En los últimos años, muchas empresas aclamadas por sus productos tecnológicos han llevado el lema "*the best tool for the job*"("la mejor herramienta para la tarea en cuestión") aún más allá, promocionando la construcción de aplicaciones políglotas, esto es, que hacen uso de diferentes tecnologías de bases de datos en simultáneo.

En este trabajo se busca descubrir la implementación óptima para un sistema de pequeña a mediana escala y un cierto conjunto de requerimientos. En particular, se quiere verificar si la construcción de una aplicación donde algunos requerimientos se resuelvan utilizando una base de datos y otros otra, resulta en mejor rendimiento, ponderando también la experiencia del desarrollador. Para encontrar la combinación óptima, primero se implementa el sistema en diferentes versiones, utilizando una base de datos relacional (Postgres), una documental (Mongodb) y finalmente una de grafos (Neo4j).

Tras una fase de experimentación donde se comparan los tiempos de respuesta bajo estrés, los tiempos de carga de datos y la experiencia del desarrollador, se concluye que Neo4j destaca por la legibilidad del código de las consultas y su rendimiento en específicamente en uno de los requerimientos. Mongo destaca por su rendimiento en otro de los requerimientos. Postgres es consistentemente satisfactorio en todos los aspectos. Sin embargo, se concluye que para la escala y requerimientos estudiados, si el sistema se implementa con Postgres, no obtiene ganancias substanciales de la introducción de las otras dos que necesariamente justifique la complejidad agregada.

El documento se organiza en seis secciones, comenzando con la sección corriente, que corresponde a la introducción I. En la sección II se presentan los trabajos relacionados. En la sección III, se introduce la temática y los requerimientos de la aplicación. En la sección IV, se expande en las decisiones y detalles que conciernen a la implementación. En la sección V, se describen la fase de análisis presentando los resultados obtenidos. Finalmente, en la sección VI se recopilan las conclusiones y se plantea posible trabajo futuro.

II. Trabajos relacionados

Este trabajo expande en el patrón *"Polyglot Persistence"* que, según Martin Fowler, fue originalmente planteado por Scott Leberknight en 2008 [9] [3]. Scott se inspiró en el término *"Polyglot Programming"* [2], que expresa que las aplicaciones deben hacer uso de los diferentes lenguajes de programación para beneficiarse del hecho que diferentes lenguajes son apropiados para atacar diferentes problemas. La idea de *"Polyglot Persistence"* es la misma pero aplicada a los datos: Utilizar el almacenamiento de datos correcto en las partes de la aplicación que corresponda. En lugar de utilizar un solo tipo de almacenamiento y un solo tipo de consistencia, el patrón busca flexibilidad, incorporando *"NoSQL data stores"* para satisfacer requerimientos específicos. Con la aparición de micro-servicios el término se popularizó aún más, remarcando que cada micro-servicio puede utilizar su propia tecnología de almacenamiento, de manera transparente para sus consumidores.

En su planteo original el patrón no define una única forma de implementación, al momento ya habiéndose sido aplicado de diferentes maneras por las grandes empresas. El artículo [6] es uno de los varios donde se presentan diferentes formas de implementación del patrón ya sea en sistemas distribuidos o dentro de la misma aplicación.

Sin embargo, su utilización no viene sin sus varias desventajas, que van desde la pérdida de la consistencia y atomicidad en algunos casos, hasta el aumento de la complejidad y costo del mantenimiento del sistema, por lo que la decisión de introducirse debe ser consciente.

En los últimos años, muchos libros y artículos de empresas tecnológicas reconocidas han promocionado sus experiencias implementando este patrón. Dado que la mayoría de los proyectos en la práctica no tienen esta escala, no es fácil determinar la línea de cuándo amerita la introducción de estas técnicas.

Se han llevado acabo algunos experimentos y trabajos que comparan soluciones en sistemas políglotas, como es el ejemplo del artículo encontrado en [7]. En él se hace una comparación entre un sistema clásico implementado en ArangoDB versus un enfoque políglota construido en base a Neo4j y Mongo, concluyendo que el sistema políglota presenta mejor rendimiento para una cantidad de datos grande.

La motivación de este proyecto es profundizar en el caso de una aplicación estándar de mediano porte, considerado a partir del momento donde los datos almacenados ya no quepan confortablemente en memoria. Se pone énfasis en los atributos de calidad rendimiento, escalabilidad y mantenibilidad del código de las consultas, ignorando enteramente la consistencia e integridad.

A su vez, varias publicaciones comparan el rendimiento de ciertas consultas entre bases de datos. Por ejemplo, el artículo de la página de Neo4j [11] cita que la popular consulta 'amigos de amigos de amigos' es 180 veces más performante en Neo4j que en Postgres. En este estudio se incorpora un requerimiento de este tipo con la hipótesis de que la adición de Neo4j para resolverlo será conveniente.

III. Descripción de los requerimientos de la aplicación

La temática de la aplicación gira entorno al mundo empresarial, con la misión de brindar, tanto al emprendedor, como al inversor, herramientas de análisis para apoyar sus decisiones. Se relevaron tres requerimientos funcionales que la plataforma debería cumplir para prototipar una solución. Estos son:

- Visualización del perfil de una *startup*: Dado el nombre de una empresa, obtener sus datos básicos y la lista de sus rondas de inversión, con los datos de los inversores que participaron en cada una.
- Recomendar potenciales inversores para una *startup*: Dada una ciudad y un rubro de empresa, listar los veinte inversores que han invertido la mayor cantidad de veces en esa ciudad y rubro, ordenados por la cantidad de inversiones de forma descendente.
- Recomendar nuevas inversiones a un inversor: Dado el *id* de un inversor, listar veinte recomendaciones para su próxima inversión, esto es, las empresas con la mayor cantidad de inversiones hechas por sus co-inversores.

IV. Descripción de la solución

IV-A. Conjuntos de datos utilizados

La fuente más popular y fiable para obtener información empresarial sobre compañías privadas y públicas es Crunchbase [1]. Esta plataforma permite a sus usuarios *enterprise* exportar toda la información actualizada en formato CSV. Sin embargo, para minimizar costos, en el proyecto se utilizó un *dump* de Crunchbase del año 2013 que tiene el mismo formato pero fue extraído de la plataforma Kaggle [5] de manera gratuita. Dicho *dump* se encuentra disponible en [4].

El *dump* contiene once archivos en formato CSV, de los cuales solo tres se determinaron necesarios para resolver los requerimientos relevados. Una descripción más detallada del formato de los archivos se apreciar en el anexo en la sección VII-A. El tamaño aproximado de los archivos del *dataset* es el siguiente:

Archivo	Tamaño MB	Cantidad filas	Cantidad columnas
Objects	285	463k	40
Funding Rounds	13	53k	23
Investments	6	81k	6

Dado a que el *dump* contiene algunas filas corruptas, esto es, *foreign keys* a filas inexistentes en los otros archivos, fue necesario realizar una sanitización previa. Para esto, se realizó un pequeño *script* en python para eliminar estas filas. El tamaño del *dataset* resultante siguió siendo representativo dado que los datos corruptos eran menos de mil en más de 400 mil registros.

IV-B. Implementación de la aplicación

Para alcanzar los objetivos del proyecto no fue necesaria la construcción de una interfaz de usuario. Se puso énfasis en la implementación del lado *back-end* del sistema. Se construyó una API en NodeJS, utilizando los *drivers* para Javascript de cada una de las tecnologías de bases de datos, sin la ayuda de un *ORM*.

