Trabajo práctico Final BD2

Integrantes:

- Santiago Andres Ponte Ahon (16123/6)
- Iñaki Recalde (16370/0)
- Francisco Godoy (16370/0)

Indice

Enunciado

El presente trabajo inició como un trabajo de promoción al cual se le extendió el enunciado para que encaje como un Trabajo Práctico Final de la materia “Base De Datos 2”.

El enunciado original era el siguiente:

> *Para el trabajo de promoción se deberá:*
>
> 1. *montar un entorno utilizando Docker*
> 2. *configurar un motor de bases de datos postgres*
> 3. *configurar un motor NoSQL Mongodb (instalación básica)*
> 4. *importar el dataset de accidentes de tránsito en cada una de las bases de datos*
> 5. *implementar una aplicación utilizando la arquitectura de base presentada en la materia*
> 6. *implementar las siguientes consultas apuntando a cada una de las bases de datos según su conveniencia:*
>
>   - *devolver todos los accidentes ocurridos entre 2 fechas dadas*
>   - *determinar las condiciones más comunes en los accidentes (hora del día, condiciones climáticas, etc)*
>   - *dado un punto geográfico y un radio (expresado en kilómetros) devolver todos los accidentes ocurridos dentro del radio.*
>   - *obtener la distancia promedio desde el inicio al fin del accidente*
>
>   *Consultas para equipos de 3 personas:*
>
>   - *devolver los 5 puntos más peligrosos (definiendo un determinado radio y utilizando los datos de los accidentes registrados).*
>   - *devolver la distancia promedio que existe entre cada accidente y los 10 más cercanos.*
>   - *devolver el nombre de las 5 calles con más accidentes.*

Naturalmente nuestro equipo era de 3 personas, por lo que también tuvimos que resolver las 3 consultas extras que el enunciado indicaba.

La extensión para que se considere trabajo práctico final se muestra a continuación:

> *El trabajo "extra" o adicional es poder pasar la info que tienen en el mongo a un elastic (también sobre docker) y poder probar dos consultas a elección de uds para ver la performance entre elastic versus mongo.*

Para fines prácticos resumimos esta extensión en los siguientes ítems:

> 7. *configurar un motor de bases de datos elasticsearch*

8. *implementar 2 de las consultas solucionadas anteriormente con mongo, pero ahora haciendo uso de elasticsearch*

9. *comparar el rendimiento entre elasticsearch y mongodb*

En su momento se consultó respecto si era necesario montar toda la aplicación en docker o si solo era necesario hacerlo con docker y se no, la aplicación como tal puede entregarse como un proyecto *mvn* típico, pero si es necesario que las bases de datos funcionen a modo de contenedores docker. Por lo que el trabajo se resolvió teniendo esto en cuenta.

DBs y su inicialización con Docker

Primero se hablará de cómo se solucionaron los puntos 1, 2, 3, 4 y 7. Los cuales hacen referencia a la construcción en docker de 3 bases de datos (PostgreSQL, MongoDB, ElasticSearch) y su respectiva inicialización.

La solución general consiste en el uso de una archivo “docker-compose.yml” para realizar las distintas DBs de la forma mas automática posible, este archivo levanta todos los servicios necesarios a la par que se asegura de enlazarlos correctamente. Naturalmente este requiere hacer uso de múltiples archivos de configuración secundarios los cuales les dará a cada servicio al iniciarlo, por lo que para evitar una mezcla de estos archivos respecto a los de la aplicación java se optó por centralizar todos los archivos relacionados a docker en una carpeta “docker” que se encuentra en la raíz del repositorio.

A continuación se explican los servicios relacionados a cada DB y como es su método de inicialización:

