Implementación y Evaluación de una Base de Datos para una Red Social Básica

Introducción:

Este trabajo práctico tiene como objetivo diseñar y modelar un caso real de base de datos para una red social básica. Se explorará cómo implementar funcionalidades esenciales, como registro de usuarios, posteos, interacciones (likes, comentarios) y relaciones entre usuarios (seguir/dejar de seguir).

Para este propósito, se utilizará una base de datos orientada a grafos, Neo4j, y se realizará una comparación con una implementación equivalente en una base de datos relacional, Oracle PL/SQL.

Se evaluará cómo el paradigma de grafos mejora la escalabilidad, rendimiento y facilidad de programación en comparación con una base de datos relacional. Al final, se justificará por qué la elección de Neo4j puede ser más eficiente y económica a largo plazo para este tipo de sistema.

La red social se diseñará con las siguientes características:

1. Nodos principales:

- Usuarios.

- Posteos.

- Cookies de sesión.

2. Relaciones entre nodos:

- Sigue (usuarios siguiendo a otros usuarios).

- Posteo (usuarios publicando contenido).

- Le\_gusto (usuarios interactuando con posteos mediante likes).

- Comentó (usuarios comentando posteos).

3. Funciones principales a implementar:

- Registrarse.

- Postear.

- Dar y quitar likes.

- Seguir y dejar de seguir usuarios.

- Comentar y eliminar comentarios.

**Funciones de Creación**

**Registrar Usuario**

CREATE (:Usuario {id: apoc.create.uuid(), nombre: $nombre, apellido: $apellido, usuario: $usuario, email: $email, contraseña: $contraseña, fecha\_creacion: datetime(), edad: $edad})

Variables**:**

-$nombre: Nombre del usuario.

-$apellido: Apellido del usuario.

-$usuario: Nombre de usuario único.

-$email: Correo electrónico.

-$contraseña: Contraseña del usuario.

-$fecha\_creacion: Fecha y hora de creación del usuario.

-$edad: Edad del usuario.

**Postear**

MATCH (u:Usuario {username: $username})

CREATE (u)-[:POSTEO]->(:Post {id: apoc.create.uuid(), contenido: $contenido, fecha: datetime()})

Variables:

- $username: Nombre de usuario del autor.

- $contenido: Texto del posteo.

**Eliminar Post**

MATCH (p:Post {id: $postId})

DETACH DELETE p

Variables:

- $postId: ID único del posteo.

**Dar Like a un Posteo**

MATCH (u:Usuario {username: $username}), (p:Post {id: $postId})

MERGE (u)-[:LIKE]->(p)

Variables:

- $username: Usuario que da el like.

- $postId: ID del posteo que recibe el like.

**Quitar Like a un Posteo**

MATCH (u:Usuario {username: $username})-[l:LIKE]->(p:Post {id: $postId})

DELETE l

Variables:

- $username: Usuario que quita el like.

- $postId: ID del posteo.

**Seguir a un Usuario**

MATCH (u1:Usuario {username: $username1}), (u2:Usuario {username: $username2})

MERGE (u1)-[:FOLLOW]->(u2)

Variables:

- $username1: Usuario que quiere seguir.

- $username2: Usuario a quien se sigue.

**Dejar de Seguir a un Usuario**

MATCH (u1:Usuario {username: $username1})-[f:FOLLOW]->(u2:Usuario {username: $username2})

DELETE f

Variables:

- $username1: Usuario que deja de seguir.

- $username2: Usuario al que se deja de seguir.

**Comentar un Post**

MATCH (u:Usuario {username: $username}), (p:Post {id: $postId})

CREATE (u)-[:COMENTO {texto: $texto, fecha: datetime()}]->(p)

Variables:

- $username: Usuario que comenta.

- $postId: ID del posteo comentado.

- $texto: Contenido del comentario.

**Eliminar Comentario de un Post**

MATCH (u:Usuario {username: $username})-[c:COMENTO]->(p:Post {id: $postId})

DELETE c

Variables:

- $username: Usuario que elimina el comentario.

- $postId: ID del posteo comentado.