IV-C. Modelado de las bases de datos

En esta sección se describe el modelado de la solución en cada paradigma junto con su justificación.

En los tres modelos se determinó necesario soportar que un inversor, ya sea persona, organización financiera o empresa, invierta en una ronda de inversión que puede pertenecer a una empresa, producto de una empresa o entidad financiera.

De las empresas, productos y entidades financieras interesa principalmente su nombre, categoría, ciudad, país, dirección, página web, descripción, logo, entre otros. De las personas se conoce su nombre, ciudad, página web, etc.

De las rondas de inversión se conoce su fecha, para qué empresa, producto o entidad financiera recaudan, el tipo de ronda (Serie A, B, C, etc.), y el total recaudado. Se conoce qué inversores, ya sea individuo, entidad financiera o empresa, participaron en una ronda. Lamentablemente Crunchbase no especifica la cantidad que invirtió cada inversor específico.

IV-C1. ***Modelado relacional:*** En el modelado relacional se representaron tres entidades que se traducen a tres tablas de Postgres. Estas son `Objects`, `FundingRounds` e `Investments`.

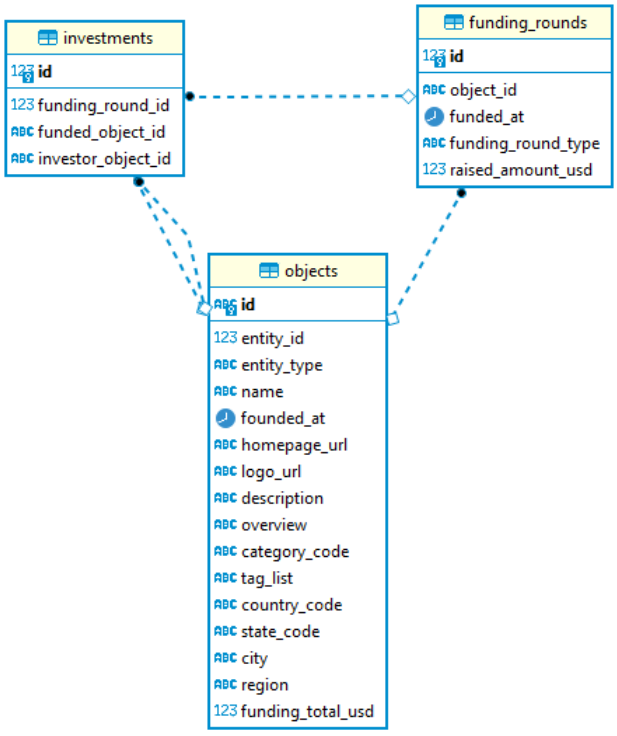

Figura 1: Modelo Postgres

La tabla `Objects` incluye una columna `entity_type` que puede tomar el valor de "`Person`", "`Company`", "`Product`" o "`FinancialOrg`". Esto es para soportar organismos que invierten y reciben inversiones al mismo tiempo. También permite distinguir inversiones en una empresa o en un producto específico de una empresa. Dado a que los diferentes tipos de `Objects` comparten la mayoría de los atributos, la cantidad de `null` en la tabla no se considera un problema.

`Investments` conecta una `FundingRound` con un `Object` que representa el inversor, conformando una relación N-N.

Un aspecto a notar es la de-normalización en la tabla `Investments`, donde se guarda el `investor_object_id` para poder accederlo directamente, evitando tener que hacer *join* con la tabla de `FundingRounds` para obtenerlo cada vez. La

aplicación de esta técnica tuvo un impacto positivo en el rendimiento de las tres consultas.

Con el mismo fin de optimizar las consultas, se indexaron todas las claves primarias y foráneas, así como otras columnas por las que se busca como *objects.name*. La columna *objects.name* es de tipo *CITEXT* para permitir la búsqueda ignorando el *case* por defecto, pero pudiendo hacer el uso del índice. También se colocó un índice compuesto por *objects.category_code* y *objects.city* ya que la segunda consulta busca por las dos al mismo tiempo.

IV-C2. ***Modelado documental****:* Al momento del diseño de la base Mongo se priorizó la primer consulta, esta es, la visualización del perfil de una empresa. Para eso se decidió modelar una única colección, `objects`, cuyos documentos se correspondan con el *JSON* final a retornar al buscar una empresa. Se hizo uso del patrón *Embedded Document* donde se definieron las `funding rounds` y los datos de sus `investors` dentro de cada `object`. Se siguió la recomendación en la documentación oficial de mongo, que se puede encontrar en [10].

Un ejemplo de un documento de la colección figura en el listado 1.

Listado 1 Ejemplo de documento Mongo

```
{
  "id": "c:1",
  "name": "Wetpaint",
  "foundedAt": "2005-10-16T02:00:00.000Z",
  "homepageURL": "http://wetpaint-inc.com",
  "logoURL": "http://s3.amazonaws.com/crunchbase_prod_assets/assets/images/resized/0000/3601/3601v1-max
      -250x250.png",
  "description": "Technology Platform Company",
  "overview": "Wetpaint is a technology platform company that uses its proprietary state-of-the-art...",
  "categoryCode": "web",
  "tagList": "wiki, seattle, elowitz, media-industry, media-platform, social-distribution-system",
  "countryCode": "USA",
  "stateCode": "WA",
  "city": "Seattle",
  "region": "Seattle",
  "fundingTotalUsd": "39750000",
  "fundingRounds": [
    {
      "id": 888,
      "type":"series-a",
      "raisedAmountUSD": "5250000",
      "fundedAt": "2005-09-30T03:00:00.000Z",
      "investors": [
        {
          "id": "f:430",
          "name": "Frazier Technology Ventures",
          "type": "FinancialOrg"
        }
      ]
    }
  ]
}
```

En este modelo también se colocaron índices por las claves buscadas, como *objects.id* y *objects.name*. Por el mismo motivo que en el caso de Postgres, se colocó un índice compuesto por *objects.category_code* y *objects.city*. También fue necesario indexar por *fundingrounds.investors.id* para permitir que los *lookups* performen bajo un tiempo aceptable.

IV-C3. ***Modelado de grafos****:* Para la base Neo4j, se decidió modelar *city* y *category* como nodos separados para poder sacarle provecho a la segunda consulta, donde se sugieren inversores para un emprendimiento considerando su historial de inversión en la ciudad y categoría. Sin embargo, como se describirá en la sección de experimentación, esta decisión pudo haber tenido un pequeño impacto negativo en el rendimiento a causa de perder la oportunidad de utilizar un índice compuesto para encontrar objetos dada una ciudad y categoría.

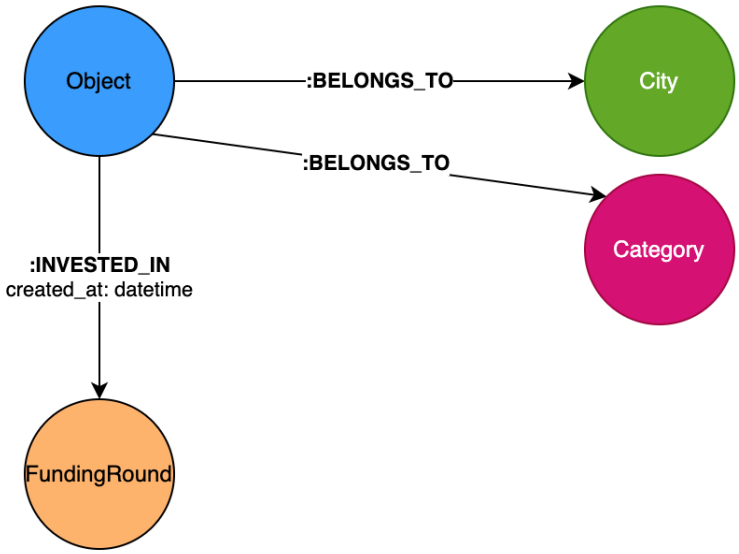

Figura 2: Modelo Neo4j

Dado a que el *dataset* cuenta con filas de *Objects* para los que no se conoce su ciudad y categoría, se decidió modelar un nodo de ciudad y un nodo de categoría que tuviesen el nombre *Unknown*. De esta manera, todos los *Objects* tienen una relación con una ciudad y una categoría y se permite que si se desea, se localicen rápidamente los *Objects* para los cuales no se conoce su ciudad y categoría.