- PostgreSQL: Para esta DB “docker-compose.yml” tiene especificados 2 servicios. “postgres” el cual actúa como la propia DB y “pgadmin” el cual es útil para realizar la visualización de los datos de la DB desde el navegador (se puede prescindir de este último servicio si no desea visualizar los datos).
  Todos los detalles de configuración se pueden observar en el archivo “docker-compose.yml”, pero a grandes rasgos lo importante es saber que la DB postgre se encuentra en el puerto “5432”, su usuario es “root”, su contraseña también es “root” y todos los datos del contenedor se mapean a un volumen local llamado “postgres-volume”. Por otro lado pgadmin se encuentra en el puerto 80, su email de autenticación es “admin@admin.com”, su contraseña es “admin” y se puede acceder a él desde el navegador llendo a “http://localhost/browser/”
  Respecto a carga de los datos del archivo “US_Accidents_Dec19.csv” esta es solucionada mediante una opción de inicialización que el servicio soporta. Para usarlo es importante que con anterioridad se haya creado una carpeta “csv” en la carpeta “docker” y dentro de esta poner el archivo “US_Accidents_Dec19.csv” (importante que tenga ese nombre), en el archivo “docker-compose.yml” está especificado que el servicio postgre copie tanto el contenido de la carpeta “csv” como el archivo de inicialización “init-postgre.sql”. El contenedor al iniciarse ejecuta las sentencias sql que se encuentran en este archivo, sólo cuando no hay datos previos cargados en “postgres-volume”. “init-postgre.sql” crea la tabla sobre la que se cargaran los accidentes crea los distintos índices que nos interesa tener para optimizar las consultas que se realizaran sobre la DB e importa el contenido de “/csv/US_Accidents_Dec19.csv” dentro de la tabla accident.
- MongoDB: Está DB solo consiste en un único servicio llamado “mongodb”, está en el puerto “27017”, su usuario es “root”, su contraseña es “root”, al igual que postgre mapea dentro el contenido de la carpeta “docker/csv/” y 2 archivos de inicializacion “init-mongo-1.sh” e “init-mongo-2.js” que son ejecutados de forma automática al iniciar el contenedor. “init-mongo-1.sh” ejecuta una serie de sentencias bash que se encargan de importar el contenido de “/csv/US_Accidents_Dec19.csv” en la DB. “init-mongo-2.js” crea varios índices útiles entre los que se encuentra el indice “2dsphere” se requiere para realizar consultas geoespaciales.

Los datos de esta DB se pueden observar haciendo uso de la app "MongoDBCompass" (ésta no se pudo colocar en un contenedor docker por lo que se debe descargar e instalar de forma local), con ella debe establecer una conexión a "mongodb://root:root@localhost:27017/?authMechanism=DEFAULT".

- ElasticSearch: Notara que en las anteriores DB se procuró que la carga de datos se realice de la forma mas automática posible, lamentablemente los contenedores naturales de ElasticSearch no ofrecen esta posibilidad, por lo que para su correcta inicialización se requiere un mínimo de interacción de parte del usuario. Todo el proceso de instanciación está detallado en el archivo "README.md", pero mencionaremos brevemente en qué consiste.
  En "docker-compose.yml" podrá encontrar 3 servicios relacionados al ElasticSearch, estos son "Elasticsearch" (puerto: 9200-9300) "Kibana" (puerto: 5601) y "Logstash" (puerto: 9600). "Elasticsearch" es la DB como tal todos sus datos se respaldan en un volumen administrados por el propio docker llamado "elastic_data", este no aparece en él como una carpeta, pero si desea verlo o borrarlo puede hacerlo yendo a la sección "Volumes" de "Docker Desktop". "Kibana" es un cliente desde el cual se puede interactuar con la base de datos ElasticSearch, consultas, modificaciones y creación de índices son algunas de sus funciones, tambien tiene una funcion de carga de archivos csv pero solo funciona con archivos de hasta 100Mb, lo cual es un poco desalentador si consideramos el hecho de que nuestro archivo contiene mas de 2 Gb de datos. "Logstash" se presenta como solución a esto último ya que es la forma que ElasticSearch permite la carga masiva de datos, en su especificación de "docker-compose.yml" se puede observar que mapea para sí mismo el contenido dentro de la carpeta "docker/csv/", y el archivo de configuración "logstash.conf" que se encuentra en "docker/logstash/conf/", este archivo realiza la importación de "US_Accidents_Dec19.csv" y realiza algunas mutaciones en los datos que son necesarias para el manejo de consultas geoespaciales.
  Naturalmente "Logstash" ejecuta "logstash.conf" ni bien se inicia, pero como dijimos antes, para nuestro problema es imposible automatizar la importación en su totalidad. Es imposible especificar en "logstash.conf" que un campo es de tipo Date o de tipo Point (útil para las consultas geoespaciales), si dejamos que se importen por su cuenta solo podremos disponer los datos como si fueran datos de tipo "Text", lo que es poco util para las consultas mas complejas. Para solucionar esto se soluciono una secuencia de 3 pasos para levantar la DB ElasticSearch:
  1. Ejecutar la siguiente secuencia de comandos para levantar todos los contenedores excepto "Logstash"

     ```
     docker-compose up -d

     docker-compose stop Logstash

     cd .
     ```

  2. Esperar 30-60 segundos para que se inicializen los servicios e ir con el navegador a http://localhost:5601/app/dev_tools#/console para, con la consola que nos ofrece Kibana, ejecutar la siguiente consulta.

     ```
     PUT /accident

     {
     ```

```
"mappings": {
  "properties": {
    "location": { "type":"geo_point" },
    "start_time": {
      "type": "date",
      "format": "yyyy-MM-dd HH:mm:ss"
    }
  }
 }
}
```

Esta consulta establece el tipo de varios campos que nos interesa que tengan un tipo específico, teniendo en cuenta las consultas que haremos sobre ElasticSearch.

