Universidad Diego Portales

Sistemas Distribuidos

Tarea 1

Daniel Ignacio Salas Pinto
Marcos David Valderrama Carrasco

Profesor: Nicolás Hidalgo

Ayudantes:
Joaquín Fernandez
Cristian Villancico
Nicolás Núñez

19 de septiembre de 2022

Índice

1. Problema y Solución

Se pide la creación de un Web Search Engine que tenga un sistema de caching e índice, el cual tenga la posibilidad de ser fácil escalable en un posible futuro. Para solucionar este problema se hará un sistema distribuido con diferentes módulos tal que se cumpla el siguiente esquema:

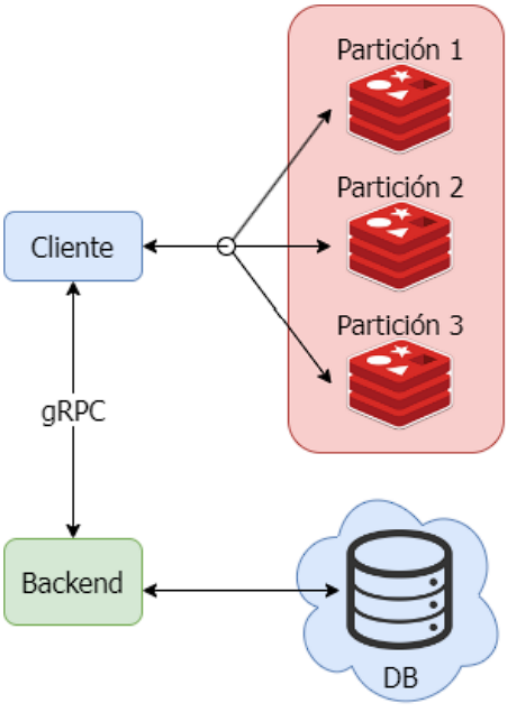

Figura 1: Esquema solución

Donde se separará cada parte del sistema en varios componentes utilizando Docker y estos se comunicarán vía gRPC, dichos componentes serán:

- Un cliente.
- Un servidor/backend.
- Una base de datos relacional (en este caso Postgres).
- Un sistema de caching para las consultas que se le harán a la base de datos (en este caso un clúster de Redis).
- Un crawler del dataset que vayamos a utilizar.

2. Módulos y Código utilizado

2.1. gRPC

Remote Procedural Controll (RPC) es un framework de código abierto de alta eficiencia y flexibilidad, ya que permite la comunicación entre diferentes servicios y datacenters soportando balanceos de carga, seguimiento de datos, verificación de integridad, autenticación, entre otros.

Para este caso, la conexión entre la base de datos y el backend se hizo mediante gRPC, utilizando la implementación de esta mostrada durante ayudantía (ver en Referencias). Se definió de la siguiente manera en nuestro docker-compose:

```
grpc_server:
  container_name: server-grpc
  build: ./gRPC
  depends_on:
    - postgres
  ports:
    - "8000:8000"
```

Listing 1: Docker Compose de gRPC

2.2. PostgreSQL

Es la base de datos relacional que elegimos utilizar por recomendación del documento instructivo de la tarea, se utilizó el modulo dockerizado generado por bitnami.

```
postgres:
  image: docker.io/bitnami/postgresql:14
  volumes:
    - ./Database/db/:/docker-entrypoint-initdb.d/test-init.sql
  environment:
    - POSTGRESQL_USERNAME=postgres
    - POSTGRESQL_PASSWORD=postgres
    - ALLOW_EMPTY_PASSWORD=yes
    - POSTGRESQL_DATABASE=dataset
  ports:
    - "5432:5432"
```

Listing 2: Docker Compose PostgreSQL

El archivo de la base de datos finalmente, queda ubicado en "/path/al/proyecto/Database/db/" con el nombre de test-init.sql, el cual se generará con el Crawler que se describirá en la siguiente sección.

2.3. Crawler

Un Crawler consiste en un software que recopila información de los sitios webs al ingresar en estos. Su uso esta orientado para reunir y crear índices, palabras claves, información y por sobre todo guardar la url de las paginas para motores de búsqueda de la web.