Los índices creados fueron por *object.id*, *object.name*, *city.name*, *category.name* y *fundinground.id*.

IV-D. Carga de Datos

En esta sección se describe el procedimiento seguido para la carga de datos. En el caso de Postgres y Mongo, se implementaron y compararon dos enfoques distintos para cada una, en la búsqueda de un mecanismo performante y escalable. Todos los procedimientos se encuentran automatizados y versionados dentro del repositorio.

IV-D1. ***Base Postgres****:* Como se mencionó anteriormente, para la carga de la base de datos Postgres se probaron dos enfoques distintos.

El primero es a fuerza bruta, realizando una operación `INSERT` para cada fila de cada CSV, el cual presentó tiempos de ejecución de más de 15 minutos.

El segundo enfoque es aquel utilizado en la práctica y recomendado por la comunidad, que implica el uso de Streams para la lectura e inserción. Esta técnica permite consumir el archivo CSV de a tandas de filas, como un flujo continuo, evitando cargar todo el contenido del archivo a memoria de una sola vez. A medida que se lee las tandas, se las procesa e inserta a la base en *bulk*, resultando mucho más performante y escalable. Con este enfoque, la carga de las tres tablas finaliza en menos de un minuto.

IV-D2. ***Base Mongo****:* En el caso de Mongo, también se experimentó con dos enfoques diferentes. Ambos utilizan la base Postgres como la fuente original, como se realiza normalmente en la práctica.

El primer enfoque itera sobre cada fila de la tabla `Objects`. Para cada fila se realiza una consulta para formar la estructura final del documento a insertar en Mongo, esto es, un JSON con los datos de la empresa con sus rondas e inversores. Luego se realiza una consulta a la base Mongo para insertar el documento, y se procede a repetir para las demás filas. Este algoritmo presentó tiempos de ejecución de hasta 5 minutos.

El segundo enfoque se basa en primero exportar desde Postgres un JSON conformado por un array con los documentos a insertar. Esto se hizo mediante la consulta 22 que figura en el anexo. Luego se utiliza la función `COPY` de Postgres para guardar ese JSON en un archivo temporal, como se aprecia en el siguiente código:

Listado 2 Export Postgres a Mongo

```
COPY (
     SELECT array_to_json(array_agg(results))
     FROM (
       ${sqlQuery}
     ) results
   ) TO '${path}' WITH (FORMAT text, HEADER FALSE)`
```

Luego, se utiliza el comando `mongoimport` para insertar cada elemento del array del archivo en la nueva colección, ejemplificado a continuación. Este algoritmo finalizó en 33 segundos, donde 11 segundos corresponden a la exportación de los datos de Postgres al archivo JSON.

Listado 3 Export Postgres a Mongo

```
mongoimport --uri mongodb://localhost:27017/bdnr --collection objects --jsonArray data/objects.json
```

IV-D3. ***Base Neo4j****:* El *script* de carga de datos de Neo4j asume que se corrió previamente el *script* para Mongo, ya que utiliza el mismo archivo JSON que se exportó desde Postgres. La carga de los datos del *array* se hizo mediante la consulta encontrada en el Anexo en 22 la cual utiliza la función `apoc.load.json`. Este algoritmo finalizó en 41 segundos.

IV-E. Descripción de los endpoints

En esta sección se describe la especificación de los *endpoints* implementados.

El primer *endpoint* se utiliza para obtener el perfil de una compañía y refiere al primer requerimiento descrito en la sección III. Este *endpoint* se utiliza mediante un *GET*, y requiere el comienzo del nombre de una empresa por la que se busca en los parámetros de la *request*. La búsqueda es indiferente a mayúsculas y minúsculas.

La firma de la consulta es la siguiente: `GET /company?name=companyName`.

Un ejemplo de petición y respuesta es:

Listado 4 Request Comapany

```
req: {
     "method": "GET",
     "url": "/v2/company?name=Fitbit",
     "query": {
       "name": "Fitbit"
     },
   }
```

Listado 5 Response Comapany

```
{
   "company": {
      "_id": "62bcbebe756234fc61bbf926",
      "id": "c:10015",
      "name": "Fitbit",
      "founded_at": "2007-10-01",
      "homepage_url": "http://www.fitbit.com",
      "logo_url": "http://s3.amazonaws.com/crunchbase_prod_assets/assets/images/resized/0002/4083/24083v4
          -max-250x250.png",
      "description": null,
      "overview": "Fitbit inspires people to exercise more, eat better and live healthier lifestyles..",
      "category_code": "health",
      "country_code": "USA",
      "state_code": "CA",
      "city": "San Francisco",
      "region": "SF Bay",
      "funding_total_usd": 68069200,
      "entity_type": "Company",
      "funding_rounds": [
         {
            "id": 3619,
```

```
            "raised_amount_usd": 2000000,
            "funded_at": "2008-10-10",
            "type": "series-a",
            "investors": [
               {
                  "id": "f:74",
                  "name": "True Ventures",
                  "entity_type": "FinancialOrg"
               },
               {
                  "id": "f:737",
                  "name": "SoftTech VC",
                  "entity_type": "FinancialOrg"
               }
            ]
         },
         ...

      ]
   }
}
```

El segundo *endpoint* corresponde al segundo requerimiento y se utiliza para recomendar potenciales inversores para una *startup* dado una categoría y una ciudad. Se debe introducir el nombre completo de la categoría y la ciudad por la que se filtra.

La firma es la siguiente: `GET /investors?category=categoryName&city=cityName`.

Un ejemplo de petición y respuesta es:

Listado 6 Request Investors

```
req: {
     "method": "GET",
     "url": "/v2/investors?category=software&city=London",
     "query": {
       "category": "software",
       "city": "London"
     }
   }
```

Listado 7 Response Investors

```
{
   "investors": [
      {
         "id": "f:2469",
         "name": "Seraphim Capital",
         "entity_type": "FinancialOrg",
         "investments_count": 5
      },
      {
         "id": "f:2485",
         "name": "YFM Venture Finance",
         "entity_type": "FinancialOrg",
         "investments_count": 5
      },
      {
         "id": "f:998",
         "name": "DFJ Esprit",
         "entity_type": "FinancialOrg",
         "investments_count": 5
      },
      ...
   ]
}
```

El tercer *endpoint* corresponde al tercer requerimiento y se utiliza para recomendar nuevas inversiones a un inversor, utilizando el identificador del inversor.

La firma de la consulta es la siguiente: `GET /investmentRecommendations?investorId=investorId`.