3. Iniciar el contenedor “Logstash” con el siguiente comando para que, ahora sí, “logstash.conf” se ejecute importando todos los datos con los tipos que nos interesan dadas las consultas a realizar.

Es importante que tenga en cuenta que al levantar de esta forma todos los servicios las 3 DBs van a estar ejecutando sus respectivas importaciones al mismo tiempo, esto supondrá cierta carga para el equipos, sobre todo si lo estamos usando en otras tareas mientras esto ocurre. Si lo desea puede detener los servicios “pgadmin” y “Kibana” luego de haberlos usado, su detención no afectará la carga de datos y pueden ser reiniciados las veces que quiera. Sin embargo, es importante que no reinicie ni detenga el servicio “Logstash”, dado que esto podría provocar la duplicación de todos los datos de elasticSearch, si por accidente llegase a borrarlo entonces es importante que borre el los Volúmenes que se pueden ver en “Docker Desktop > Volumes” y que repita todos los pasos del proceso de importación de datos en ElasticSearch. No obstante una vez la carga de datos finalice el servicio “Logstash” ya no será relevante por lo que si lo desea puede detenerlo.

Dadas las dimensiones del csv que se nos dio PostgreSQL probablemente tardará entre 40-60 minutos en terminar de cargar los datos, MongoDB le seguirá un tiempo casi idéntico, y ElasticSearch terminará último pudiendo tardar hasta hora y media en la carga de sus datos.

NOTA: En el archivo “README.md” del repositorio está explicado paso a paso cómo hacer para levantar todos los servicios y sus pormenores, además se explica una forma de comprobar que los datos se hayan terminado de cargar en cada DB.

Aplicación Principal e implementación de Querys

En esta sección se explica lo relacionado con la resolución de los incisos 5, 6 y 8 del enunciado.

La aplicación base consiste en un proyecto maven el cual está concentrado en la carpeta “promoTP/” del repositorio. Como se solicitó el proyecto fue construido tomando como base el proyecto que desarrollamos a lo largo de la cursada de la materia.

Si inspecciona el código podrá notar que todo está convenientemente separado en distintos paquetes, a continuación se explica brevemente el contenido de cada uno:

- com.mitocode: Aquí se halla el proceso main de la aplicación es lo que se debe ejecutar para iniciar la aplicación (Click derecho en el archivo y seleccionar “Run As > Java Aplication”). El archivo también especifica una serie de decoradores sumamente relevantes para la inicialización de la aplicación, su api y su conexión con las bases de datos.

- com.mitocode.controller: Aqui se haya el archivo “TestController.java”, esta clase, correctamente identificada con el decorador “@RestController”, especifica los distintos endpoints que la aplicación pondrá a disposición de los usuarios. para solucionarlos hace uso de un objeto que cumpla con la interfaz AccidentService que es auto inyectado gracias a la annotation “@Autowired”.

- com.mitocode.model.persistance: Aquí se hallan los modelos que se usan para mapear los datos de los distintos tipos de entidades que almacena cada tipo de DB. En este caso solo existe “Accident.java” ya que es la única entidad que nos interesa guardar en nuestras DBs.

- com.mitocode.model.schema: Las clases que hay aquí mapeos de clases que almacenan las DBs (del tipo que nos interesa que se construyan tablas en mongoDB), se trata de clases que se utilizan para mapear las respuestas de las querys que realizan los repositorios a clases java legibles por el código y que se pueden traducir a una respuesta REST apta para que la API ofrecida por la aplicación responda.

- com.mitocode.repo.elastic: Aquí se define la interfaz “ElasticsearchAccidentRepository” subclase de “ElasticsearchRepository<Accident, String>” que utilizamos para interactuar con la base de datos ElasticSearch. En la misma se definen un par de métodos personalizados que mediante anotaciones especifican consultas específicas útiles para resolver lo pedido por el enunciado.

- com.mitocode.repo.mongo: Aquí se define la interfaz “MongodbAccidentRepository” subclase de “MongoRepository<Accident, String>” que utilizamos para interactuar con la base de datos MongoDB. Al igual que en la anterior esta también define varios métodos extra que mediante anotaciones especifican consultas útiles para resolver lo pedido por el enunciado.

- com.mitocode.repo.postgre: Aquí se define la interfaz “PostgreAccidentRepository” subclase de “JpaRepository<Accident, String>” que utilizamos para interactuar con la

base de datos PostgreSQL. Al igual que en la anterior esta también define varios métodos extra que mediante anotaciones especifican consultas útiles para resolver lo pedido por el enunciado.

