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.

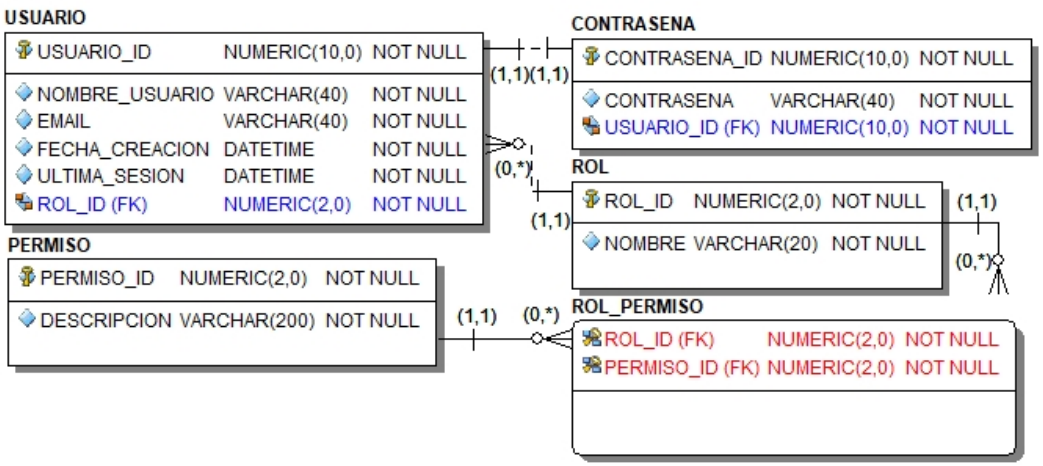

Figura 1: Modelo Relacional

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.