Un ejemplo de petición y respuesta es:

Listado 8 Request Investments Recommendations

```
req: {
     "method": "GET",
     "url": "/v3/investmentRecommendations?investorId=f%3A1294",
     "query": {
       "investorId": "f:1294"
     },
   }
```

Listado 9 Response Investments Recommendations

```
{
   "recommendations": [
      {
         "id": "c:135515",
         "name": "Romotive",
         "matches_count": 4
      },
      {
         "id": "c:29052",
         "name": "Graphicly",
         "matches_count": 4
      },
      {
         "id": "c:4060",
         "name": "Univa UD",
         "matches_count": 4
      },
      ...
   ]
}
```

IV-F. Implementación de las consultas

Para la implementación de las consultas se iteró para encontrar soluciones lo más optimizadas posibles, experimentando también en cuanto a los índices que se colocaron. Los resultados se validaron a mano para algunas entradas sencillas y se compararon entre las tecnologías para validar que retornaran lo mismo. A continuación, se describe brevemente la implementación de cada consulta a alto nivel. En los listados 13, 16 y 19 se encuentra el código completo de las consultas de Postgres, Mongo y Neo4j respectivamente.

IV-F1. ***Consulta 1****:* Para resolver el primer requerimiento, la API requiere retornar un JSON con los datos la empresa buscada por el comienzo del nombre, y debe incluir los datos de sus asociaciones de manera anidada.

En Mongo es trivial, debido a que el documento ya contiene la estructura final del *JSON* a retornar, alcanzando con un `findOne` con un filtro sobre el atributo `name` utilizando una expresión regular.

En el caso de Postgres y Neo4j resultó más fácil armar el *JSON* directamente desde la consulta que implementar código de la aplicación para construirlo. Si no fuese por esto las consultas hubiesen sido más directas, utilizando `LEFT JOIN` en Postgres, y pudiendo evitar usar `WITH` en Neo4j.

En Postgres se implementó utilizando sub-consultas para las asociaciones, y la función `array_agg` para convertir el resultado de las sub-consultas a un array para que el resultado final tenga el formato deseado.

En Neo4j se realiza primero un `MATCH` buscando nodos de tipo `Object` por el nombre dado, seguido de múltiples `OPTIONAL MATCH` para encontrar los nodos correspondientes a la ciudad, categoría, rondas de inversión e inversores. Es necesario `OPTIONAL MATCH` en el caso de las rondas de inversión e inversores, para asegurarse que empresas que no tengan estas relaciones sean retornadas de todas formas. Luego la consulta se delimita en partes separadas con `WITH`, para ir formando la estructura final deseada desde adentro hacia afuera. Esto es, primero definir mediante alias (`AS`) la estructura de cada `investor` de una `funding round` y luego, la estructura de la `funding round` de las empresas seleccionadas.

IV-F2. ***Consulta 2****:* Para la segunda consulta, que recomienda posibles inversores, se busca retornar los 20 inversores con la mayor cantidad de inversiones que pertenecen a una cierta categoría y ciudad.

En Postgres se resuelve fácilmente haciendo uso de `GROUP BY` por el ID del inversor, y `COUNT`, luego de haber localizado las inversiones en las empresas relevantes.

En Mongo, se implementa con *Aggregation Pipeline*. Primero se utiliza un `match` para localizar los documentos de empresas por categoría y nombre. Al igual que en Postgres, se verificó que internamente se estuviese haciendo uso del índice compuesto por estas claves. Luego es necesario hacer `unwind` de las `funding_rounds` y una vez más de sus respectivos `investors`. Hasta este punto se cuenta con un objeto por inversión, lo que permite que sean agrupados por ID del inversor utilizando `group` y contarlos utilizando `sum: 1` a medida que se agrupan.

En Neo4j, la consulta se resuelve con un solo gran `MATCH`, que busca los nodos de la categoría y ciudad dados, y navega hasta localizar todos los inversores que han invertido en empresas que pertenezcan. Luego se utiliza `WITH` para obtener un campo computado con `COUNT`. Cabe acotar que la razón por la que se puede hacer `MATCH`, en lugar de `OPTIONAL MATCH`, es que todos los objetos tienen una relación con un nodo de categoría y ciudad. Los `nulls` del *dataset* en categoría y ciudad se representaron como dos nodos con nombre `Unknown`.

Para las tres bases, los pasos finales de las consultas son muy similares, incluyendo ordenar el conjunto resultante y limitar la cantidad a retornar.

IV-F3. ***Consulta 3****:* En la tercer consulta, que recomienda nuevas inversiones, se busca retornar las 20 empresas o productos con mayor cantidad de inversiones hechas por co-inversores del inversor dado.

En Postgres, se resuelve haciendo una sub-consulta para encontrar los co-inversores, para luego obtener sus inversiones, agruparlas por empresa y contarlas. Finalmente se descartan las empresas en las que el inversor dado ya ha invertido, se ordena el conjunto resultante y se limita la cantidad a retornar.

En Mongo, se comenzó haciendo un `match` por `funding_rounds.investors.id` del ID dado, para buscar los `Objects` donde el inversor ha invertido. A continuación, se hace `unwind` de `funding_rounds` y una vez más de `funding_rounds.investors`. Hasta este punto se cuenta con un elemento por cada inversión hecha por los co-inversores del inversor dado, pero solamente en empresas donde el inversor ya ha invertido. Es por eso que luego se realiza una proyección para quedarse solamente con los IDs de los co-inversores. Luego se agrupa para obtener solo IDs únicos para hacer el siguiente paso más barato. A continuación, se realiza un `lookup` para encontrar todas las inversiones hechas por los co-inversoras. Más adelante se realiza un `group` por ID de la empresa, sumando uno en `matches_count` por cada elemento de una empresa que se agrupa, y utilizando `$addToSet` para mantener el array de IDs de inversores que participaron cada empresa. Este array luego se utiliza para remover las empresas donde ya se encuentre el inversor dado. Finalmente se ordena, se limita y se limpian los campos retornados.

En Neo4j, esta consulta requirió varias iteraciones y esfuerzo para lograr escribirla de una forma que fuese performante. La primera versión del código de la consulta era extremadamente sencilla y más legible que la actual, pero su rendimiento era diez veces peor. La versión actual realiza primero un `MATCH` para encontrar todas las empresas en las que el inversor con el ID dado invirtió. A continuación se busca la unicidad de los datos resultantes, por lo que hace uso de `COLLECT(distinct mycompanies)`, y precisando un `UNWIND` a continuación ya que el atributo computado es de tipo array. Este paso fue una de las adiciones de la nueva versión que tuvo significativas mejoras en el rendimiento. En un segundo `MATCH` se encuentran todos los inversores que invirtieron en estas empresas (co-inversores). Nuevamente se aplica `COLLECT(distinct coinvestors)` y `UNWIND` para obtener los co-inversores una sola vez cada uno y optimizar los siguientes pasos. En un tercer `MATCH` se buscan todas las empresas en las que los co-inversores invirtieron. Luego se quitan las empresas en las que el inversor dado ya invirtió. Luego se utiliza `WITH` para poder agrupar por ID del inversor y así contar cuántos co-inversores han invertido en cada una. Finalmente, se ordena el conjunto resultante y se limita la cantidad a retornar.

V. Experimentación

En esta sección se comparan las versiones en tres aspectos diferentes: Tiempo de carga de datos, rendimiento de las consultas y experiencia del desarrollador. Todas las pruebas fueron realizadas en el mismo hardware, bajo las mismas situaciones y con los mismos recursos asignados.

