Gestor de Base de Datos No Relacional con Árboles AVL y Persistencia en JSON

Alejandro Nuñez Barrera
Código: 20231020139
Ingeniería de Sistemas
Universidad Distrital Francisco José de Caldas

David Felipe García León
Código: 20231020202
Ingeniería de Sistemas
Universidad Distrital Francisco José de Caldas

***Resumen*—Este documento presenta el proyecto final de la asignatura Ciencias de la Computación I. Se implementó un gestor de base de datos no relacional que almacena objetos en formato JSON, utiliza un árbol AVL como índice por clave principal y mantiene la persistencia en un archivo de texto plano mediante JSON por línea. Se describe la arquitectura general del sistema, se justifica la elección del árbol AVL como estructura de indexación y se muestran ejemplos de las operaciones soportadas.**

***Index Terms*—Árbol AVL, bases de datos no relacionales, JSON, persistencia, estructuras de datos.**

I. Introducción

El objetivo del proyecto es desarrollar un gestor de base de datos no relacional que permita almacenar, consultar, actualizar y eliminar objetos JSON de manera eficiente. Para lograrlo, se integra una estructura de datos autobalanceada (árbol AVL) con un mecanismo simple de persistencia basado en archivos de texto plano.

El contexto académico del trabajo es la asignatura Ciencias de la Computación I del programa de Ingeniería de Sistemas de la Universidad Distrital Francisco José de Caldas. El proyecto busca poner en práctica conceptos de estructuras de datos, algoritmos de búsqueda y manejo de archivos.

II. Descripción de la Arquitectura

La arquitectura propuesta se organiza en capas y módulos independientes, lo que favorece la claridad del código y su mantenibilidad. A alto nivel, el sistema se compone de cuatro módulos principales:

- **Módulo de índice (árbol AVL)**: implementado en el archivo `avl_tree.py`, mantiene en memoria un árbol AVL donde cada nodo almacena una clave principal (`id`) y el objeto JSON completo asociado.
- **Módulo de persistencia**: definido en `storage_manager.py`, se encarga de leer y escribir los objetos JSON en un archivo de texto plano utilizando el formato JSON por línea (JSON Lines), es decir, un objeto JSON por cada línea del archivo.
- **Módulo gestor**: contenido en `database_manager.py`, coordina el árbol AVL y la capa de persistencia. Ofrece operaciones de alto nivel como insertar, buscar por `id`, actualizar, eliminar y consultar por criterios.
- **Interfaz de usuario**: implementada en `main.py`, proporciona un menú de consola que permite al usuario final interactuar con el sistema y ejecutar las operaciones principales de la base de datos.

Al iniciar la aplicación, el módulo gestor carga todos los registros existentes desde el archivo de persistencia mediante el *StorageManager* y los inserta en el árbol AVL. De esta forma, el índice en memoria refleja el estado actual del archivo y permite realizar búsquedas por clave principal en tiempo logarítmico.

Cada operación de modificación (inserción, actualización o eliminación) sigue el mismo patrón: primero se actualiza el árbol AVL y luego se sincroniza el archivo sobrescribiendo la lista completa de registros. Este enfoque simplifica el manejo de la persistencia y garantiza que el archivo siempre refleje el estado consistente del gestor.

III. Justificación del Árbol AVL

Para el índice por clave principal se eligió un árbol AVL, que es un árbol binario de búsqueda autobalanceado. En un árbol AVL se cumple que, para cada nodo, la diferencia entre las alturas de sus subárboles izquierdo y derecho es, como máximo, uno. Esta condición de equilibrio garantiza que la altura total del árbol crezca en orden $O(\log n)$ respecto al número de nodos.

En este proyecto, la clase `AVLTree` mantiene en cada nodo un campo de altura y calcula el factor de balance como la diferencia entre las alturas de los subárboles. Después de cada inserción o eliminación, se actualizan las alturas y se verifica el factor de balance. Si un nodo queda desbalanceado, se corrige mediante rotaciones simples o dobles (casos LL, RR, LR y RL), que se implementan como operaciones locales de costo constante.

La elección del árbol AVL se justifica por las siguientes razones:

- **Eficiencia garantizada**: al mantener la altura del árbol acotada por $O(\log n)$ en el peor caso, las operaciones de búsqueda, inserción y borrado por `id` se ejecutan en tiempo $O(\log n)$.
- **Simetría con el contenido del curso**: los árboles AVL son una estructura clásica en ciencias de la computación, lo que facilita su explicación teórica y la verificación de su correcta implementación.

- **Implementación directa**: la lógica de balanceo se basa en una combinación clara de cálculo de alturas y rotaciones, lo que resulta adecuada para un proyecto docente.

IV. Ejemplos de Operaciones

En esta sección se describen brevemente las operaciones que soporta el gestor, ilustrando su comportamiento desde el punto de vista del usuario y su efecto en la arquitectura interna.

IV-A. Inserción de un registro

Para insertar un nuevo objeto, el usuario proporciona un JSON que contiene, al menos, el campo `id`. Por ejemplo:

```
{"id": 1, "nombre": "Ana", "edad": 20}
```

La interfaz de consola pasa este diccionario al método `insert` del *DatabaseManager*. Internamente:

1. Se obtiene la clave principal `id`.
2. Se inserta el par (`id`, objeto) en el árbol AVL.
3. Se leen todos los registros actuales del archivo.
4. Si el `id` ya existía, se reemplaza el registro; en caso contrario, se añade al final.
5. Se sobrescribe el archivo con la lista actualizada.

Así, el nuevo registro queda disponible tanto en memoria como en el archivo de persistencia.

IV-B. Búsqueda por clave principal

Para buscar un registro por su clave principal, el usuario indica el valor de `id` mediante el menú. El gestor invoca `get_by_id`, que delega la operación al árbol AVL. Gracias a la propiedad de árbol binario de búsqueda y al balanceo, la búsqueda recorre un camino desde la raíz hasta un nodo hoja en tiempo $O(\log n)$.

IV-C. Actualización de un registro

La actualización se realiza indicando primero el `id` del registro a modificar y luego un nuevo objeto JSON con los campos actualizados. El sistema:

1. Verifica que exista un registro con ese `id` en el AVL.
2. Fuerza que el nuevo objeto conserve el mismo `id`.
3. Inserta el nuevo objeto en el árbol, sobrescribiendo el valor anterior.
4. Recorre la lista de registros del archivo, reemplaza el objeto correspondiente y guarda todos los registros de nuevo.

IV-D. Eliminación de un registro

Para eliminar un registro, el usuario proporciona el `id`. El método `delete` del *DatabaseManager*:

1. Llama a `delete` en el árbol AVL para eliminar el nodo correspondiente, aplicando las rotaciones necesarias para mantener el balance.
2. Carga todos los registros desde el archivo y filtra aquellos cuyo `id` sea distinto.
3. Sobrescribe el archivo con la lista filtrada.

IV-E. Consultas por criterios

Además de la búsqueda por clave principal, el gestor permite realizar consultas por otros campos utilizando un predicado. Por ejemplo, se puede buscar todos los registros con `edad >25` o `ciudad == "Bogotá"`. Estas consultas se implementan como búsquedas lineales sobre el archivo de persistencia, lo que simplifica la lógica y evita mantener múltiples índices en memoria.

V. Conclusiones

El proyecto integra de manera práctica conceptos de estructuras de datos (árboles AVL), persistencia en archivos y manejo de objetos JSON. La arquitectura modular facilita la comprensión del sistema y su posible extensión futura, por ejemplo, hacia nuevos tipos de índices o interfaces de usuario más avanzadas.