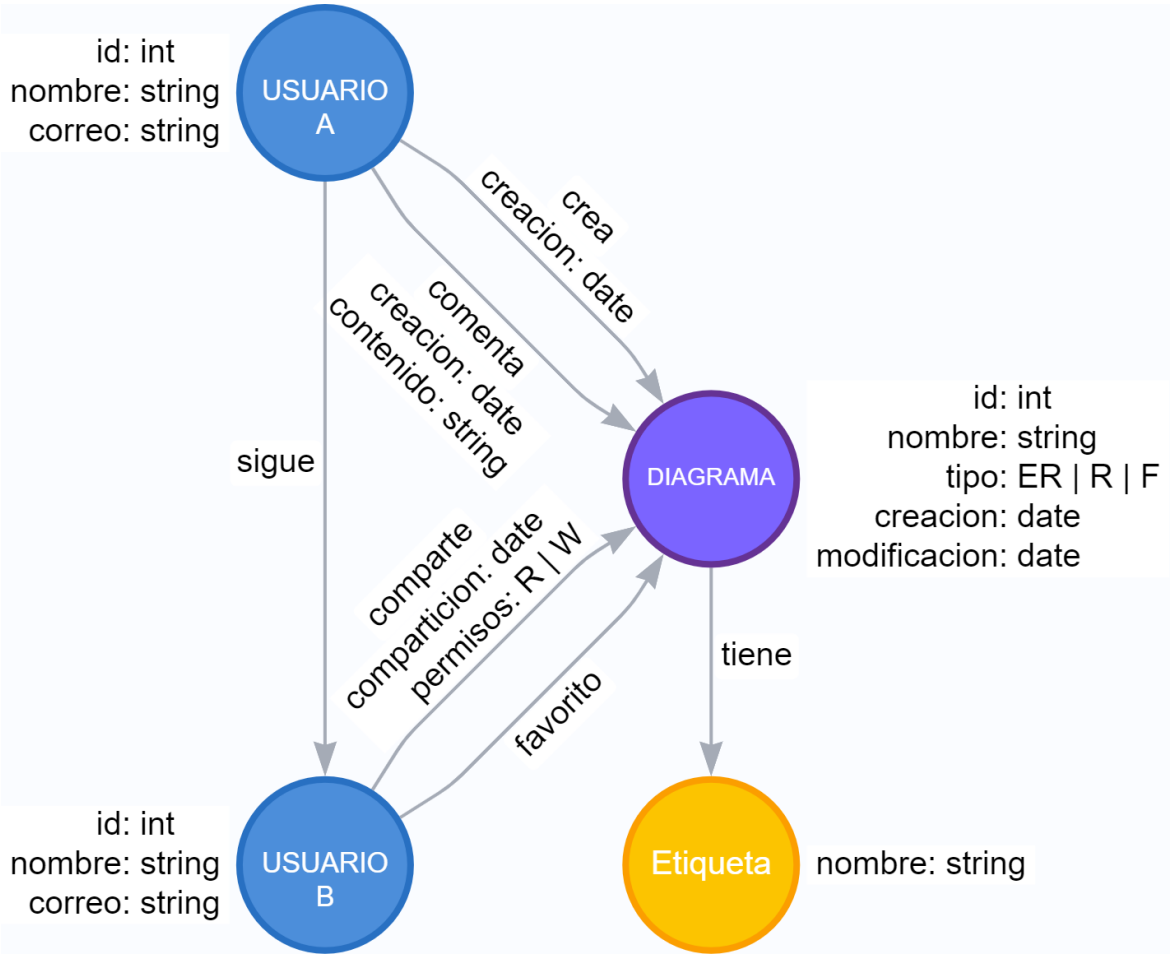

Figura 2: Diagrama de Neo4J

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

Se decidió implementar Riak KV 2.2.3 al ser una versión estable en comparación con Riak KV 3+.

3.2.1. Entorno

Se empleó la imagen oficial de Riak KV para la elaboración del cluster de Riak. A continuación, se muestra el archivo *yml* que será utilizado para la creación del cluster.

```yaml
services:
  coordinator:
    image: basho/riak-kv
    ports:
      - "8087"
      - "8098"
    environment:
      - CLUSTER_NAME=riakkv
    labels:
      - "com.basho.riak.cluster.name=riak-kv"
    volumes:
      - schemas:/etc/riak/schemas
      - coordinator_data:/var/lib/riak
      - coordinator_logs:/var/log/riak
    network_mode: bridge
  member:
    image: basho/riak-kv
    ports:
      - "8087"
      - "8098"
    labels:
      - "com.basho.riak.cluster.name=riak-kv"
    links:
      - coordinator
    network_mode: bridge
    depends_on:
      - coordinator
```

```
    environment:
      - CLUSTER_NAME=riakkv
      - COORDINATOR_NODE=coordinator

volumes:
  schemas: {}
  coordinator_data: {}
  coordinator_logs: {}
```

Una vez se cuenta con la descripción del cluster de riak es posible crearlo mediante la instrucción:

```
$ docker compose scale coordinator=1 member=3
```

A continuación, se observa el estado del cluster mediante la instrucción

```
$ docker compose exec coordinator riak-admin cluster status
```

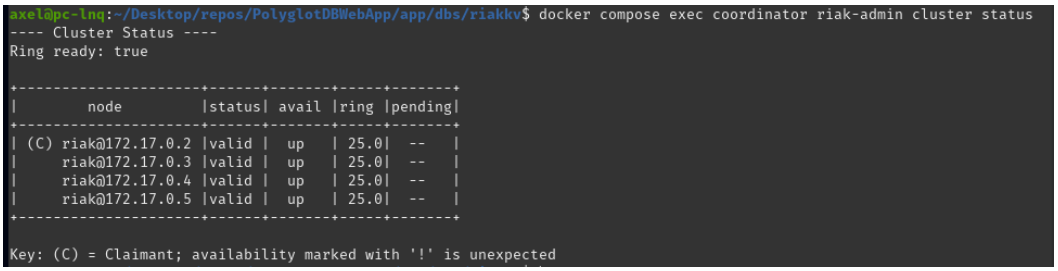

Figura 4: Cluster de Riak KV

y se verifica la existencia de registros en dos nodos mediante la API de HTTP.

