Universidad Nacional Autónoma de México
Facultad de Ingeniería

Proyecto Final

Bases de Datos No Estructuradas

Grupo 0600

Profesor:

Ing. Jorge Alberto Rodríguez Campos

Alumno:

Núñez Quintana Luis Axel

Ciudad de México — 07 de junio, 2024

Índice

Proyecto Final

Núñez Quintana Luis Axel

07 de junio, 2024

1. Planteamiento

El presente proyecto tiene como objetivo desarrollar una aplicación web que haga uso de una base de datos políglota, permitiendo a los usuarios registrarse, iniciar sesión, crear y compartir contenido de manera eficiente. La aplicación está diseñada para manejar grandes volúmenes de datos, ofreciendo un rendimiento y escalabilidad.

Para lograr estos objetivos, se emplea una combinación de bases de datos relacionales y no relacionales. La base de datos relacional se utilizará para manejar datos estructurados y garantizar la integridad de los datos. Se empleará PostgreSQL al tratar con datos de usuarios y credenciales de inicio de sesión. Por otro lado, las bases de datos no relacionales se encargarán de gestionar datos no estructurados y semi-estructurados, permitiendo una mayor flexibilidad y velocidad en la recuperación y almacenamiento de información. Principalmente se encargarán de almacenar los datos de las sesiones de la página web, el contenido proveniente de la misma y también las relaciones que puedan surgir de dicho contenido.

En esta etapa inicial del proyecto, el enfoque se centrará exclusivamente en la capa de almacenamiento de datos y en la primera etapa del backend, utilizando Node.js y Express.js. Se busca implementar una API RESTful que gestione eficazmente las operaciones de registro, autenticación y gestión de contenido de los usuarios.

El desarrollo de esta aplicación no solo busca satisfacer las necesidades actuales de los usuarios, sino también prepararse para un crecimiento futuro. La elección de una base de datos políglota, junto con un backend eficaz y adaptable, asegura que la aplicación pueda escalar vertical y horizontalmente. Esto permite manejar incrementos en la cantidad de usuarios y volumen de datos sin comprometer el rendimiento o la experiencia del usuario final.

2. Modelado

La primera etapa del proyecto se enfocó en el modelado de la capa de almacenamiento, estableciendo la arquitectura sobre la cual se desarrollará la aplicación web. En esta fase inicial, se definieron las responsabilidades específicas de cada tipo de base de datos.

Se crearon esquemas detallados para cada base de datos, asegurando que tanto los datos estructurados como los no estructurados sean gestionados de manera eficiente y coherente con los objetivos del proyecto.

Esta fase no solo estableció la arquitectura sobre la cual se desarrollará la aplicación web, sino que también sentó las bases sólidas para una implementación exitosa, asegurando que cada base de datos cumpla con su papel específico.

2.1. PostgreSQL

En primera instancia, se delegó el manejo de datos de las cuentas de los usuarios a la base de datos relacional PostgreSQL. Esta base de datos será la encargada de almacenar las credenciales de los usuarios y la información de sus cuentas, incluyendo roles y permisos dentro de la aplicación. La elección de PostgreSQL se fundamenta en su capacidad para gestionar de manera segura campos sensibles, como las contraseñas de los usuarios.

A continuación, se muestra el diagrama relacional de la base de datos.