V-A. *Hardware y versiones utilizadas*

A continuación se especifica el hardware y versiones del software utilizados durante la experimentación.

`Hardware y sistema operativo:`

- Sistema Operativo: MacOs Monterrey
- Procesador: 3.1GHz Intel Core i7 4 núcleos
- RAM: 16GB 2133 Mhz LPDDR3

`Versiones de herramientas:`

- PostgreSQL: 14.3
- Mongo: v5.0.7
- Neo4j: 4.4.8
- Node: 16.14.2

V-B. *Comparación de tiempos de carga de datos*

En primer lugar, se compararon los distintos tiempos de carga de la base de datos descritos en la sección IV-D. Para esta etapa solo se compararon las soluciones finales ya optimizadas de los *scripts* de carga de datos de cada base. En otras palabras, no se consideraron los algoritmos a fuerza bruta que se implementaron inicialmente.

Los tiempos de ejecución comprenden únicamente los pasos relacionados a la carga de datos una vez que la base ya se encuentra vacía. En otras palabras, se descontó el tiempo de eliminado de los datos existentes, y, en el caso de Mongo y Neo4j, la generación del archivo JSON a partir de Postgres. Un mayor detalle del desglose de los tiempos de cada *script* se pueden observar en el Anexo en la sección VII-B.

Base de Datos	Tiempo de carga de datos
Postgres	36s
Mongo	21s
Neo4j	41s

Cuadro I: Comparación de tiempo de ejecución de carga de datos

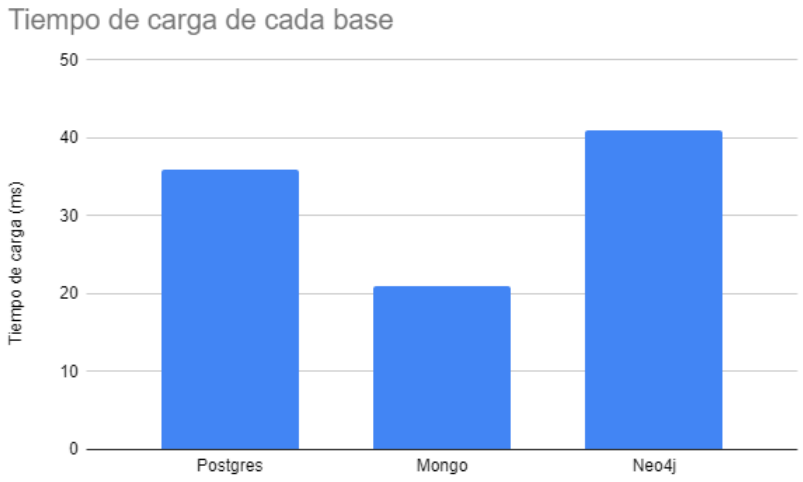

Figura 3: Comparación tiempos de carga

Como conclusión, vemos que fue posible construir un *script* de carga de datos escalable para cada una de las bases de datos, donde para el *dataset* de 300MB finalice por debajo de los 40 segundos. Se observó que recrear la base de Mongo a partir de un archivo *JSON* resultó un 50 % más rápido que en Neo4J con el mismo archivo de entrada. Sin embargo, esto seguramente se debe al hecho de que la inserción en Mongo es directa (un elemento del *array* corresponde a un documento). En Neo4j, cada elemento del *array* implica varios pasos, donde se debe buscar o crear los nodos correspondientes a la ciudad, categoría,

rondas de inversión e inversores, resultando en una gran cantidad de nodos y relaciones, y consultas para buscarlos.

V-C. Comparación de tiempos de respuesta bajo carga

A continuación, se comparó la *performance* de las tres versiones de los tres *endpoints* de la aplicación. Las pruebas de *performance* se realizaron con la herramienta *Open Source* K6 [8]. Los *scripts* se encuentran en el repositorio en la sección [12].

Para la selección de los parámetros de carga, se determinó arbitrariamente una duración de 30 segundos. Se lanzaron 10 hilos concurrentes, cada uno realizando peticiones sin descanso entre ellas, esperando a que una complete para lanzar la siguiente, todas las veces que sea posible hasta agotar el tiempo de la prueba. Las pruebas se corrieron por separado para que no se interfieran entre ellas, de manera local en el *hardware* especificado anteriormente. Las tablas que resumen los resultados de los tiempos de respuesta de cada *endpoint* son las siguientes:

	Mediana	Throughput
Postgres	8,14ms	1167,6req/s
Mongo	5,8ms	1122,1req/s
Neo4j	9,9ms	793,0req/s

Cuadro II: Comparación de rendimiento del endpoint 1

	Mediana	Throughput
Postgres	6,1ms	1000,0req/s
Mongo	5,6ms	996,6req/s
Neo4j	11,2ms	648,7req/s

Cuadro III: Comparación de rendimiento del endpoint 2

	Mediana	Throughput
Postgres	80,0ms	98,8req/s
Mongo	196,9ms	41,1req/s
Neo4j	62,82ms	157,2req/s

Cuadro IV: Comparación de rendimiento del endpoint 3

De los resultados se confirmaron algunas hipótesis y se invalidaron otras. En lo que concierne al primer *endpoint*, todos los resultados se encontraron en el mismo orden de magnitud, con Mongo resultando un 40 % y 47 % más performante que Neo4j y Postgres respectivamente. Considerando que el modelado de Mongo fue optimizado para este propósito, se esperaba ver una diferencia incluso más notoria. Es por eso que se repitió la prueba con hasta 500 hilos concurrentes para descartar una carga poco desafiante. Sin embargo, los resultados se mantuvieron proporcionales. Se concluye que, o la carga es aún insufienciente, el *dataset* no tiene el tamaño suficiente, o la consulta de Postgres no es lo suficientemente costosa como para ameritar la introducción de Mongo al sistema únicamente por este requerimiento.

Con respecto al segundo *endpoint*, las tres versiones se consideran satisfactorias, con un promedio similar a los tiempos vistos en la primera consulta. Cabe destacar que Mongo obtuvo resultados muy similares a Postgres incluso cuando el código de la consulta de Mongo es, a primera vista, mucho más compleja. Neo4j performó casi un 40 % peor que las otras dos, aunque debido a la naturaleza de la consulta se esperaba lo opuesto. Sin embargo, se sospecha que la decisión de modelar la categoría y ciudad como nodos independientes en Neo4j puede haber tenido un impacto negativo, dado que se pierde la posibilidad de tener un índice compuesto como se hizo en Postgres y Mongo.

Finalmente en cuanto el tercer *endpoint*, se observaron tiempos mucho mayores a las primeras dos consultas. Esto es de esperar, dado que la consulta requiere la manipulación de muchísimos datos (todas las inversiones de todos los co-inversores de todas las rondas de inversión de un inversos dado). En este caso, Neo4j destacó, triplicando el *throughput* de Mongo y un 50 % mejor que Postgres.

V-D. Comparación de experiencia del desarrollador

Por último, se compara la facilidad de desarrollo para las tres consultas. Cabe mencionar que si bien la mantenibilidad del código no es el foco principal del estudio, en la práctica este atributo tiene un alto impacto en el costo del sistema a largo plazo, por que lo que es igual, o en algunos casos hasta más, importante que el rendimiento.

Para evaluar este atributo se le adjuntó una nota, del 1 al 5, a cada consulta realizada. Este puntaje busca denotar la complejidad cognitiva y el tiempo dedicado a construir y para entender el código de la consulta, donde 5 significa `Muy Accesible` y un 1 significa `Muy Complejo`.