- com.mitocode.service.interf: Aquí se definen las interfaces de los distintos servicios que se quieran ofrecer, aunque debido al problema que se está resolviendo solo se requiere hacer uso de una unica interfaz “AccidentService.java”, esta es la que usa “TestController.java” para resolver las Querys entrantes. Notar que el uso de interfaces para la inyección de variables tiene cierta utilidad práctica, ya que permite al que la use abstraerse de su implementación concreta, si uno tiene que solucionar algo del procesamiento puede modificar al que implementa la interfaz sin miedo de que por ello tenga que modificar tambien el codigo del que la usa.

- com.mitocode.service.implementation: En este paquete se hallan las implementaciones de los servicios que se requieren, naturalmente en este caso tenemos una sola interfaz, por lo que solo definimos una implementación. “AccidentServiceImpl” resuelve haciendo uso de los distintos repositorios todas los métodos requeridos por la interfaz “AccidentService”, no necesariamente un repositorio tiene que poder resolverlo todo, el service puede usar los distintos repositorios o reemplazarlos entre sí, según le convenga.

- com.mitocode.util: Este paquete, tal como su nombre indica, contiene contiene clases y utilidades varias que nos pareció adecuado definir para un código correctamente armado, pero no que encajaban en los demás paquetes.

Hemos de decir que la implementación de múltiples DBs en un único proyecto maven fue un completo desafío, de cada implementación individual es posible encontrar cierto grado de documentación, pero muchas veces estas tienen distintos estilos y cada página parece resolver la conexión de formas distintas (aunque a veces similares). Esto de por si no es un problema tan grave, pero cuando uno intenta implementar dos BDs con métodos distintos ahí es cuando empiezan los problemas. Los tutoriales de implementación simultáneas son verdaderamente escasos, muchas veces sobre bases de datos que no nos son de interés, lo mejor que pudimos conseguir respecto a eso fue un tutorial que mostraba cómo conectar una Base de datos MySQL y MongoDB al mismo tiempo. Además, muchas veces las anotaciones que cada DB requiere para sus mapeos se superponen a las de las demás, momentos en los que hay que buscar soluciones en lo profundo de la documentación junto con el uso de cierto grado de creatividad. ElasticSearch por su parte fue un problema aún mayor ya que fue particularmente difícil hallar y comprender su documentación, además de que posee ciertas limitaciones que no son claras ni en la documentación ni en los mensajes de error, aunque esto último fue algo que ocurrió por lo menos una vez con cada DB.

Aun así las conexiones fueron resueltas y sin duda nos aportaron un montón de experiencia práctica que nos servirá para poder realizar construcciones relativamente rápidas de este tipo de servicios en el futuro.

NOTA: El archivo “promoTP/src/main/resources/application.propieties” especifica puertos y variables varias que el proyecto requiere para establecer correctamente la conexión con cada DB. Capaz le interese modificarlo si tuvo que modificar algún valor del archivo “docker-compose.yml”.

Listado de Endpoints

A continuación se explican los endpoints base que la aplicación ofrece, notara que en algunos se implementa un sistema de paginación, esto lo hicimos así por que pensamos que podría en las consultas que pueden llegar a incluir una gran cantidad de elementos devolverlos todos juntos, al paginarlos las bases de datos pueden optimizar mejor los recursos y le da cierta flexibilidad al usuario por si este no requiere la totalidad de los datos al mismo tiempo. También encontrará una sección de notas en la tabla de cada endpoint, allí encontrará detalles de esa query concreta que nos parece interesante mencionar, ya sean respecto al rendimiento de las mismas o respecto a limitaciones del framework para cosas concretas.

Url	localhost:8080/databases/accidentsBetweenTwoDates
Tipo	GET
Descripción	Devolver todos los accidentes ocurridos entre 2 fechas dadas (sin incluir las de esta última)
Parámetros	• startDate: fecha de inicio • endDate: fecha de fin • pageNumber: numero de pagina a devolver • pageSize: tamaño de las páginas
Notas	

Url	localhost:8080/databases/mostCommonConditions
Tipo	GET
Descripción	Determinar las condiciones más comunes en los accidentes (hora del día, condiciones climáticas, etc)
Parámetros	
Notas	Esta query consulta por el valor repetido en gran cantidad de los atributos de la Tabla accident en PostgreSQL, en un inicio al utilizarla en conjunto con la base de datos Entera su rendimiento era pésimo, pudiendo la misma llegar a tardar hasta 30 minutos en su ejecución. Fue sorprendente la mejora que hubo en su rendimiento al incorporar el uso de índices en la BD PostgreSQL, siendo que ahora su ejecución sobre la BD entera tarda menos de 20 segundos.