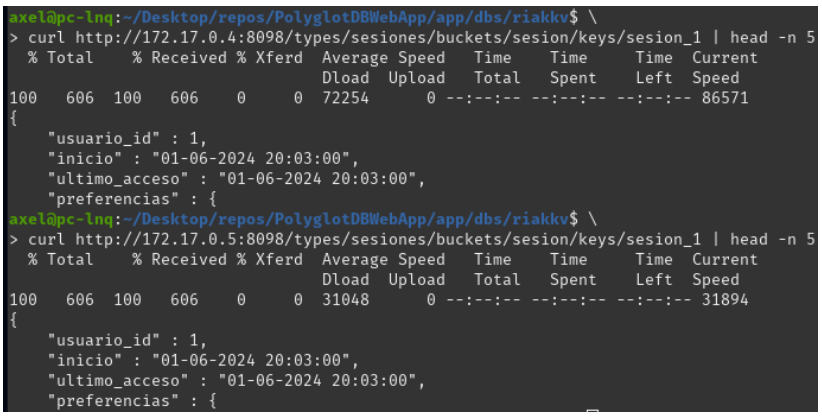

Figura 5: Consultas HTTP en Riak KV

3.2.2. Carga inicial

Para la carga inicial se elaboró un script en bash mediante el cual se define la ip y puerto para realizar conexiones al cluster. Luego se crea y activa el bucket-type en caso de no existir y se insertan 3 objetos en caso de no existir.

```
#!/bin/bash

# Step 1: Obten la dirección ip del nodo coordinador
COORDINATOR_IP=$(docker inspect -f '{{.NetworkSettings.IPAddress}}'
 ↪  riakkv-coordinator-1)
echo "IP: ${COORDINATOR_IP}"

# Step 2: Guarda el valor del puerto de riak en variable
PORT=8098
echo "PORT: ${PORT}"

# Step 3: Crea JSON para el bucket-type del proyecto
BUCKET_TYPE="sesiones"
BUCKET_TYPE_JSON='{"props":{"n_val":4}}'

# Step 4: Crea y activa el bucket-type si no existe
echo "Creando y activando bucket-type ${BUCKET_TYPE}"
RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" "http://${COORDINATO
 ↪  R_IP}:${PORT}/types/${BUCKET_TYPE}/buckets?buckets=true")
if [ "${RESPONSE}" -ne 200 ]; then
  echo "Creando bucket-type ${BUCKET_TYPE}..."
  docker compose exec coordinator riak-admin bucket-type create
   ↪  ${BUCKET_TYPE} ${BUCKET_TYPE_JSON}
  echo "Activando bucket-type ${BUCKET_TYPE}..."
  docker compose exec coordinator riak-admin bucket-type activate
   ↪  ${BUCKET_TYPE}
else
  echo "El bucket-type ${BUCKET_TYPE} ya existe y se encuentra
   ↪  activado"
fi
```

```
# Step 5: Verifica bucket-type
echo "Verificando bucket-type ${BUCKET_TYPE}..."
RESPONSE=$(curl -s -o /dev/null -w "%{http_code}" "http://${COORDINATO
↪  R_IP}:${PORT}/types/${BUCKET_TYPE}/buckets?buckets=true")
echo "Código de respuesta: ${RESPONSE}"

# Step 6: Crea 3 JSONs para inserción de objetos
OBJECT1=$(cat <<EOF
{
    "usuario_id" : 1,
    "inicio" : "01-06-2024 20:03:00",
    "ultimo_acceso" : "01-06-2024 20:03:00",
    "preferencias" : {
        "tema" : "oscuro",
        "idioma" : "spa",
        "vista" : "compacta"
    },
    "configuraciones_temporales" : {
        "diagrama_abierto" : {
            "diagrama_id" : 1,
            "estado" : "editando",
            "ultima_actualización" : "01-06-2024 20:03:00"
      },
      "busqueda_reciente" : {
        "query" : "usuario_id = 1",
        "filtros" : {
            "tipo" : "ER",
            "fecha" : "01-06-2024 20:03:00"
        }
      }
    }
}
EOF
)
# ... OBJ 2, 3
```

```
# Step 7: Insertando objetos
BUCKET="sesion"

# Funcion para verificar e insertar objeto
insert_and_verify() {
  local key=$1
  local value=$2
  RESPONSE=$(curl -s -o /dev/null -w "%{http_code}"
  ↪  "http://${COORDINATOR_IP}:${PORT}/types/${BUCKET_TYPE}/buckets/$
  ↪  {BUCKET}/keys/$key")
  if [ "${RESPONSE}" -ne 200 ]; then
    echo "Insertando objeto con key: ${key}..."
    curl -XPUT "http://${COORDINATOR_IP}:${PORT}/types/${BUCKET_TYPE}/
    ↪  buckets/${BUCKET}/keys/${key}" -H "Content-Type:
    ↪  application/json" -d "${value}"
  else
    echo "Objeto con key: ${key}, ya existe."
  fi
  echo "Vericando objecto con key: ${key}..."
  curl -XGET "http://${COORDINATOR_IP}:${PORT}/types/${BUCKET_TYPE}/bu
  ↪  ckets/${BUCKET}/keys/${key}"
  echo
}

insert_and_verify "sesion_1" "${OBJECT1}"
insert_and_verify "sesion_2" "${OBJECT2}"
insert_and_verify "sesion_3" "${OBJECT3}"

# echo para asegurar que el prompt the bash se encuentre en una nueva
↪  linea
echo
```