A continuación se presenta una tabla con los puntajes asignados y una breve explicación de cada uno de estos.

	Consulta 1	Consulta 2	Consulta 3
Postgres	3	3	3
Mongo	5	2	1
Neo4j	3	5	4

Cuadro V: Comparación de facilidad de implementación de las consultas

La primera consulta en Mongo es trivial, debido a que el documento ya contiene la estructura final del *JSON* a retornar, alcanzando con un *findOne*. En cambio, la complejidad de la consulta en Postgres y Neo4j se vio afectada negativamente por el hecho de que la API requiere retornar un *JSON* con los datos de las asociaciones anidadas, y resultó más fácil obtenerlo directamente desde la consulta que implementar código de la aplicación para construirlo. Si no fuese por esto las consultas hubiesen sido más directas.

En lo que concierne a la segunda y tercera consulta, llegar a una primera versión que funcione en Mongo fue donde se requirió más tiempo. Los *aggregation pipelines* resultantes son extensos, sobretodo en la tercera consulta, requiriendo la adición de comentarios en cada paso para hacer acuerdo de la estrategia tomada y de qué representan los resultados intermedios.

En general, las consultas en Neo4j son más legibles y cortas, debido a la naturaleza de la sintaxis de Neo4j y las decisiones de modelado. Sin embargo, cabe destacar que al implementarlas inicialmente se cometieron errores que solo se identificaron al comparar los resultados con las otras versiones. Por ejemplo, el optimizar la tercer consulta para llegar al *throughput* descrito no fue tarea fácil y afectó negativamente a la legibilidad de la consulta, como se desarrolló en la sección IV-F.

VI. Conclusiones y trabajo futuro

El principal objetivo que se planteó inicialmente es el de descubrir la implementación más conveniente para un sistema de mediana escala y con cierto conjunto de requerimientos relevados. Para ello el sistema se implementó utilizando tres paradigmas de bases de datos diferentes: Relacional con Postgres, documental con Mongo, y de grafos con Neo4j. Para cada uno se experimentó el modelar la solución, automatizar el proceso de carga de datos utilizando diferentes estrategias, e implementar y optimizar consultas con diferentes complejidades. También se destacó la experiencia de desarrollar pruebas de carga, así como de implementar una API utilizando los *drivers* de las tres tecnologías, sin la ayuda de un *ORM*.

En lo que respecta a las conclusiones del análisis comparativo, Neo4j presentó mejores resultados en cuanto a facilidad de codificación de las consultas, pero significativamente peor rendimiento en todos las consultas menos la última, y Mongo exactamente lo contrario. Postgres presentó resultados consistentes y satisfactorios en todos los puntos evaluados. Si la prioridad se pone puramente en el rendimiento, se debe concluir que la solución final más performante consiste de un sistema políglota donde el primer y segundo *endpoint* son servidos desde Mongo, y el tercero desde Neo4j. Sin embargo, en la práctica no necesariamente se justifica la complejidad de mantener una aplicación políglota, considerando que Postgres resultó casi tan bueno como la mejor opción en todos los aspectos. Es por eso que se concluye que, en la práctica, este sistema se podría implementar utilizando solamente Postgres. En un futuro, si la escala de datos los requiere, se podría introducir Mongo y/o Neo4j.

Se logró evidenciar la popularidad de la elección de utilizar Postgres en los sistemas para *startups*. Postgres demostró ser una tecnología fácil de configurar, consultar y consistentemente performante para escalas como la de este estudio.

Trabajos a futuros pueden incluir el experimentar con escenarios más agresivos en las pruebas de carga, con un orden mayor de datos almacenados, o con otro conjunto de requerimientos para ver en qué punto Postgres deja de ser una buena opción.

VII. Anexo

VII-A. ***Columnas de los archivos CSV***

```
id
entity_type
entity_id
parent_id
name
normalized_name
permalink
category_code
status
founded_at
closed_at
domain
homepage_url
twitter_username
logo_url
logo_width
logo_height
short_description
description
overview
tag_list
country_code
state_code
city
region
first_investment_at
last_investment_at
investment_rounds
invested_companies
first_funding_at
last_funding_at
funding_rounds
funding_total_usd
first_milestone_at
last_milestone_at
milestones
relationships
created_by
created_at
updated_at
```

Listado 10 Estructura archivo Objects

```
id
funding_round_id
object_id
funded_at
funding_round_type
funding_round_code
raised_amount_usd
raised_amount
raised_currency_code
pre_money_valuation_usd
pre_money_valuation
pre_money_currency_code
post_money_valuation_usd
post_money_valuation
post_money_currency_code
participants
is_first_round
is_last_round
source_url
source_description
created_by
created_at
updated_at
```

Listado 11 Columnas del archivo Funding Rounds

```
id
funding_round_id
funded_object_id
investor_object_id
created_at
updated_at
```

Listado 12 Columnas del archivo Investments

*VII-B. **Resultado de ejecución de scripts de carga de datos***

```
└─$ yarn setup:postgres
yarn run v1.22.19
$ node bin/setup/postgres.js
[18:34:39.867] INFO: Connection succeeded
[18:34:39.920] INFO: Enabled extensions
[18:34:40.071] INFO: Dropped table objects
[18:34:40.121] INFO: Dropped table funding_rounds
[18:34:40.126] INFO: Dropped table investments
[18:34:40.211] INFO: Created table objects
[18:34:40.225] INFO: Created table funding_rounds
[18:34:40.230] INFO: Created table investments
[18:34:40.230] INFO: Start processing file: objects.csv
[18:35:08.821] INFO: Parsed 462650 rows
[18:35:08.829] INFO: Finish processing file: objects.csv
[18:35:08.831] INFO: Start processing file: funding_rounds.csv
[18:35:11.081] INFO: Parsed 52626 rows
[18:35:11.756] INFO: Finish processing file: funding_rounds.csv
[18:35:11.757] INFO: Start processing file: investments.csv
[18:35:13.229] INFO: Parsed 80245 rows
[18:35:16.316] INFO: Finish processing file: investments.csv
✨  Done in 38.17s.
```

Figura 4: Carga base de datos Postgres

```
└─$ yarn setup:mongo
yarn run v1.22.19
$ node bin/setup/mongo.js
[18:51:41.974] INFO: Finished exporting PG objects to json.
    File path: /Users/santiago/Documentos Mac /bdnr/data/objects.json
    Elapsed time: 7 seconds
[18:51:41.974] INFO: Starting to sanitize file...
[18:51:45.226] INFO: Finished sanitizing file. Elapsed time: 11 seconds
[18:51:45.339] INFO: Finished dropping objects collection
2022-06-25T18:51:46.061-0300    connected to: mongodb://localhost:27017/bdnr
2022-06-25T18:51:49.061-0300    [###.....................] bdnr.objects 42.4MB/301MB (14.1%)
2022-06-25T18:51:52.061-0300    [######..................] bdnr.objects 87.5MB/301MB (29.1%)
2022-06-25T18:51:55.062-0300    [##########..............] bdnr.objects 134MB/301MB (44.6%)
2022-06-25T18:51:58.061-0300    [##############..........] bdnr.objects 181MB/301MB (60.3%)
2022-06-25T18:52:01.061-0300    [##################......] bdnr.objects 224MB/301MB (74.6%)
2022-06-25T18:52:04.062-0300    [######################...] bdnr.objects 266MB/301MB (88.6%)
2022-06-25T18:52:06.622-0300    [########################] bdnr.objects 301MB/301MB (100.0%)
2022-06-25T18:52:06.622-0300    462650 document(s) imported successfully. 0 document(s) failed to import.
[18:51:45.744] INFO: Finished indexing objects collection
[18:51:45.744] INFO: Importing into mongodb...
[18:52:06.634] INFO: Finished importing into mongodb. Elapsed time: 32 seconds
✨  Done in 33.27s.
```