Url	localhost:8080/databases/accidentsNearAPointInARadius
Tipo	GET
Descripción	Dado un punto geográfico y un radio (expresado en kilómetros) devolver

	todos los accidentes ocurridos dentro del radio.
Parámetros	• latitude: latitud correspondiente al punto geográfico de interés • longitude: longitud correspondiente al punto geográfico de interés • radius: radio en kilómetros • pageNumber: numero de pagina a devolver • pageSize: tamaño de las páginas
Notas	Si mira la implementación java notara que la gran mayoría de las querys que devuelven listados, utilizan el tipo de retorno Page, sin embargo, por alguna razón que se nos escapa, en MongoDB cuando haces consultas de características geoespaciales y el tipo de retorno es Page, las consultas fallan detectar que dentro del radio de calculo entra mas de una cierta cantidad de elementos. Buscamos información online pero lo único que encontramos fue una consulta perdida en un foro, la cual nunca tuvo respuesta. Afortunadamente logramos solucionar este extraño fallo cambiando el tipo de retorno de esta query a Slice (Super clase de Page). Por lo que debido a limitaciones del framework de MongoDB esta es la única clase que utiliza este otro tipo de retorno para el paginado.

Url	localhost:8080/databases/averageDistanceOfAccidentsFromBeginningToEnd
Tipo	GET
Descripción	Obtener la distancia promedio desde el inicio al fin del accidente
Parámetros	
Notas	

Url	localhost:8080/databases/fiveMostDangerousPoints
Tipo	GET
Descripción	Devolver los 5 puntos más peligrosos (definiendo un determinado radio y utilizando los datos de los accidentes registrados)
Parámetros	• latitude: latitud correspondiente al punto geográfico de interés • longitude: longitud correspondiente al punto geográfico de interés • radius: radio en kilómetros
Notas	

Url	localhost:8080/databases/allAvgDistanceBetweenTop10NearestAccidents

Tipo	GET
Descripción	Devolver la distancia promedio que existe entre cada accidente y los 10 más cercanos
Parámetros	• pageNumber: numero de pagina a devolver • pageSize: tamaño de las páginas
Notas	Notará que esta query es altamente exigente ya que requiere que cada accidente de la base de datos sea comparado con todos los demás accidentes de la misma en pos de determinar la distancia promedio que hay entre todos estos, es básicamente un N^2 siendo N = (Cantidad de accidentes en la BD). Por esto se prestó especial importancia al paginado de esta query, ya que sería impensable hacer esta cuenta para todas las querys de la DB al mismo tiempo, se procesan solo las que correspondan a en la página solicitada, permitiendo dosificar así una superconsulta que probablemente tardaría horas en ejecutarse en consultas mas pequeñas con tiempos razonables de respuesta. Aun con estas medidas la misma sigue siendo particularmente exigente por lo que al ejecutarla tenga en cuenta que puede tardar varios segundos en mostrar su resultado.

Url	localhost:8080/databases/fiveStreetsWithMoreAccidents
Tipo	GET
Descripción	Devolver las 5 Calles con más accidentes
Parámetros	• pageNumber: numero de pagina a devolver • pageSize: tamaño de las páginas
Notas	

Las anteriores se corresponden a las consultas base construidas para satisfacer el enunciado básico, a continuación se especifican los endpoints correspondientes a las variaciones implementadas con ElasticSearch requeridas por la parte extendida del enunciado.

Url	localhost:8080/databases/accidentsBetweenTwoDatesElasticVersion
Tipo	GET
Descripción	Devolver todos los accidentes ocurridos entre 2 fechas dadas (sin incluir las de esta última)
Parámetros	• startDate: fecha de inicio • endDate: fecha de fin • pageNumber: numero de pagina a devolver • pageSize: tamaño de las páginas

Notas	

Url	localhost:8080/databases/accidentsNearAPointInARadius
Tipo	GET
Descripción	Dado un punto geográfico y un radio (expresado en kilómetros) devolver todos los accidentes ocurridos dentro del radio.
Parámetros	• latitude: latitud correspondiente al punto geográfico de interés • longitude: longitud correspondiente al punto geográfico de interés • radius: radio en kilómetros • pageNumber: numero de pagina a devolver • pageSize: tamaño de las páginas
Notas	Recordará que en el endpoint de la implementación base que se corresponde con este mencionamos que era imposible usar el tipo Page para esta query haciendo uso de MongoRepository, por lo que nos vimos obligados a usar para el mismo el tipo Slice. Pues resulta que en ElasticRepository falla la totalidad de las veces cuando se le pide un tipo Slice, por lo que nos vimos obligados a hacer que este devuelva un elemento de tipo Page el cual no presenta problemas para con ElasticRepository. Evidentemente el uso de Slice es minimamente mas eficiente que el de Page, ya que no tiene que consultar respecto a la cantidad total de elementos de la query, sin embargo no nos quedó opción en este caso, por lo que tendremos en cuenta esta pequeña diferencia al comparar el rendimiento de las versiones de esta Query en MongoDB y en ElasticSearch.