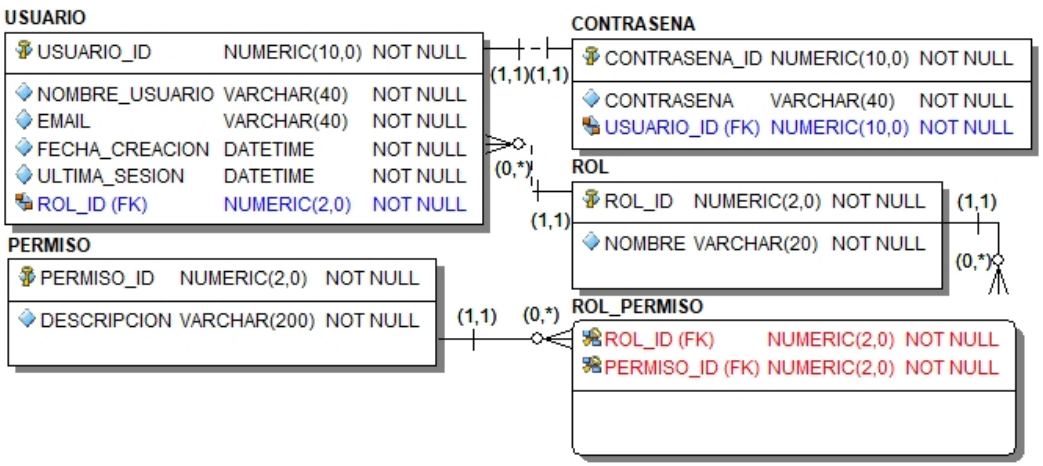

Figura 1: Modelo Relacional

2.2. Riak KV

Para gestionar los datos de la sesión de usuarios se decidió emplear Riak KV. Cada sesión de usuario se almacena como un objeto JSON asociado a una clave única, lo que permite un acceso rápido y flexible a datos como las preferencias de usuario, configuraciones temporales y registros de actividad.

A continuación, se describe cómo se puede estructurar y manejar el JSON proporcionado usando Riak KV:

```
"sesion_id" : {
    "usuario_id" : <usuario_id>,
    "inicio" : <fecha_inicio>,
    "ultimo_acceso" : <fecha_ultimo_acceso>,
    "preferencias" : {
        "tema" : <oscuro || claro>,
        "idioma" : <idioma>,
        "vista" : <expandida || compacta>
    },
    "configuraciones_temporales" : {
        "diagrama_abierto" : {
            "diagrama_id" : <diagrama_id>,
            "estado" : <estado>,
            "ultima_actualización" : <fecha>
      },
      "busqueda_reciente" : {
        "query" : <query>,
        "filtros" : {
            "tipo" : <'ER' || 'R' || 'F'>,
            "fecha" : <fecha>
        }
      }
    }
}
```

2.3. MongoDB

Continuando, se eligió utilizar MongoDB para gestionar los documentos resultantes de la aplicación de manera eficiente. Cada diagrama se almacena como un documento BSON en una colección, permitiendo una estructura flexible y anidada que incluye entidades, relaciones, y sus atributos correspondientes.

A continuación, se muestra un ejemplo del documento de un diagrama:

```
{
  "_id": <diagrama_id>,
  "usuario_id": <usuario_id>,
  "nombre_diagrama": <Titulo diagrama>,
  "tipo_diagrama": <ER || R || F>,
  "contenido": {
    "entidades": [
      {
        "nombre": <nombre_entidad>,
        "atributos": [<atributo_1>, <atributo_2>, ... , <atributo_n>]
      },
    ],
    "relaciones": [
      {
        "nombre": <nombre_relacion>,
        "entidades": [<entidad_1>, <entidad_2>],
        "cardinalidad": <N:M>
      }
    ]
  },
  "fecha_creacion": <fecha_creacion>,
  "fecha_modificacion": <fecha_modificacion>
}
```

2.4. Neo4J

Finalmente, se emplea Neo4j para la gestión de relaciones entre usuarios y sus diagramas. Las entidades como usuario, diagrama y etiqueta se representan como nodos, mientras que las relaciones entre ellos se modelan como aristas. Esto incluye relaciones como sigue, crea, comenta, comparte, favorito y tiene. Este modelado permite una representación natural y eficiente de la interconexión entre usuarios y sus actividades en la aplicación.

A continuación, se ilustra su uso mediante un diagrama.