La implementación de Crawler utilizada en esta solución fue adaptada de un código demostrado en las ayudantías (link en referencias) hecho en python, dónde se cambiaron líneas las para coincidir con el formato del dataset descargado y los requerimientos de la tarea.
Al ejecutar el Crawler desarrollado se le ingresa como parámetro el nombre de archivo de entrada y salida, especialmente este ultimo debe ser 'test-init.sql' debido a los parametros asignados a la base de datos en el docker. Además se deben ingresar la cantidad máxima de datasets a generar. Posterior a su ejecución, el archivo de salida debe ser transferido a la ruta "/Database/db/" previo a la ejecución del docker, ya que este archivo será utilizado por la base de datos dando como resultado la creación de la tabla a utilizar por la base de datos, junto con la inserción de cada uno de los Datasets recolectados.

2.3.1. Para ejecutar este módulo en Unix se hizo lo siguiente:

```
cd \path\al\crawler

wget http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection/user-ct-test-collection-09.txt.gz

gunzip user-ct-test-collection-09.txt.gz

rm -f user-ct-test-collection-09.txt.gz

mv user-ct-test-collection-09.txt file.txt

cd ..

python3 crawler.py file.txt test-init.sql <nro líneas a sacar de file.txt>
#el archivo txt que bajamos sale como test-init.sql
```

Listing 3: Comandos Terminal de Unix

2.3.2. Este módulo en Windows se utilizó de la siguiente manera:

En el directorio del Crawler\files descargamos el dataset desde:
`http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection/`
Luego se descomprime el .txt con algún programa como 7zip
En Powershell se ejecutaron las siguientes líneas:

```
cd \path\al\crawler

py crawler.py file.txt test-init.sql <nro líneas a sacar de file.txt>
#el archivo txt que bajamos sale como test-init.sql
```

Listing 4: Comandos Terminal de Windows

Finalmente, y para ambos casos, movemos el archivo que nos queda de output ("test-init.sql") a el directorio del proyecto dónde debe quedar la base de datos de PostgreSQL, es decir, hacía:
'/path/al/proyecto/Database/db/'

2.4. Backend y API

Por último tenemos la parte del backend y la API para hacer consultas, donde el backend corresponde a un servidor corriendo a través del entorno de Node, el cual hace como intermediario entre las consultas realizadas al servicio, y las distintas instancias de cache y gRPC corriendo en el servicio.

```
const search = (req, res) => {

    const search = req.query.s;
    (async () => {

        let reply = await redis_client.get(search);
            if(reply){
                cache = JSON.parse(reply);

                // Requerimientos de Redis ( código )

                res.status(200).json(cache)
            }else{
            // Trae la respuesta desde la bdd si no lo encontró en el cache
                client.GetServerResponse({message:search}, (error,answer) =>{
                    if(error){
                        res.status(400).json(error);
                    }
                    else{
                        data = JSON.stringify(answer)
                        if (data['product']!==null){
                            // Guarda la respuesta en Redis para otra posible
    peticion
                            redis_client.set(search,data)
                            res.status(200).json(answer);
                        }

                    }
                });
            }
    })();
};
```

Listing 5: backend (index.controller.js)

Para acceder a este servicio se debe hacer una petición de tipo get HTTP a la dirección que corre el servicio Node junto al puerto pre-establecido 3000, seguido de esto se escriben los parametros correspondientes siguiendo el siguiente orden:

```
http://localhost:3000/search?s=InputConsulta
```

Listing 6: Consulta HTTP

3. Explicación del caché

3.1. Redis

Redis es un deposito en memoria de estructuras de datos usado con el fin de creación de base de datos, uso de memoria cache, uso de broker de mensajería, o bien como motor de streaming. En nuestro caso su función es usarlo como memoria cache en base a las consultas que se realizan al servicio.
El objetivo de esta instancia es que reduzca el tiempo de respuesta al guardar las ultimas consultas realizadas al servicio dentro de una instancia configurada para servir en memoria cache. De esta manera, en caso de querer escalar, se puedan generar mas instancias de Redis como nodos funcionales a memoria cache.

Se desea que la implementación de este modulo permita la escalabilidad a través de la posible ejecución de mas de una instancia de este tipo. En este caso se decidió utilizar el clúster de Redis en modo replicación, por lo que hay una imagen de Redis como Master y 2 como sus slaves, estos slaves vendrían a ser copias en tiempo real del Master que contienen exactamente la misma información y memoria pero no son la misma imagen.