Notará que las consultas que elegimos para implementarlo una versión alternativa con ElsticSearch fueron la uno y la tres, ambas están resueltas con MongoDB en la solución base por lo que se adecua a lo pedido por el enunciado.

Nota: Inicialmente hasta la consulta mas básica nos llevaba una cantidad de tiempo considerable pudiendo llegar a tardar medio minuto consultar por los accidentes que ocurren entre 2 fechas. Sin embargo, grata fue nuestra sorpresa al incorporar el uso de índices en nuestras bases de datos, ya que mejoraron consultas que tardaban 30 minutos a 30 segundos y las consultas que antes tardaban 30 segundos ahora tardaban milisegundos. Sin duda la mejora que los mismos supusieron fue en extremo considerable. Además, los 40 Mb que costaba mantener cada uno en memoria sinceramente parece insignificante respecto a las mejoras que suponen por lo que terminamos agregandolos en prácticamente todos los atributos que usamos como criterios para nuestras consultas.

Notará que hacia el final de la clase “TestController” hay un endpoint extra que no encaja con ninguna de las querys requeridas. Este endpoint lo agregamos con el propósito de ejecutar una comparación de rendimiento en tiempo útil para resolver el punto 9 del

enunciado. A continuación se deja su especificación, pero se hablará mas a detalle de sus resultados en la siguiente sección.

Url	localhost:8080/databases/accidentsBetweenTwoDatesElasticVersion
Tipo	GET
Descripción	Comparación de tiempos entre MongoDB vs ElasticSearch
Parámetros	• N: cantidad de repeticiones a realizar para obtener el promedio de tiempo de cada consulta.
Notas	La query no devuelve ningún resultado, la tabla resultante se puede ver en la consola de mostrado por eclipse al ejecutar la aplicación java.

MongoDB vs ElasticSearch

Como se comentó anteriormente decidimos facilitarnos un endpoint extra que inicia una comparación entre ElasticSearch y MongoDB del lado del servidor, esto lo hicimos así ya que nos parecía injusto incluir en la comparación los tiempos de ida y vuelta extra que resultan de la ejecución de las consultas mediante el postman y similares. Al ejecutar toda la comparación del lado del servidor el mismo puede consultar de forma seguida todos los datos directamente al service minimizando en cierta medida el sesgo de tiempos causado por la comunicación que implica el uso directo de los endpoints.

También en un inicio consideramos hacer múltiples consultas a cada query de interés para así promediar los tiempos obtenidos y en consecuencia obtener valores mas representativos del rendimiento general de cada DB. Sin embargo, pronto nos dimos cuenta de que cada DB cachaba en mayor o menor medida las consultas que ejecutaba, al repetir una consulta siempre se obtenía un tiempo significativamente menor que el obtenido, y este no era un esfuerzo real para las DBs ya que estas solo reenviaban los datos que tenían cacheados, por lo que consideramos que lo mejor seria usar una cantidad de repeticiones nulas (N=1) ya que de esta forma las DBs no podrían utilizar el mecanismo de cacheo, solo nos darían un tiempo para cada consulta individual, pero el mismo sería claramente representativa del rendimiento de cada DB. También se consideró durante la comparación un tamaño de página exageradamente grande para exigir al máximo ambas DBs.

A continuación se puede ver la comparación de los resultados obtenidos al ejecutar el Script de comparación de MongoDb vs ElasticSearch.

NOTA: Es importante mencionar que los tiempos obtenidos están sujetos al equipo que utilizamos para realizar las pruebas de rendimiento, el mismo no tiene un hardware particularmente sobresaliente y estaba ejecutando al mismo tiempo los contenedores que corresponden a las distintas DBs. Por lo que es bastante probable que la confiabilidad de los resultados sea mayor si se pudiese disponer de mejor hardware y equipos individuales para cada contenedor.
Además, si bien mencionamos que usamos 1 repetición para evitar el sesgo de caché, esta repetición fue tomada varias veces, reiniciando de forma total la app y los contenedores en esas oportunidades, obteniendo siempre resultados similares a los que mostramos mas adelante. Por lo que confiamos en que los resultados mostrados no son excepciones que ocurren una única vez, sino resultados que constantemente se repiten de forma similar.

Comparación accidentsBetweenTwoDates

La primera query a comparar consiste en la obtención de todos los accidentes que se incluyan entre 2 fechas determinadas. Se evaluó el tiempo que cada DB tardaba en resolver cada consulta variando la cantidad de meses a incluir en la misma.