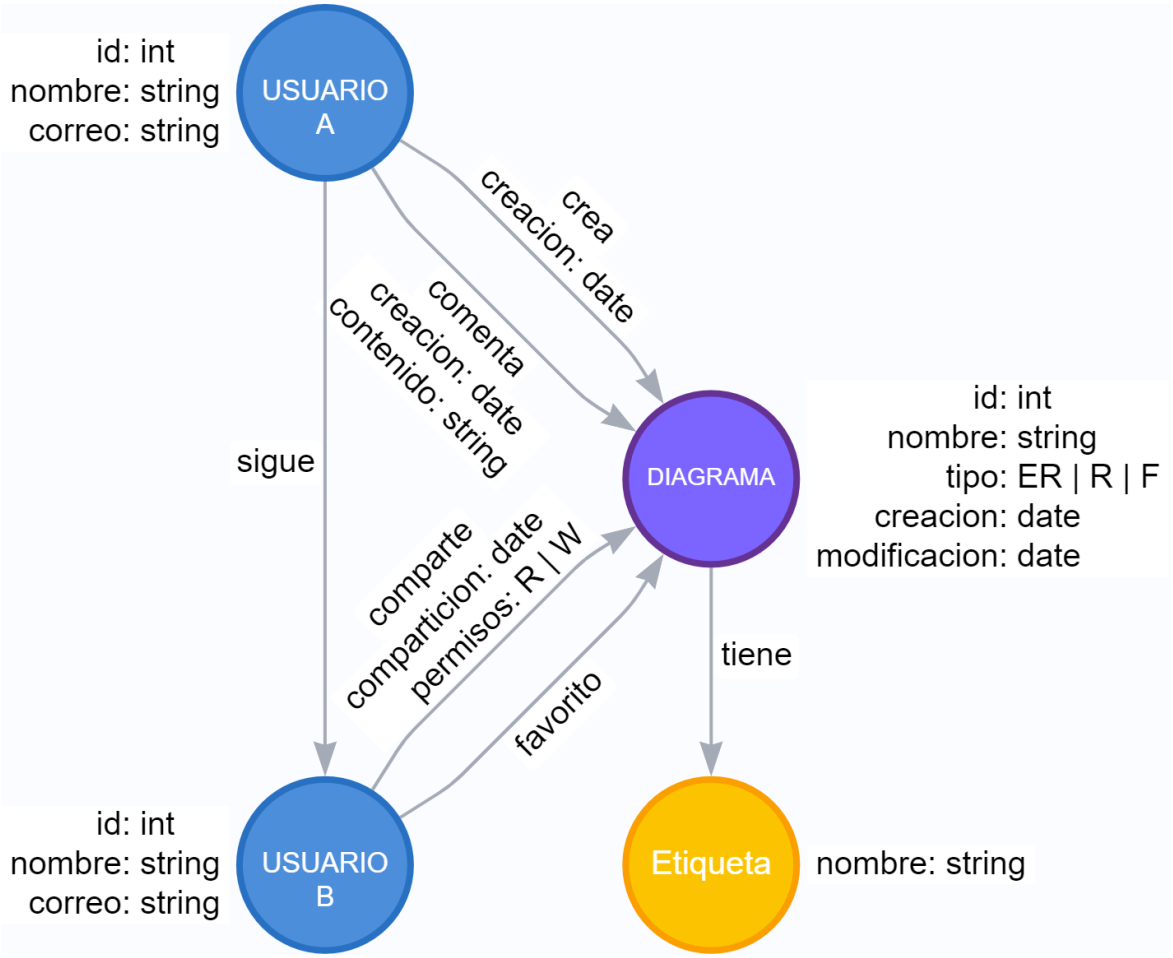

Figura 2: Diagrama de Neo4J

3. Capa de almacenamiento

Para implementar las bases de datos se crearon múltiples entornos utilizando contenedores Docker. Esta estrategia no solo facilitó la gestión y el despliegue de los diferentes componentes de la solución, sino que también proporcionó una arquitectura modular que promueve la flexibilidad y la escalabilidad del sistema.

Al emplear contenedores Docker, cada base de datos se encapsula, lo que garantiza una mayor consistencia y reproducibilidad en todos los entornos, desde el desarrollo hasta la producción. Además, este esquema permite la posibilidad de integrar soluciones como Kubernetes para favorecer la escalabilidad y disponibilidad.

3.1. PostgreSQL

La fase inicial del proceso de implementación se centró en la creación y configuración de la base de datos PostgreSQL.

3.1.1. Entorno

Se describió el entorno empleando un archivo dockerfile. Mediante este archivo se crea una imagen con PostgreSQL configurada para la solución.

```
# Utiliza la imagen oficial de PostgreSQL como base
FROM postgres:latest

# Variables de entorno para la base de datos y credenciales
ENV POSTGRES_DB=diagram_db \
    POSTGRES_USER=admin \
    POSTGRES_PASSWORD=admin123 \
    TZ=America/Mexico_City

# Scripts de inicio
COPY init-scripts/* /docker-entrypoint-initdb.d/

# Puerto de PostgreSQL
EXPOSE 5432
```

Mediante el siguiente comando se crea la imagen:

```
$ docker build -t luisaxel/postgresql:latest .
```

También se decidió publicar la imagen en Docker Hub. No fue necesario etiquetarla debido a que fue correctamente nombrada mediante el comando *docker build*.

```
$ docker push luisaxel/postgresql:latest
```