Debido a que se emplea un volumen en el coordinador, los datos no se pierden cuando se elimina el cluster.

3.3. MongoDB

Para la base de datos MongoDB se empleó la versión 6.

3.3.1. Entorno

El cluster utiliza la imagen oficial de mongo6 y se emplea un archivo de configuraciones para su creación.

A continuación, se muestra el archivo *yml* del cluster.

```
services:
  mongo1:
    image: mongo:6
    hostname: mongo1
    container_name: mongo1
    ports:
      - 27017:27017
    entrypoint: ["mongod", "--replSet", "ReplicaSet", "--bind_ip",
    ↪  "localhost,mongo1"]
    volumes:
      - mongo1_data:/data/db
  mongo2:
    image: mongo:6
    hostname: mongo2
    container_name: mongo2
    ports:
      - 27018:27017
    entrypoint: ["mongod", "--replSet", "ReplicaSet", "--bind_ip",
    ↪  "localhost,mongo2"]
    volumes:
      - mongo2_data:/data/db
  mongo3:
    image: mongo:6
    hostname: mongo3
    container_name: mongo3
    ports:
      - 27019:27017
```

```
    entrypoint: ["mongod", "--replSet", "ReplicaSet", "--bind_ip",
    ↪  "localhost,mongo3"]
  mongo4:
    image: mongo:6
    hostname: mongo4
    container_name: mongo4
    ports:
      - 27020:27017
    entrypoint: ["mongod", "--replSet", "ReplicaSet", "--bind_ip",
    ↪  "localhost,mongo4"]
  mongosetup:
    image: mongo:6
    depends_on:
      - mongo1
      - mongo2
      - mongo3
      - mongo4
    volumes:
      - .:/scripts
    restart: "no"
    entrypoint: [ "bash", "/scripts/mongo_setup.sh"]
volumes:
  mongo1_data:
  mongo2_data:
```

Se agrega la configuración del cluster:

```
#!/bin/bash
sleep 10

mongosh --host mongo1:27017 <<EOF
  var cfg = {
    "_id": "ReplicaSet",
    "version": 1,
    "members": [
      {
        "_id": 0,
```

```
        "host": "mongo1:27017",
        "priority": 2
      },
      {
        "_id": 1,
        "host": "mongo2:27017",
        "priority": 0
      },
      {
        "_id": 2,
        "host": "mongo3:27017",
        "priority": 0
      },
      {
        "_id": 3,
        "host": "mongo4:27017",
        "priority": 0
      }
    ]
  };
  rs.initiate(cfg);
EOF
```

Una vez se ha creado el cluster, es posible conocer su estado mediante el siguiente comando:

```
$ docker exec -it mongo1 mongosh --eval "rs.status()"
```