Para lograr esto en el docker-compose se desplegaron 3 imágenes de redis de la siguiente manera:

```
1  redis:
2    container_name: redis_1_master
3    image: bitnami/redis:7.0.4
4    restart: always
5    ports:
6      - "7000:6379"
7    environment:
8      - ALLOW_EMPTY_PASSWORD=yes
9      - REDIS_REPLICATION_MODE=master
10   command: /opt/bitnami/scripts/redis/run.sh --maxmemory 2mb --maxmemory-policy
     allkeys-lfu
11
12 redis2:
13   image: bitnami/redis:7.0.4
14   container_name: redis_2
15   ports:
16     - "7001:6379"
17   environment:
18     - REDIS_REPLICATION_MODE=slave
19     - REDIS_MASTER_HOST=redis
20     - ALLOW_EMPTY_PASSWORD=yes
21     - REDIS_MASTER_PORT_NUMBER=6379
22   restart: always
23
24 redis3:
25   image: bitnami/redis:7.0.4
26   container_name: redis_3
27   ports:
28     - "7002:6379"
29   environment:
```

```
30        - REDIS_REPLICATION_MODE=slave
31        - REDIS_MASTER_HOST=redis
32        - ALLOW_EMPTY_PASSWORD=yes
33        - REDIS_MASTER_PORT_NUMBER=6379
34      restart: always
```

Listing 7: Docker Compose de Redis

Cómo parámetros en el master se utilizan:

- –maxmemory 2MB: se le dan 2MB de tamaño a la imagen Master de Redis (y por lo tanto a sus copias).
- –maxmemory-policy allkeys-lfu: esta política dice que se mantendrán las keys utilizadas más frecuentemente y se borrarán las menos frecuentes apenas se llegue al tope de memoria.

4. Resultados y análisis del caché

Probaremos el cache mediante la medición del tiempo de respuesta a consultas en ms
Primero haremos 3 consultas con los términos “ESPN”, “music”y “garden”.