Entidades y Relaciones Oracle:

1. **Usuario**:
   * Atributos: id\_usuario (PK), nombre, apellido, email, usuario, contraseña, fecha\_creacion, edad.
   * Relación:
     + Uno a muchos con Publicaciones.
     + Uno a muchos con Comentarios.
     + Uno a muchos con Likeo\_Comentario.
     + Uno a muchos con Likeo\_Post.
2. **Publicaciones**:
   * Atributos: id\_publicacion (PK), contenido, fecha\_creacion, id\_usuario (FK).
   * Relación:
     + Muchos a uno con Usuario (FK).
     + Uno a muchos con Comentarios.
     + Uno a muchos con Likeo\_Post.
3. **Comentarios**:
   * Atributos: id\_comentario (PK), contenido, fecha\_creacion, id\_usuario (FK), id\_publicacion (FK).
   * Relación:
     + Muchos a uno con Usuario (FK).
     + Muchos a uno con Publicaciones (FK).
     + Uno a muchos con Likeo\_Comentario.
4. **Likeo\_Post**:
   * Atributos: id\_like (PK), fecha\_creacion, id\_usuario (FK), id\_publicacion (FK).
   * Relación:
     + Muchos a uno con Usuario (FK).
     + Muchos a uno con Publicaciones (FK).
5. **Likeo\_Comentario**:
   * Atributos: id\_like (PK), fecha\_creacion, id\_usuario (FK), id\_comentario (FK).
   * Relación:
     + Muchos a uno con Usuario (FK).
     + Muchos a uno con Comentarios (FK).

Imagen que contiene Diagrama

Descripción generada automáticamente

Bentchmarks y análisis

Hablemos del bentchmark, para este punto quisimos plantear una prueba de rendimiento en iguales condiciones; Por lo que procedimos a hacer la misma en un localhost, y ,haciendo la prueba manual en cada database para medir el tiemplo en el que cada database tarda en implementar.

Además, reducimos la estructura del código para la demostración, la cual va a ser, insertar 3 usuarios los cuales van a hacer 10 publicaciones cada uno (intercaladas en la tabla de Oracle), y la última publicación va a ser likeada por los 3 usuarios. El script lo que hará será buscar cual es la publicación likeada por 3 usuarios.

La estructura de tablas de la database en Oracle será:

-Usuarios{ id, nombre, apellido, email, usuario, contraseña, fecha\_creacion, edad}

-Publicaciones{ id, contenido, fecha\_creacion, id(usuario)[FK] }

-Comentarios { id, contenido, fecha\_creacion, id(usuario)[FK], id(publicación)[FK] }

-Likeo\_post {id, fecha\_creacion, , id(usuario)[FK], id(publicación)[FK] }

Y para la database de Neo4j será:

- Usuarios {id, nombre, apellido, email, usuario, contraseña, fecha\_creacion, edad}

- Post {id, contenido, fecha\_creacion}

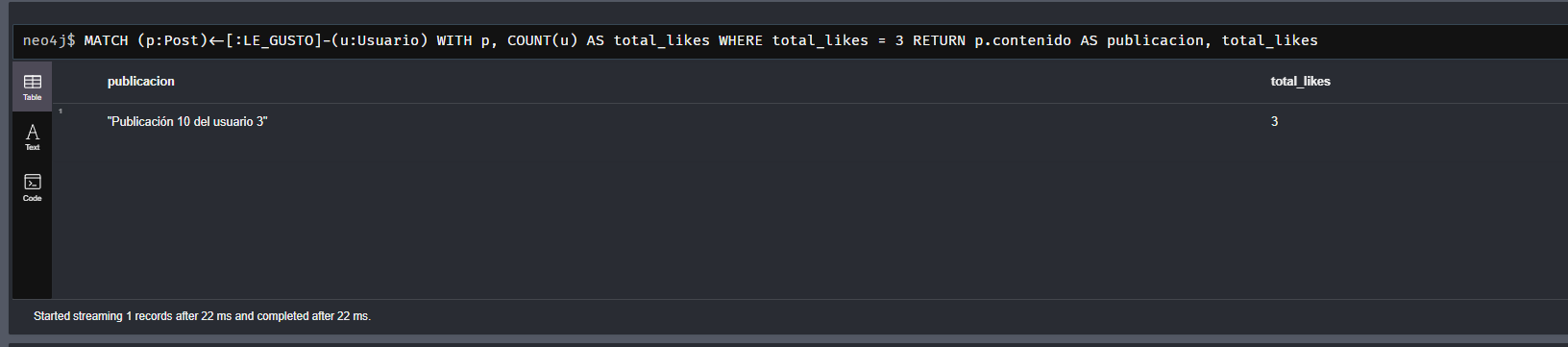
- Relaciones:

- POSTEO {Usuario -> Post}

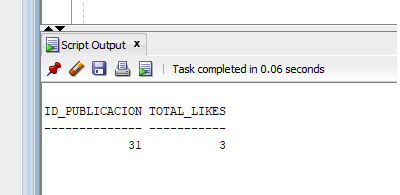
- LE\_GUSTO {Usuario -> Post}

En la prueba lograremos apreciar como Oracle tarda 60ms contra 22ms de Neo4j, pero esto ocurre dado a que tenemos una prueba chica, a gran escala las comparativas se vuelven humillantes para Oracle, si no es que esta ya lo es.

Neo4j:



Oracle:



Los scripts de creación los encontraran el los .txt, y las querys son las siguientes:

NEO:

MATCH (p:Post)<-[:LE\_GUSTO]-(u:Usuario)

WITH p, COUNT(u) AS total\_likes

WHERE total\_likes = 3

RETURN p.contenido AS publicacion, total\_likes

ORACLE:  
SELECT p.id\_publicacion, COUNT(l.id\_like) AS total\_likes

FROM Publicaciones p

JOIN Likeo\_Post l ON p.id\_publicacion = l.id\_publicacion

GROUP BY p.id\_publicacion

HAVING COUNT(l.id\_like) = 3;

Por otro lado Neo4j sobresale en grandes masas de datos debido a su modelo basado en grafos, donde las relaciones están indexadas internamente, lo que permite un acceso eficiente y un rendimiento constante incluso en consultas complejas con múltiples relaciones. A diferencia de Oracle, no necesita realizar costosos JOIN, ya que las conexiones entre nodos son directas. Sin embargo, en datasets pequeños, Oracle puede ser más rápido gracias a su optimización para consultas relacionales simples y su avanzado planificador de consultas. Además, la inicialización en Neo4j introduce cierta sobrecarga que no es significativa en grandes volúmenes, pero puede impactar en pruebas con pocos datos. En resumen, Neo4j es ideal para datos densamente conectados, mientras que Oracle sobresale en operaciones transaccionales tradicionales.

Por lo que viendo lo positivo neo tiene claras ventajas en:

-Eficiencia en Relaciones Complejas:

Neo4j está diseñado para manejar datos densamente conectados, lo que lo hace más eficiente en consultas que requieren múltiples relaciones, como encontrar publicaciones con likes o comentarios en cascada.

-Escalabilidad Natural:

A medida que el dataset crece en tamaño y relaciones, Neo4j mantiene un rendimiento constante al procesar conexiones, mientras que Oracle sufre degradación en consultas con múltiples JOIN.

-Modelo Intuitivo:

La representación como grafos es más natural para un proyecto como una red social, ya que las relaciones como likes, comentarios y publicaciones se modelan directamente sin necesidad de claves foráneas.  
  
Y una gran desventaja en rendimiento en:

Datasets pequeños:

En volúmenes de datos pequeños, lo que puede ocasionar bentchmarks curiosos, Oracle puede ser más rápido debido a su avanzado optimizador de consultas y su bajo costo de inicialización.

Curva de aprendizaje:

Neo4j puede tener una curva de aprendizaje más alta para quienes están acostumbrados a SQL, y su ecosistema de herramientas es más limitado comparado con Oracle, que cuenta con décadas de desarrollo y soporte.  
  
  
Determinamos entonces:

En conclusión, este trabajo práctico permitió analizar y comparar dos paradigmas de bases de datos, relacional y grafos, en el contexto de una red social básica. Neo4j destacó por su diseño moderno basado en grafos, que simplifica la representación y consulta de relaciones densas como likes y comentarios, ofreciendo un rendimiento constante en grandes volúmenes de datos. Su modelo elimina la necesidad de operaciones como JOIN, lo que no solo reduce la complejidad del desarrollo, sino también los costos computacionales en consultas complejas. Por otro lado, Oracle mostró su fortaleza en operaciones transaccionales y consultas relacionales complejas en datasets pequeños y medianos, donde su optimización relacional es difícil de superar. Sin embargo, a medida que el volumen de datos y las conexiones crecen, Neo4j demuestra ser más eficiente, destacándose en rendimiento y escalabilidad para sistemas densamente conectados como redes sociales, mientras que también facilita la programación al alinearse intuitivamente con el modelo de datos.