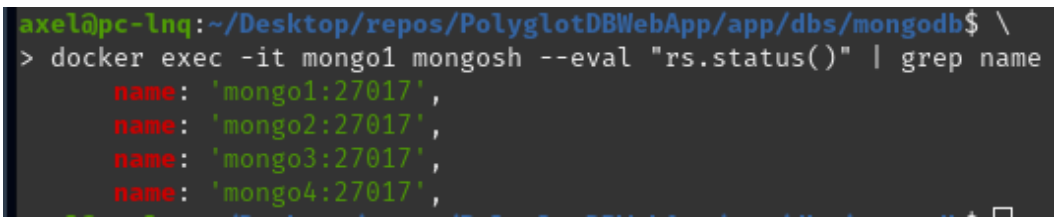

Figura 6: Cluster en MongoDB

Se verifica la existencia de registros en mongo haciendo uso de mongosh.

```
axel@pc-lnq:~/Desktop/repos/PolyglotDBWebApp/app/dbs/mongodb$ mongosh --host --quiet localhost:27017
ReplicaSet [direct: primary] test> use diagramas;
switched to db diagramas
ReplicaSet [direct: primary] diagramas> db.diagramas.find({"_id":1});
[
  {
    _id: 1,
    usuario_id: 1,
    nombre_diagrama: 'Diagrama 1',
    tipo_diagrama: 'ER',
    contenido: {
      entidades: [
        { nombre: 'Entidad1', atributos: [ 'atributo1', 'atributo2' ] }
      ],
      relaciones: [
        {
          nombre: 'Relacion1',
          entidades: [ 'Entidad1', 'Entidad2' ],
          cardinalidad: '1:1'
        }
      ]
    },
    fecha_creacion: ISODate('2024-06-02T06:47:01.410Z'),
    fecha_modificacion: ISODate('2024-06-02T06:47:01.410Z')
  }
]
```

Figura 7: Registros en MongoDB

3.3.2. Creación de esquema

Para la creación del cluster, se elaboró un script de inicialicazión que crea el esquema del proyecto en caso de no existir.

```
// Crea el schema para los diagramas
db = db.getSiblingDB("diagramas");
// Si el esquema existe, no hagas nada
var collectionExists = db.getCollectionInfos({ name: "diagramas"
↪  }).length > 0;
if (collectionExists) {
  quit(); // Exit the script
}
db.createCollection(
  "diagramas",
  {
    validator: {
      $jsonSchema: {
        bsonType: "object",
        required: ["_id", "usuario_id", "nombre_diagrama",
        ↪  "tipo_diagrama", "contenido", "fecha_creacion",
        ↪  "fecha_modificacion"],
```

```
properties: {
  _id: {
    bsonType: "int"
  },
  usuario_id: {
    bsonType: "int"
  },
  nombre_diagrama: {
    bsonType: "string"
  },
  tipo_diagrama: {
    enum: ["ER", "R", "F"]
  },
  contenido: {
    bsonType: "object",
    required: ["entidades", "relaciones"],
    properties: {
      entidades: {
        bsonType: "array",
        items: {
          bsonType: "object",
          required: ["nombre", "atributos"],
          properties: {
            nombre: {
              bsonType: "string"
            },
            atributos: {
              bsonType: "array",
              items: {
                bsonType: "string"
              }
            }
          }
        }
      },
      relaciones: {
```

```
                bsonType: "array",
                items: {
                  bsonType: "object",
                  required: ["nombre", "entidades", "cardinalidad"],
                  properties: {
                    nombre: {
                      bsonType: "string"
                    },
                    entidades: {
                      bsonType: "array",
                      items: {
                        bsonType: "string"
                      }
                    },
                    cardinalidad: {
                      bsonType: "string",
                      pattern: "^\\d+:\\d+|\\*:\\d+|\\d+:\\*|\\*|\\*$"
                    }
                  }
                }
              }
            },
            fecha_creacion: {
              bsonType: "date"
            },
            fecha_modificacion: {
              bsonType: "date"
            }
          }
        }
      }
    }
);
```