Una vez se cuenta con la imagen, es posible crear el contenedor mediante la siguiente instrucción:

```
$ docker run -d --name postgresql -p 5432:5432
↪  luisaxel/postgresql:latest
```

Para probar el funcionamiento del contenedor se accedió mediante el siguiente comando:

```
$ docker exec -it postgresql psql -U admin diagram_db
```

Dentro de la base de datos se observó la correcta definición de las entidades y la carga inicial.

```
diagram_db=# SELECT table_name FROM information_schema.tables WHERE table_type = 'BASE TABLE' and table_schema = 'public';
 table_name
-------------
 usuario
 contrasena
 rol
 rol_permiso
 permiso
(5 rows)

diagram_db=# select * from usuario;
 usuario_id | nombre_usuario |       email        |       fecha_creacion       |       ultima_sesion        | rol_id
------------+----------------+--------------------+----------------------------+----------------------------+--------
          1 | usuario1       | usuario1@gmail.com | 2024-06-01 03:36:32.823844 | 2024-06-01 03:36:32.823844 |      2
          2 | usuario2       | usuario2@gmail.com | 2024-06-01 03:36:32.823844 | 2024-06-01 03:36:32.823844 |      3
          3 | usuario3       | usuario3@gmail.com | 2024-06-01 03:36:32.823844 | 2024-06-01 03:36:32.823844 |      4
(3 rows)
```

Figura 3: Consultas en PostgreSQL

3.1.2. Creación de entidades

Siguiendo el esquema descrito en 2.1, se tiene lo siguiente:

```
-- @autor: Luis Axel Núñez Quintana
-- @Fecha creación: 01-06-2024
-- @Descripción: Script de operaciones ddl inicial para diagram_db

--
-- TABLE: CONTRASENA
```

```
--
CREATE TABLE CONTRASENA(
    CONTRASENA_ID    numeric(10, 0)  NOT NULL,
    CONTRASENA_HASH  varchar(100)    NOT NULL,
    USUARIO_ID       numeric(10, 0)  NOT NULL,
    CONSTRAINT CONTRASENA_PK PRIMARY KEY (CONTRASENA_ID)
);

--
-- TABLE: PERMISO
--
CREATE TABLE PERMISO(
    PERMISO_ID   numeric(2, 0)    NOT NULL,
    DESCRIPCION  varchar(200)     NOT NULL,
    CONSTRAINT PERMISO_PK PRIMARY KEY (PERMISO_ID)
);

--
-- TABLE: ROL
--
CREATE TABLE ROL(
    ROL_ID  numeric(2, 0)    NOT NULL,
    NOMBRE  varchar(20)      NOT NULL,
    CONSTRAINT ROL_PK PRIMARY KEY (ROL_ID)
);

--
-- TABLE: ROL_PERMISO
--
CREATE TABLE ROL_PERMISO(
    ROL_ID      numeric(2, 0)    NOT NULL,
    PERMISO_ID  numeric(2, 0)    NOT NULL,
    CONSTRAINT ROL_PERMISO_PK PRIMARY KEY (ROL_ID, PERMISO_ID)
);

--
```

```
-- TABLE: USUARIO
--
CREATE TABLE USUARIO(
    USUARIO_ID      numeric(10, 0)    NOT NULL,
    NOMBRE_USUARIO  varchar(40)       NOT NULL,
    EMAIL           varchar(40)       NOT NULL,
    FECHA_CREACION  timestamp         NOT NULL,
    ULTIMA_SESION   timestamp         NOT NULL,
    ROL_ID      numeric(2, 0)     NOT NULL,
    CONSTRAINT USUARIO_PK PRIMARY KEY (USUARIO_ID)
);

--
-- FK INDEXES
--
CREATE INDEX CONTRASENA_USUARIO_ID_IX ON CONTRASENA(USUARIO_ID);

CREATE INDEX ROL_PERMISO_ROL_ID_IX ON ROL_PERMISO(ROL_ID);

CREATE INDEX ROL_PERMISO_PERMISO_ID_IX ON ROL_PERMISO(PERMISO_ID);

CREATE INDEX USUARIO_ROL_ID_IX ON USUARIO(ROL_ID);

--
-- FK CONSTRAINTS
--

--
-- TABLE: CONTRASENA
--
ALTER TABLE CONTRASENA ADD CONSTRAINT CONTRASENA_USUARIO_USUARIO_ID_FK
    FOREIGN KEY (USUARIO_ID)
    REFERENCES USUARIO(USUARIO_ID);

--
-- TABLE: ROL_PERMISO
```

```
--
ALTER TABLE ROL_PERMISO ADD CONSTRAINT ROL_PERMISO_ROL_ROL_ID_FK
    FOREIGN KEY (ROL_ID)
    REFERENCES ROL(ROL_ID);

ALTER TABLE ROL_PERMISO ADD CONSTRAINT
↪   ROL_PERMISO_PERMISO_PERMISO_ID_FK
    FOREIGN KEY (PERMISO_ID)
    REFERENCES PERMISO(PERMISO_ID);

--
-- TABLE: USUARIO
--
ALTER TABLE USUARIO ADD CONSTRAINT USUARIO_ROL_ROL_ID_FK
    FOREIGN KEY (ROL_ID)
    REFERENCES ROL(ROL_ID);
```