Figura 5: Carga base de datos Mongo

```
└─$ yarn setup:neo4j
yarn run v1.22.19
$ node bin/setup/neo4j.js
/Users/santiago/DocumentosMac/bdnr/data/objects.json -> /usr/local/Cellar/neo4j/4.4.8/libexec/import/objects.json
[20:15:46.657] INFO: STEP 1/4 Finished copying file into imports directory: /usr/local/Cellar/neo4j/4.4.8/libexec/import/objects.json. Timestamp: 0 seconds
[20:15:46.658] INFO: STEP 2/4 Starting to destroy existent nodes. Timestamp: 0 seconds
[20:15:46.741] INFO: STEP 2/4 Deleted all Neo4j nodes. Timestamp: 1 seconds
[20:15:46.741] INFO: STEP 3/4 Starting to create indexes. Timestamp: 1 seconds
[20:15:46.762] INFO: STEP 3/4 Created indexes successfully. Timestamp: 1 seconds
[20:15:46.763] INFO: STEP 4/4 Starting import. Timestamp: 1 seconds
52626
[20:16:27.088] INFO: STEP 4/4 Imported objects successfully. Timestamp: 41 seconds
✨  Done in 42.76s.
```

Figura 6: Carga base de datos Mongo

*VII-C. **Consultas***

Listado 13 Consulta 1 PostgreSQL

```
SELECT
  c.id,
  c.name,
  c.founded_at,
  c.homepage_url,
  c.logo_url,
  c.description,
  c.overview,
  c.category_code,
  c.country_code,
  c.state_code,
  c.city,
  c.region,
  c.funding_total_usd,
(
  SELECT array_agg(f)
  FROM (
    SELECT
      f.id,
      f.raised_amount_usd,
      f.funded_at,
      f.funding_round_type as type,
      (
        SELECT array_agg(x)
        FROM (
          SELECT
            i.id,
            i.name,
            i.entity_type
          FROM investments x
          LEFT JOIN objects i
            ON i.id = x.investor_object_id
          WHERE f.id = x.funding_round_id
        ) x
      ) AS investors
    FROM funding_rounds f
    WHERE c.id = f.object_id
    ORDER BY f.funded_at
  ) f
) AS funding_rounds
FROM objects c
WHERE lower(c.name) LIKE lower($1)
ORDER BY c.name ASC
LIMIT 1
```

Listado 14 Consulta 2 PostgreSQL

```
SELECT
  i.id as id,
  i.name as name,
  i.entity_type,
  investments_count
FROM (
  SELECT
    i.id,
    i.name,
    i.entity_type,
    COUNT(*) AS investments_count
  FROM objects c
  INNER JOIN funding_rounds f
    ON c.id = f.object_id
  INNER JOIN investments x
    ON f.id = x.funding_round_id
  INNER JOIN objects i
    ON i.id = x.investor_object_id
  WHERE c.category_code = $1
    AND c.city = $2
  GROUP BY i.id, i.name, i.entity_type
) i
```

```
ORDER BY
  investments_count DESC,
  id ASC
LIMIT 20
```

Listado 15 Consulta 3 PostgreSQL

```
SELECT
  o.id,
  o.name,
  COUNT(DISTINCT x.investor_object_id) AS matches_count
FROM objects o
INNER JOIN investments x
  ON o.id = x.funded_object_id
WHERE x.investor_object_id IN (
  -- my co-inverstors ids:
  SELECT DISTINCT
    others_investments.investor_object_id AS investor_id
  FROM investments my_investments
  INNER JOIN investments others_investments
    ON others_investments.funded_object_id = my_investments.funded_object_id
    AND others_investments.investor_object_id <> $1
  WHERE my_investments.investor_object_id = $1
)
AND o.id NOT IN (
  -- ids of companies I already invested in:
  SELECT DISTINCT my_investments.funded_object_id
  FROM investments my_investments
  WHERE my_investments.investor_object_id = $1
)
GROUP BY o.id
ORDER BY COUNT(DISTINCT x.investor_object_id) DESC, o.id ASC
LIMIT 20
```

Listado 16 Consulta 1 Mongo

```

const query = { name: {
```

Listado 17 Consulta 2 Mongo

```

const pipeline = [
    {
      $match: {
        category_code: category,
        city
      }
    },
    {
      $project: {
        funding_rounds: 1
      }
    },
    {
      $unwind: '$funding_rounds'
    },
    {
      $unwind: '$funding_rounds.investors'
    },
    {
      $group: {
        _id: '$funding_rounds.investors.id',
        id: {
          $first: '$funding_rounds.investors.id'
        },
        name: {
          $first: '$funding_rounds.investors.name'
        },
        entity_type: {
          $first: '$funding_rounds.investors.entity_type'
```

```
      },
      investments_count: {
        $sum: 1
      }
    }
  },
  {
    $sort: {
      investments_count: -1,
      _id: 1
    }
  },
  {
    $limit: 20
  },
  {
    $unset: '_id'
  }
]

collection.aggregate(pipeline)
```

Listado 18 Consulta 3 Mongo

```
const pipeline = [
  {
    $match: {
      'funding_rounds.investors.id': investorId // select the objects the investor invested in. Uses
          index
    }
  },
  {
    $unwind: '$funding_rounds'
  },
  {
    $unwind: '$funding_rounds.investors' // get one record per co-investor investment
  },
  {
    $project: {
      investor_id: '$funding_rounds.investors.id' // we only care about co-investor's ids
    }
  },
  {
    $group: {
      _id: '$investor_id' // get distinct records only
    }
  },
  {
    $set: {
      investor_id: '$_id'
    }
  },
  {
    $unset: '_id'
  }, // up until here we have the list of distinct co-investors ids
  {
    $lookup: { // get each coinvestor's investments
      from: 'objects',
      localField: 'investor_id',
      foreignField: 'funding_rounds.investors.id',
      as: 'investments_tmp',
      pipeline: [
        {
          $project: {
            _id: 0,
            id: '$id',
            name: 1
          }
        }
      ]
    }
  },
  {
    $unwind: '$investments_tmp'
```

```
    },
    {
      $project: {
        investor_id: 1,
        id: '$investments_tmp.id',
        name: '$investments_tmp.name'
      }
    }, // up until here we have an array with all coinvestor's investments. Eg: [{ investor_id: 'c:70937',
         id: 'c:189377', name: 'Polar' }]
    {
      $group: { // count number of occurrences by object id (how many investments each company got from
          the co-investors)
        _id: '$id',
        name: {
          $first: '$name'
        },
        matches_count: {
          $sum: 1
        },
        investor_ids: { $addToSet: '$investor_id' }
      }
    },
    {
      $match: {
        $expr: { $not: { $in: [investorId, '$investor_ids'] } } } // filter out objects the current investor
            already invested in
      }
    },
    {
      $sort: {
        matches_count: -1,
        _id: 1
      }
    },
    {
      $limit: 20
    },
    {
      $project: {
        _id: 0,
        id: '$_id',
        name: '$name',
        matches_count: '$matches_count'
      }
    }
  ]

  collection.aggregate(pipeline)
```