3.3.3. Carga inicial

Como se mencionó anteriormente, se creó un script de inicialización, por lo cual se cuenta con la inserción de 3 registros.

```
db = db.getSiblingDB("diagramas");

// Si hay documentos, no hagas nada
var count = db.diagramas.count();
if (count > 0) {
  quit(); // Exit the script
}

db.diagramas.insertMany([
  {
    "_id": 1,
    "usuario_id": 1,
    "nombre_diagrama": "Diagrama 1",
    "tipo_diagrama": "ER",
    "contenido": {
      "entidades": [{
          "nombre": "Entidad1",
          "atributos": ["atributo1", "atributo2"]
        }],
      "relaciones": [{
          "nombre": "Relacion1",
          "entidades": ["Entidad1", "Entidad2"],
          "cardinalidad": "1:1"
        }]
    },
    "fecha_creacion": new Date(),
    "fecha_modificacion": new Date()
  },
  ... Registros 2 y 3
]);
```

3.4. Neo4J

Para la base de datos Neo4J se empleó la imagen oficial de neo4j 5.20 enterprise.

3.4.1. Entorno

Al igual que las otras bases NoSQL, se empleó un archivo docker-compose para la elaboración del cluster.

```
services:
  neo4j1:
    image: neo4j:5.20.0-enterprise
    container_name: neo4j1
    hostname: neo4j1
    ports:
      - "7474:7474"
      - "7473:7473"
      - "7687:7687"
    networks:
      - neo4j-cluster
    environment:
      NEO4J_initial_server_mode__constraint: PRIMARY
      NEO4J_dbms_cluster_discovery_endpoints: neo4j1:5000,neo4j2:
5000,neo4j3:5000
      NEO4J_ACCEPT_LICENSE_AGREEMENT: yes
      NEO4J_server_bolt_advertised__address: neo4j1:7687
      NEO4J_server_http_advertised__address: neo4j1:7474
      NEO4J_AUTH: neo4j/neo4j123
      NEO4J_initial_dbms_default__primaries__count: 3
      NEO4J_initial_dbms_default__database: diagramas
    volumes:
      - neo4j1-data:/data
      - neo4j1-logs:/logs

  neo4j2:
    image: neo4j:5.20.0-enterprise
    container_name: neo4j2
```

```
    hostname: neo4j2
    ports:
      - "8474:7474"
      - "8473:7473"
      - "8687:7687"
    networks:
      - neo4j-cluster
    environment:
      NEO4J_initial_server_mode__constraint: PRIMARY
      NEO4J_dbms_cluster_discovery_endpoints: neo4j1:5000,neo4j2:
5000,neo4j3:5000
      NEO4J_ACCEPT_LICENSE_AGREEMENT: yes
      NEO4J_server_bolt_advertised__address: neo4j2:7687
      NEO4J_server_http_advertised__address: neo4j2:7474
      NEO4J_AUTH: neo4j/neo4j123
      NEO4J_initial_dbms_default__primaries__count: 3
      NEO4J_initial_dbms_default__database: diagramas
    volumes:
      - neo4j2-data:/data
      - neo4j2-logs:/logs

  neo4j3:
    image: neo4j:5.20.0-enterprise
    container_name: neo4j3
    hostname: neo4j3
    ports:
      - "9474:7474"
      - "9473:7473"
      - "9687:7687"
    networks:
      - neo4j-cluster
    environment:
      NEO4J_initial_server_mode__constraint: PRIMARY
      NEO4J_dbms_cluster_discovery_endpoints: neo4j1:5000,neo4j2:
5000,neo4j3:5000
      NEO4J_ACCEPT_LICENSE_AGREEMENT: yes
```

```
      NEO4J_server_bolt_advertised__address: neo4j3:7687
      NEO4J_server_http_advertised__address: neo4j3:7474
      NEO4J_AUTH: neo4j/neo4j123
      NEO4J_initial_dbms_default__primaries__count: 3
      NEO4J_initial_dbms_default__database: diagramas
    volumes:
      - neo4j3-data:/data
      - neo4j3-logs:/logs
networks:
  neo4j-cluster:
    driver: bridge

volumes:
  neo4j1-data:
  neo4j1-logs:
  neo4j2-data:
  neo4j2-logs:
  neo4j3-data:
  neo4j3-logs:
```

Cada uno de los nodos cuenta con su propio volumen para que los datos persistan al crear y eliminar el contenedor.