3.1.3. Carga inicial

Terminando con PostgreSQL en la capa de almacenamiento, se decidió crear una pequeña carga inicial:

```
-- @autor: Luis Axel Núñez Quintana
-- @Fecha creación: 01-06-2024
-- @Descripción: Script de carga inicial para diagram_db

-- PERMISO
INSERT INTO PERMISO (PERMISO_ID, DESCRIPCION)
VALUES (1, 'Permiso de conversion a relacional'),
       (2, 'Permiso de conversion a entidad relacion'),
       (3, 'Permiso de conversion a modelo fisico');

-- ROL
INSERT INTO ROL (ROL_ID, NOMBRE)
VALUES (1, 'Invitado'),
       (2, 'Regular'),
       (3, 'Premium'),
```

```
       (4, 'Administrador');

-- ROL_PERMISO
INSERT INTO ROL_PERMISO (ROL_ID, PERMISO_ID)
VALUES (2, 1),
       (2, 2),
       (3, 1),
       (3, 2),
       (3, 3),
       (4, 1),
       (4, 2),
       (4, 3);

-- USUARIO
INSERT INTO USUARIO (USUARIO_ID, NOMBRE_USUARIO, EMAIL, FECHA_CREACION,
↪  ULTIMA_SESION, ROL_ID)
VALUES (1, 'usuario1', 'usuario1@gmail.com', CURRENT_TIMESTAMP,
↪  CURRENT_TIMESTAMP, 2),
       (2, 'usuario2', 'usuario2@gmail.com', CURRENT_TIMESTAMP,
       ↪  CURRENT_TIMESTAMP, 3),
       (3, 'usuario3', 'usuario3@gmail.com', CURRENT_TIMESTAMP,
       ↪  CURRENT_TIMESTAMP, 4);

-- CONTRASEÑA
INSERT INTO CONTRASENA (CONTRASENA_ID, CONTRASENA_HASH, USUARIO_ID)
VALUES (1, 'contrasena_123', 1),
       (2, 'contrasena_456', 2),
       (3, 'contrasena_789', 3);

COMMIT;
```

3.2. Riak KV

3.2.1. Entorno

3.2.2. Carga inicial

3.3. MongoDB

3.3.1. Entorno

3.3.2. Creación de esquema

3.3.3. Carga inicial

3.4. Neo4J

3.4.1. Entorno

3.4.2. Creación de entidades inicial

3.4.3. Creación de relaciones inicial

4. Backend

4.1. Comunicación

4.1.1. PostgreSQL

4.1.2. Riak KV

4.1.3. MongoDB

4.1.4. Neo4J

4.2. Métodos CRUD

4.2.1. PostgreSQL

4.2.2. Riak KV

4.2.3. MongoDB

4.2.4. Neo4J

4.3. Demo de uso

5. Resultados

6. Conclusiones

Referencias

[1] Basho Technologies. *Riak KV Documentation*. Recuperado: 2024-05-29. 2024. URL: https://docs.riak.com/riak/kv/latest/.

[2] MongoDB, Inc. *MongoDB Documentation*. Recuperado: 2024-05-29. 2024. URL: https://docs.mongodb.com/.

[3] Neo4j, Inc. *Neo4j Documentation*. Recuperado: 2024-05-29. 2024. URL: https://neo4j.com/docs/.

[4] Oracle Corporation. *Oracle Database Documentation*. Recuperado: 2024-05-29. 2024. URL: https://docs.oracle.com/en/database/.