Listado 19 Consulta 1 Neo4j

```
MATCH (c:Object)
WHERE c.name STARTS WITH $name
OPTIONAL MATCH (c)-[:BELONGS_TO]->(city:City)
OPTIONAL MATCH (c)-[:BELONGS_TO]->(category:Category)
OPTIONAL MATCH (f:FundingRound)-[:BELONGS_TO]->(c)
OPTIONAL MATCH (i:Object)-[:INVESTED_IN]->(f)
WITH c, i, f, city, category
ORDER BY c.name ASC
LIMIT 1
WITH c,f,city,category,
  {
    id: i.id,
    name: i.name,
    type: i.entity_type
  } as investors
WITH c,city,category,
  {
    id: f.id,
    type: f.type,
    funded_at: f.funded_at,
    raised_amount_usd: f.raised_amount_usd,
    investors: collect(investors)
  } as funding_rounds
WITH {
```

```
    category_code: category.name,
    city: city.name,
    id: c.id,
    name: c.name,
    founded_at: c.founded_at,
    homepage_url: c.homepage_url,
    logo_url: c.logo_url,
    description: c.description,
    overview: c.overview,
    country_code: c.country_code,
    state_code: c.state_code,
    region: c.region,
    funding_total_usd: c.funding_total_usd,
    funding_rounds: collect(funding_rounds)
  } as companies
RETURN companies
LIMIT 1
```

Listado 20 Consulta 2 Neo4j

```
MATCH (i:Object)-[:INVESTED_IN]->(f:FundingRound)-[:BELONGS_TO]->(c:Object),
  (c)-[:BELONGS_TO]->(city:City { name: $city }),
  (c)-[:BELONGS_TO]->(category:Category { name: $category })
WITH i, COUNT(f) AS investments_count
RETURN i.id AS id, i.name AS name, i.entity_type AS entity_type, investments_count
ORDER BY investments_count DESC, i.id ASC
LIMIT 20
```

Listado 21 Consulta 3 Neo4j

```
MATCH
  (me:Object {id: $investorId})-[:INVESTED_IN]->(:FundingRound)-[:BELONGS_TO]->(mycompany:Object)
WITH me, collect(distinct mycompany) AS mycompanies
UNWIND mycompanies AS mycompany
MATCH
  (coinvestor:Object)-[:INVESTED_IN]->(:FundingRound)-[:BELONGS_TO]->(mycompany)
WITH mycompanies, collect(distinct coinvestor) AS coinvestors
UNWIND coinvestors AS coinvestor
MATCH
  (coinvestor)-[:INVESTED_IN]->(:FundingRound)-[:BELONGS_TO]->(othercompany:Object)
WHERE NOT othercompany IN mycompanies
WITH othercompany, COUNT(distinct coinvestor) AS matches_count
ORDER BY matches_count DESC, othercompany.id ASC
LIMIT 20
RETURN othercompany.id AS id, othercompany.name AS name, matches_count
```

Listado 22 Consulta PostgreSQL para exportar datos para Mongo y Neo4j

```
SELECT
  c.id,
  c.name,
  c.founded_at,
  c.homepage_url,
  c.logo_url,
  c.description,
  c.overview,
  c.category_code,
  c.country_code,
  c.state_code,
  c.city,
  c.region,
  c.funding_total_usd,
  c.entity_type AS entity_type,
  (
  SELECT array_to_json(array_agg(f))
  FROM (
    SELECT
      f.id,
      f.raised_amount_usd,
      f.funded_at,
      f.funding_round_type as type,
```

```
      (
       SELECT array_to_json(array_agg(x))
       FROM (
        SELECT
          i.id,
          i.name,
          i.entity_type as entity_type
        FROM investments x
        LEFT JOIN objects i
          ON i.id = x.investor_object_id
        WHERE f.id = x.funding_round_id
       ) x
      ) AS investors
    FROM funding_rounds f
    WHERE c.id = f.object_id
    ORDER BY f.funded_At
  ) f
) AS funding_rounds
FROM objects c
ORDER BY c.name ASC
```

Listado 23 Consulta carga datos Neo4j

```
CALL apoc.load.json('file:/\${jsonFileName}')
YIELD value AS value
MERGE (city:City {
  name: coalesce(value.city, "Unknown")
})
MERGE (category:Category {
  name: coalesce(value.category_code, "Unknown")
})
CREATE (o:Object {
  id: value.id,
  name: value.name,
  logo_url: value.logo_url,
  homepage_url: value.homepage_url,
  entity_type: value.entity_type,
  founded_at: value.founded_at,
  homepage_url: value.homepage_url,
  description: value.description,
  overview: value.overview,
  funding_total_usd: value.funding_total_usd,
  country_code: value.country_code,
  state_code: value.state_code,
  region: value.region
})
CREATE (o)-[:BELONGS_TO]->(city)
CREATE (o)-[:BELONGS_TO]->(category)
WITH o, value
UNWIND value.funding_rounds AS funding_round
MERGE (f:FundingRound {
  id: toInteger(funding_round.id),
  type: funding_round.type,
  raised_amount_usd: funding_round.raised_amount_usd,
  funded_at: coalesce(funding_round.funded_at, "")
})
CREATE (f)-[:BELONGS_TO]->(o)
FOREACH (investor IN funding_round.investors |
  MERGE (i:Object {
    id: investor.id
  })
  MERGE (i)-[:INVESTED_IN]->(f)
)
RETURN count(*)
```

REFERENCIAS

[1] Crunchbase. Crunchbase official page. https://www.crunchbase.com/. Accessed: June 2022.
[2] Neal Ford. Polyglot Programming - Neal Ford. http://memeagora.blogspot.com/2006/12/polyglot-programming.htmll. Accessed: June 2022.
[3] Martin Fowler. PolyglotPersistence - Martin Fowler. https://martinfowler.com/bliki/PolyglotPersistence.html. Accessed: June 2022.
[4] Kaggle. Dump de datos sobre Crunchbase 2013. https://www.kaggle.com/datasets/justinas/startup-investments?select=objects.csv. Accessed: June 2022.
[5] Kaggle. Kaggle Datasets. https://www.kaggle.com/datasets. Accessed: June 2022.
[6] Pwint Khine and Zhaoshun Wang. A review of polyglot persistence in the big data world. *Information*, 10:141, 04 2019.

[7] Neha Singh Rohit Chandra Sunil B. Shambharkar J. J. Kulkarni. Review of nosql databases and performance comparison between multimodel and polyglot persistence approach. *INTERNATIONAL JOURNAL OF SCIENTIFIC TECHNOLOGY RESEARCH*, 9:5, 01 2020.
[8] Grafana Labs. K6 Load testing tool. https://k6.io. Accessed: June 2022.
[9] Scott Leberknight. Polyglot Persistence - Scott Leberknight's Weblog. http://www.sleberknight.com/blog/sleberkn/entry/polyglot_persistence. Accessed: June 2022.
[10] MongoDB. Model One-to-Many Relationships with Embedded Documents. https://www.mongodb.com/docs/manual/tutorial/model-embedded-one-to-many-relationships-between-documents/#embedded-document-pattern. Accessed: June 2022.
[11] Neo4j. How much faster is a graph database, really? https://neo4j.com/news/how-much-faster-is-a-graph-database-really/. Accessed: July 2022.
[12] Donato Aguirre Santiago Acevedo. Repositorio BDNR Grupo 5. https://gitlab.fing.edu.uy/santiago.acevedo/bdnr-grupo5. Accessed: June 2022.