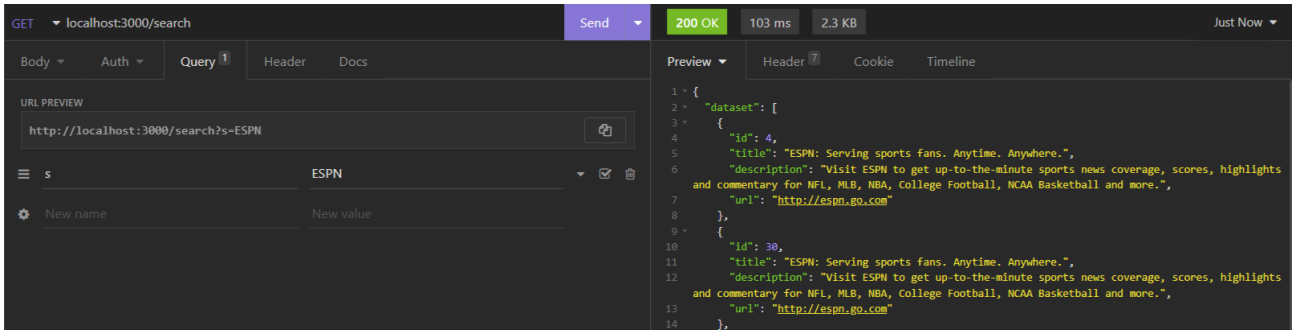

Figura 2: Consulta por ESPN

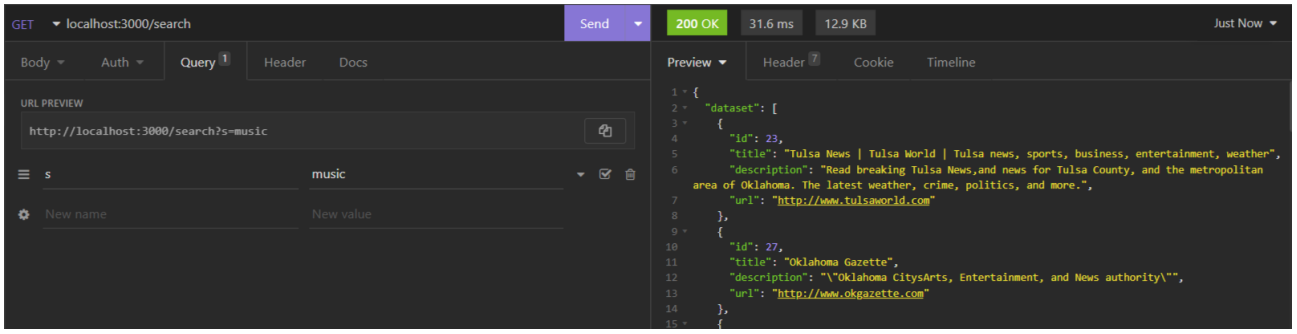

Figura 3: Consulta por music

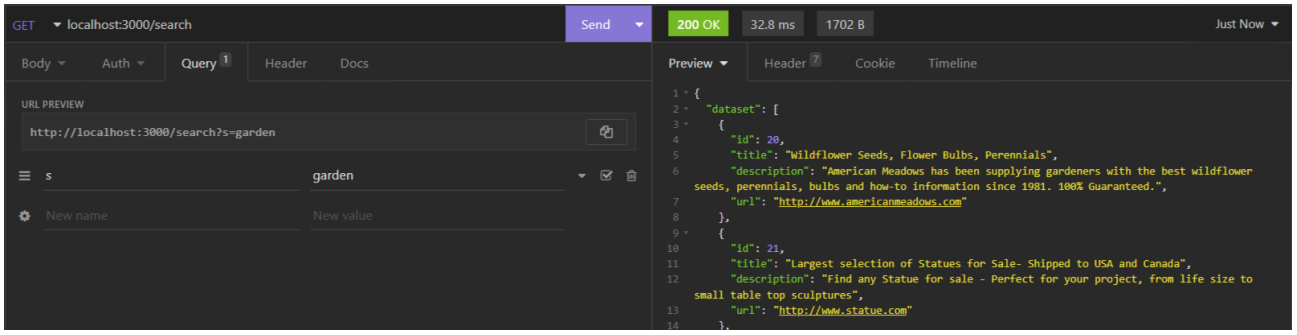

Figura 4: Consulta por garden

Luego volvemos a realizar las mismas 3 consultas para ver si algo ha cambiado:

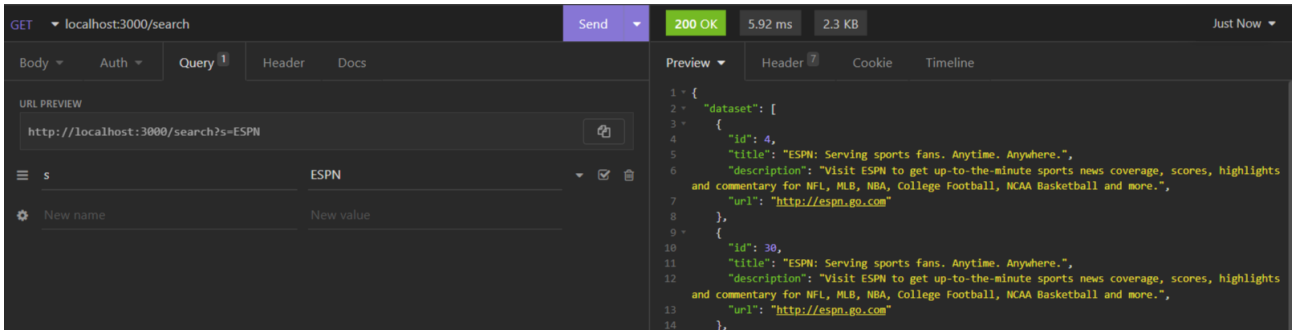

Figura 5: 2da consulta por ESPN

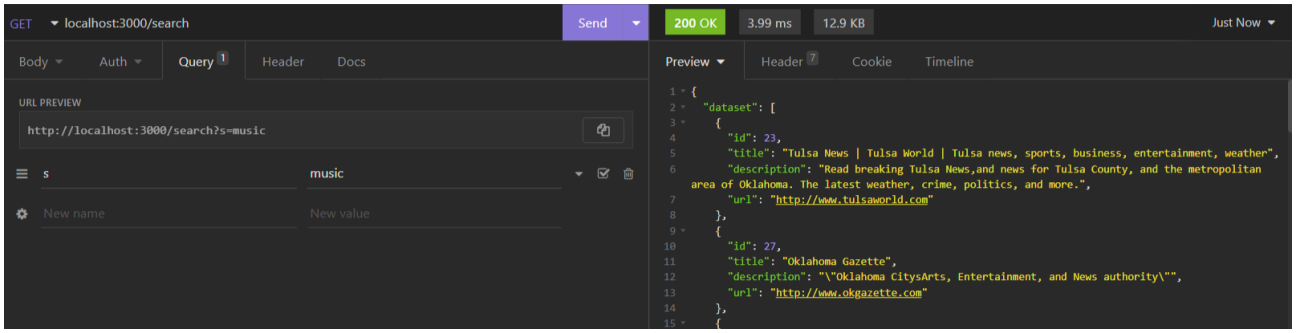

Figura 6: 2da consulta por music

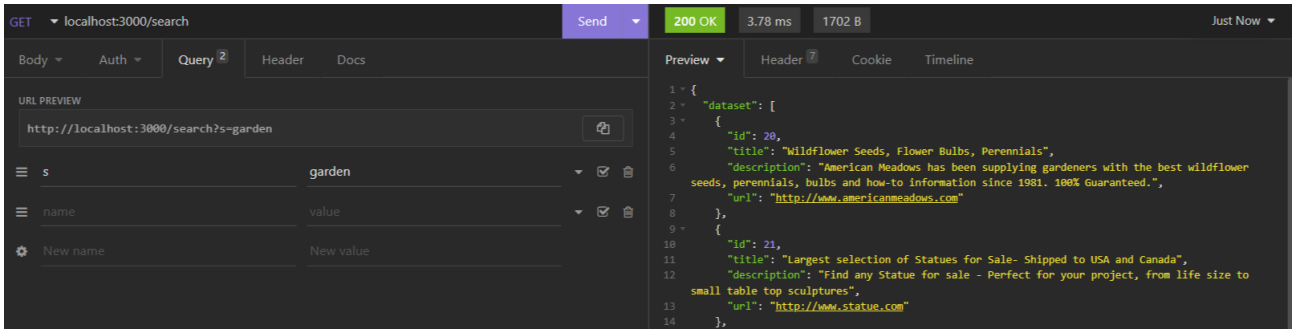

Figura 7: 2da consulta por garden

Query:	ESPN	music	garden
Primera vez:	103 ms	31,6 ms	32,8 ms
Segunda vez:	5,92 ms	3,99 ms	3,78 ms

Cuadro 1: Diferencia en el tiempo de respuesta

Como podemos observar han cambiado los tiempos de respuesta al volver a realizar las querys, mostrando como mejora la performance (en este caso siendo medida por la demora en recibir las respuestas) al tener un caché activo que guarda las consultas más frecuentes o repetidas, permitiendo no tener que hacer consultas directas a la base de datos cada vez que se necesite saber algo.

5. Conclusión

En esta implementación tener los módulos separados entre sí, nos permite aislar posibles errores o falencias que podrían darse si tuviéramos todo centralizado, lo que complicaría la búsqueda de soluciones a posibles errores, para esto, el uso de docker ha sido un gran apoyo puesto que nos permite simular un sistema de esta índole a pesar de no tener físicamente una gran diversidad de hardware.

Por otra parte, en el contexto del caché un buscador, tener los resultados más frecuentes a mano para no tener que navegar una base de datos es probablemente la mejor opción debido a la gran cantidad de entradas que podría tener un buscador, lo que aliviaría la carga tanto del servidor como de la base de datos.

6. Enlaces

6.1. Video y Github

Enlaces del video explicativo y de documentacion de github:

- `https://www.youtube.com/watch?v=8L0OK7jyPCs`
- `https://github.com/MrC0u/SD_Tarea_1`

6.2. Referencias

Ayudantías desde las cuales se obtuvieron los códigos de las conexiones para utilizar gRPC, caching con Redis, el Crawler y la BDD:

- `https://github.com/Oscurt/sd_202201`
- `https://github.com/Joacker/Ayu-SD-2022-2`

Documentación de Redis:

- `https://raw.githubusercontent.com/redis/redis/7.0/redis.conf`