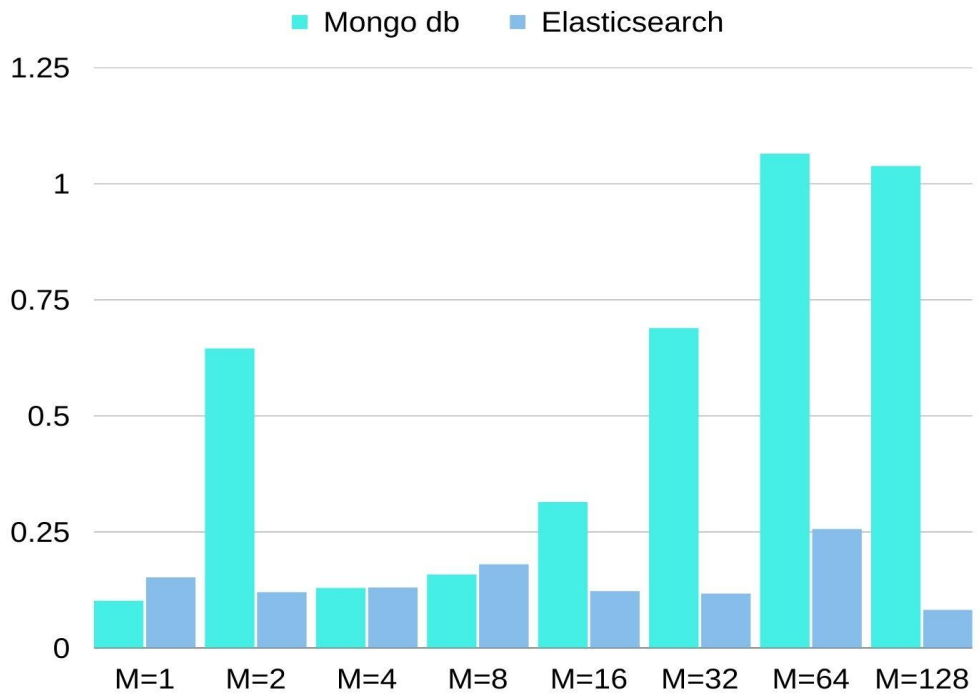
A continuación se muestra un gráfico con los resultados de la comparación, en la parte de arriba hay una breve descripción de lo que se está viendo, debajo está el resultado dado en consola por nuestro algoritmo de comparación e inmediatamente debajo se puede ver un

diagrama que contrasta la cantidad de meses incluidos con el tiempo que tardó en ejecutarse la consulta en MongoDB y ElasticSearch respectivamente.

TABLA 1

Promedio en segundos que tarda cada DB para la Consulta 1 a lo largo de 1 repetición, variando la dimensión M de la cantidad de meses que contempla el intervalo.

	M=1	M=2	M=4	M=8	M=16	M=32	M=64	M=128
Mongo db	0,1010	0,6440	0,1280	0,1570	0,3130	0,6880	1,0640	1,0370
Elasticsearch	0,1510	0,1200	0,1290	0,1790	0,1220	0,1170	0,2560	0,0820



Luego de ver el gráfico es notable el en una primera instancia MongoDB mantenía un rendimiento general ligeramente superior al de ElasticSearch, pero a medida que su la cantidad de meses fue creciendo también lo hizo el tiempo que tardaba en informar los resultados, sin embargo ElasticSearch se mantuvo el tiempo para cada consulta en un rango relativamente constante, por lo que pareciera que ElasticSearch presenta una solución mas eficiente para este tipo de querys frente a MongoDB.

NOTA: Ambas DBs cuentan con índices en las variables que involucran estas consultas.