El cluster se crea la instrucción:

```
$ docker compose up -d
```

Una vez creado, es posible utilizar la interfaz web y observar el estado del cluster y los datos ingresados.

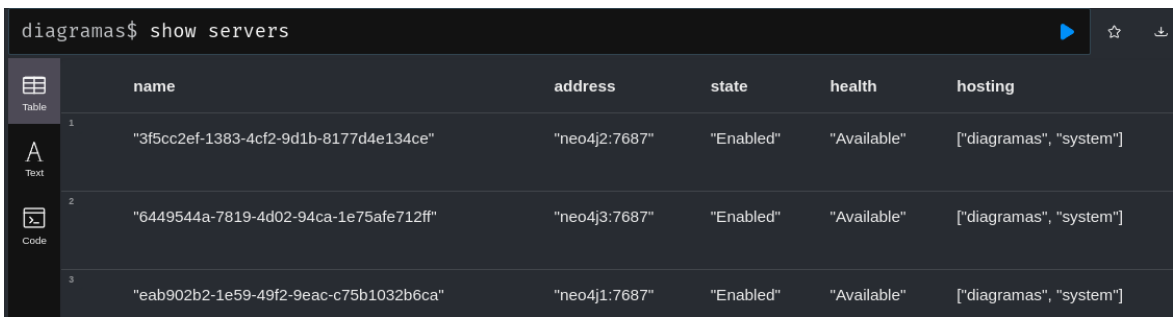

Figura 8: Cluster en Neo4j

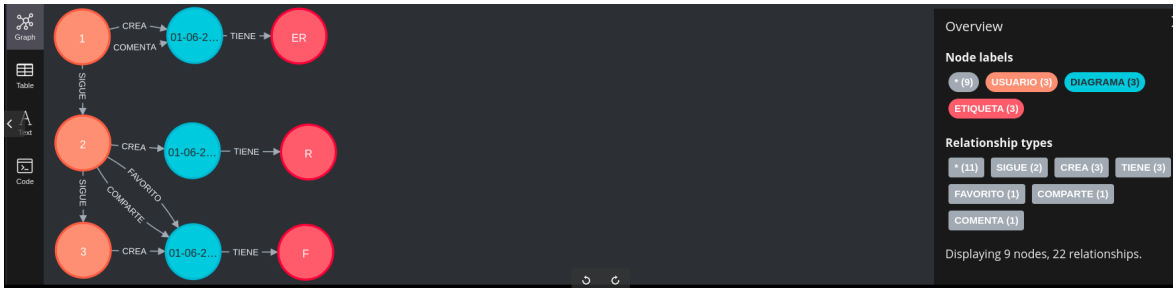

Figura 9: Vista de grafo de datos en Neo4j

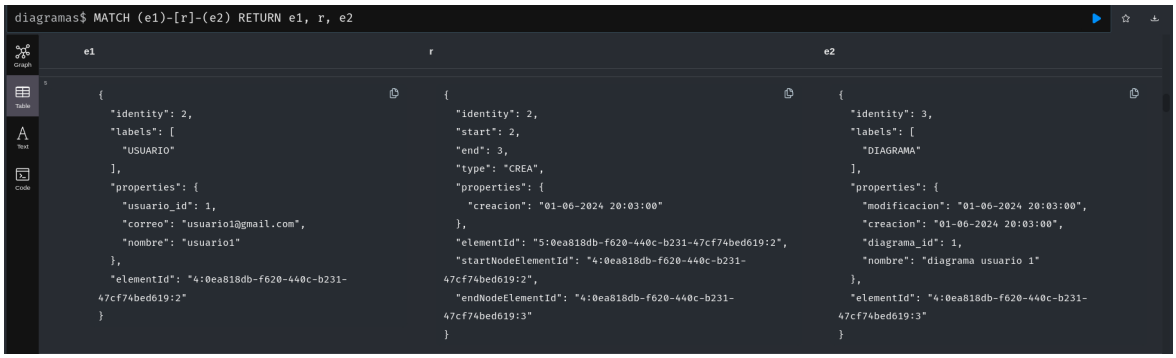

Figura 10: Vista de registros en Neo4j

3.4.2. Creación de entidades y relaciones inicial

Para llevar a cabo esta tarea, se empleó la aplicación web arrow de Neo4j. Se crearon 3 usuarios, 3 diagramas y 3 etiquetas, así como las relaciones entre las entidades.