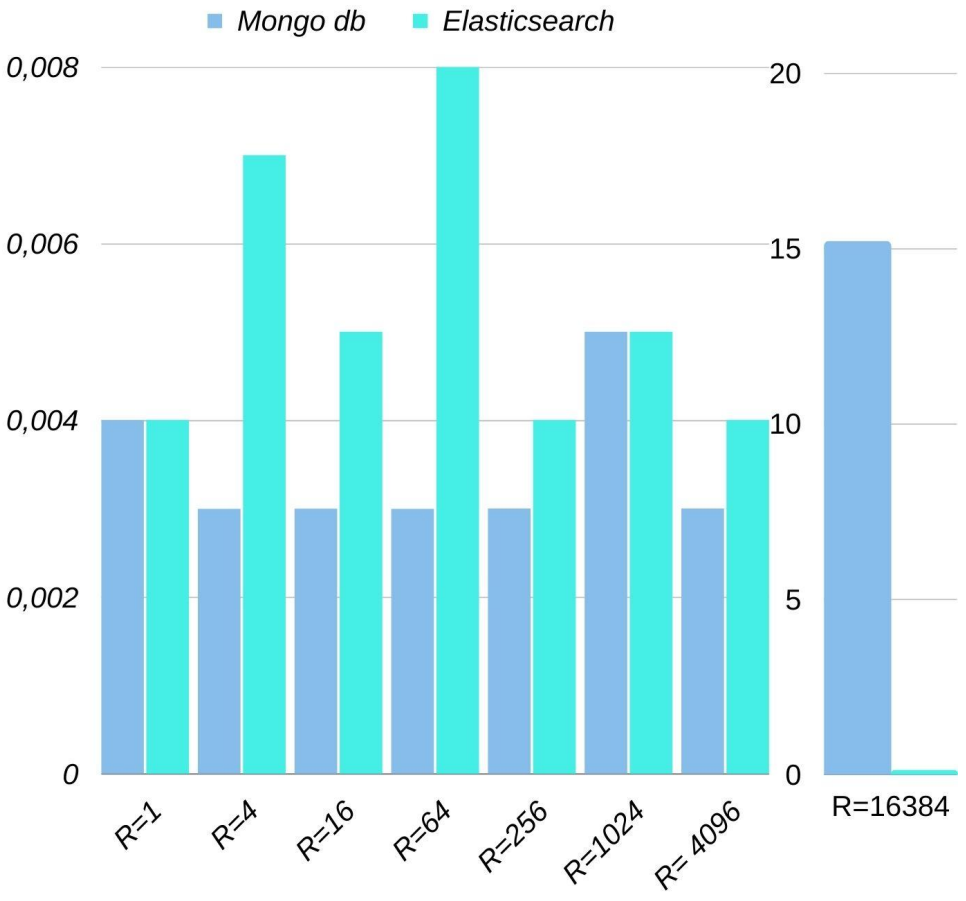
Comparación accidentsNearAPointInARadius

En esta segunda comparación la query involucrada consiste en la obtención de todos los accidentes ocurridos dentro del rango definido por un punto dado y su respectivo radio (en kilómetros). Se evaluó el tiempo que cada DB tardaba en resolver cada consulta variando el tamaño del radio a considerar en cada una.
A continuación se muestra un gráfico con los resultados de la comparación, en la parte de arriba hay una breve descripción de lo que se está viendo, debajo está el resultado dado en consola por nuestro algoritmo de comparación e inmediatamente debajo se puede ver un diagrama que contrasta el tamaño del radio usado con el tiempo que tardó en ejecutarse la consulta en MongoDB y ElasticSearch respectivamente.

TABLA 2

Promedio en segundos que tarda cada DB para la Consulta 2 a lo largo de 1 repeticion, variando el radio R a utilizar.

	R=1	R=4	R=16	R=64	R=256	R=1024	R=4096	R=16384
Mongo db	0,0040	0,0030	0,0030	0,0030	0,0030	0,0050	0,0030	15,2090
Elasticsearch	0,0040	0,0070	0,0050	0,0080	0,0040	0,0050	0,0040	0,1320



En esta oportunidad ambos empezaron de forma igualada, sin embargo para radios pequeños menores a los 4096 km MongoDB parece tener un rendimiento mejor que el de ElasticSearch a excepción de R=1 y R=1024 en los cuales su rendimiento está igualado. Sin embargo, al momento de pasar a radios significativamente grandes, sea R=16384 entonces ElasticSearch arrasa obteniendo el resultado en 0,13 segundos mientras que mongo DB tarda 15.2 segundos en obtener el mismo resultado. Pareciera que ElasticSearch a pesar de tener un rendimiento ligeramente peor que MongoDB para consultas pequeñas logra mantener tiempos relativamente constantes a medida que la consulta se vuelve mas exigente, lo cual termina poniéndolo por delante de MongoDB a partir de cierto umbral.

Conclusión general de la comparación

Pareciera que MongoDB, si bien comienza con un rendimiento muy bueno en ambas consultas, empeora su rendimiento conforme la consulta se va volviendo más y más exigente. Mientras que ElasticSearch a pesar de empezar un poco peor que mongo termina teniendo tiempos mas constantes, lo cual es especialmente apreciable cuando las consultas van creciendo de tamaño, ya que su tiempo la cantidad de tiempo extra que le toma una el incrementar la exigencia de una consulta es poca considerando el incremento de dificultad que la misma implica.

Por lo menos para las 2 querys de que nos interesaban el rendimiento de ElasticSearch parece ser mejor mas que nada por lo escable que llegan a ser sus consultas. Aunque si se conoce las limitaciones que tendrá la aplicación (no pasará de X meses o no interesan los accidentes que estén a mas de Y kilómetros) entonces podría ser que MongoDB sea una buena opción a considerar.