Posterior a la elaboración del diagrama, se generó el código de cypher mediante la misma herramienta y se creó un script de bash para su ejecución.

Script de cypher:

```
CREATE (`USUARIO 3`:USUARIO {usuario_id: 3, nombre: "usuario3", correo:
↪  "usuario3@gmail.com"})<-[:SIGUE]-(`USUARIO 2`:USUARIO {usuario_id:
↪  2, nombre: "usuario2", correo:
↪  "usuario2@gmail.com"})<-[:SIGUE]-(`USUARIO 1`:USUARIO {usuario_id:
↪  1, nombre: "usuario1", correo: "usuario1@gmail.com"})-[:CREA
↪  {creacion: "01-06-2024 20:03:00"}]->(`DIAGRAMA 1`:DIAGRAMA
↪  {diagrama_id: 1, nombre: "diagrama usuario 1", creacion:
↪  "01-06-2024 20:03:00", modificacion: "01-06-2024
↪  20:03:00"})-[:TIENE]->(:ETIQUETA {nombre: "ER"}),
```

```
(`USUARIO 2`)-[:FAVORITO]->(`DIAGRAMA 3`:DIAGRAMA {diagrama_id: 3,
↪ nombre: "diagrama usuario 3", creacion: "01-06-2024 20:03:00",
↪ modificacion: "01-06-2024 20:03:00"})<-[:COMPARTE {comparticion:
↪ "01-06-2024 20:03:00", permisos: "RW"}]-(`USUARIO 2`)-[:CREA
↪ {creacion: "01-06-2024 20:03:00"}]->(:DIAGRAMA {diagrama_id: 2,
↪ nombre: "diagrama usuario 2", creacion: "01-06-2024 20:03:00",
↪ modificacion: "01-06-2024 20:03:00"})-[:TIENE]->(:ETIQUETA {nombre:
↪ "R"}),
(`USUARIO 3`)-[:CREA {creacion: "01-06-2024 20:03:00"}]->(`DIAGRAMA
↪ 3`)-[:TIENE]->(:ETIQUETA {nombre: "F"}),(`USUARIO 1`)-[:COMENTA
↪ {creacion: "01-06-2024 20:03:00", contenido: "mi primer
↪ documento"}]->(`DIAGRAMA 1`)
```

Script de bash:

```
#!/bin/bash

# Ruta en contenedor
container_file_path="/entities-relations.cypher"

if ! docker exec neo4j1 test -f "${container_file_path}"; then
    echo "Copiando y ejecutando archivo..."
    # Copia y ejecuta archivo
    docker cp entities-relations.cypher neo4j1:"${container_file_path}"
    docker exec -it neo4j1 cypher-shell -u neo4j -p neo4j123 -f
    ↪ /entities-relations.cypher
    echo "Terminando ejecucion."
else
    echo "Archivo encontrado en el contenedor, terminando ejecucion."
fi
```

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: 2024-06-01. 2024. URL: https://docs.riak.com/riak/kv/latest/.

[2] MongoDB, Inc. *MongoDB Documentation*. Recuperado: 2024-06-01. 2024. URL: https://docs.mongodb.com/.

[3] Neo4j, Inc. *Neo4j Documentation*. Recuperado: 2024-06-02. 2024. URL: https://neo4j.com/docs/.

[4] OpenJS Foundation. *Express.js Documentation*. Recuperado: 2024-06-03. 2024. URL: https://expressjs.com/.

[5] OpenJS Foundation. *Node.js Documentation*. Recuperado: 2024-06-03. 2024. URL: https://nodejs.org/en/docs/.

[6] The PostgreSQL Global Development Group. *PostgreSQL Documentation*. Recuperado: 2024-05-31. 2024. URL: https://www.postgresql.org/docs/.