Arquitectura Técnica y Diseño de BonsaiDB

Autor: John Fajardo
Versión: 1.1
Fecha: 11 de julio de 2025

---

1. Introducción y Motivación

BonsaiDB es un motor de almacenamiento ligero, desarrollado en `C++17`, que surge como un proyecto para explorar los fundamentos de los sistemas de bases de datos. A diferencia de soluciones comerciales robustas o bases de datos embebidas como *SQLite*, el objetivo principal de BonsaiDB es demostrar una comprensión profunda de conceptos clave como la gestión de memoria, el acceso a disco, las estructuras de datos de indexación (Árbol B+) y la programación concurrente.

Este documento detalla la arquitectura interna del motor, el modelo de almacenamiento en disco, el diseño del índice B+ y las estrategias de concurrencia implementadas para garantizar la integridad de los datos en un entorno multihilo.

---

2. Modelo de Almacenamiento en Disco

El almacenamiento persistente es la base de cualquier motor de base de datos. En BonsaiDB, el acceso al disco se optimiza mediante un modelo de paginación y un formato de registro de tamaño fijo, gestionado por la clase `FileManager`.

2.1. Paginación

El archivo de la base de datos (`.db`) no se manipula byte a byte. En su lugar, se organiza en bloques de tamaño fijo denominados **páginas**.

- **Tamaño de Página**: Cada página tiene un tamaño fijo de **4096 bytes (4 KB)**, definido por la constante `PAGE_SIZE`. Este es un estándar en la industria, ya que a menudo se alinea con el tamaño de página del sistema operativo, optimizando las operaciones de I/O.
- **Gestión de Páginas**: La clase `FileManager` centraliza la responsabilidad de leer (`readRawPage`) y escribir (`writeRawPage`) estas páginas. También gestiona la asignación de nuevas páginas (`allocatePage`) cuando el espacio se agota.

2.2. Formato del Registro

Para simplificar la serialización y el cálculo de espacio, los registros emplean una estructura de tamaño fijo definida en la clase `Record`.

- **Estructura**: Un registro se compone de los siguientes campos:

- `id`: `int32_t`
- `name`: `char[50]`
- `age`: `int32_t`
- `balance`: `double`

- **Tamaño Fijo**: El tamaño total de un registro serializado es constante, lo que permite un cálculo predecible del espacio disponible en cada página de datos.

2.3. Serialización y Deserialización

La conversión de la estructura `Record` a un formato de bytes para su almacenamiento en disco es un proceso crítico.

- **Técnica Utilizada**: La serialización se realiza mediante copias de memoria de bajo nivel con `memcpy`. Este enfoque es extremadamente eficiente para estructuras de datos de layout fijo.
- **Proceso**: Los métodos `Record::serialize` y `Record::deserialize` copian los bytes de cada miembro de la estructura directamente entre el objeto y un buffer (`std::vector<char>`).
- **Nota sobre Portabilidad**: Este método de serialización es altamente performante pero no es portable entre arquitecturas con diferente *endianness* o reglas de alineamiento de memoria. Para un sistema de producción, se requeriría un formato de serialización neutral (ej. Protocol Buffers, FlatBuffers o serialización manual campo a campo).

---

3. Diseño del Índice de Árbol B+

Realizar una búsqueda lineal en todas las páginas para localizar un registro es un método ineficiente. BonsaiDB implementa un **Árbol B+ (B+ Tree)** como estructura de indexación primaria para acelerar las búsquedas basadas en la clave `id`.

3.1. Estructura del Nodo

El árbol está compuesto por nodos (`BPlusNode`), y cada uno se almacena en una página de 4 KB. Cada nodo contiene:

- `is_leaf`: Un booleano que especifica si el nodo es una hoja (apunta a datos) o un nodo interno (apunta a nodos hijos).
- `keys`: Un vector que almacena las claves (`id` de los registros).
- `children_page_ids`: (Solo en nodos internos) Un vector de IDs de página que apuntan a los nodos hijos.

- `data_page_ids`: (Solo en nodos hoja) Un vector de IDs de página que apuntan a las páginas de datos donde residen los registros.
- `next_leaf_id`: (Solo en nodos hoja) Un puntero al siguiente nodo hoja en la secuencia, lo que permite escaneos de rango eficientes.

3.2. Algoritmos Fundamentales

La clase `BPlusTree` implementa la lógica para manipular la estructura del árbol.

- **Búsqueda (`search`)**: El algoritmo navega desde la raíz hacia las hojas. En cada nodo interno, utiliza la clave para determinar cuál es el siguiente nodo hijo a visitar. El proceso concluye al llegar a un nodo hoja, donde se localiza la clave y el `page_id` del dato correspondiente.
- **Inserción (`insert`)**: La inserción se realiza siempre en un nodo hoja. Si la nueva clave excede la capacidad del nodo (definida por `BPLUS_TREE_ORDER`), este se divide en dos:
  - *División de Nodos (Splitting)*: El nodo se divide en dos nodos hermanos. La clave mediana es "promovida" al nodo padre para actuar como separador. Este proceso de división puede propagarse recursivamente hasta la raíz, lo que puede incrementar la altura del árbol.
- **Eliminación (`remove`)**: La eliminación localiza la clave en un nodo hoja y la suprime. El diseño actual es simplificado: la clave y su puntero a datos se eliminan del nodo, pero el árbol no implementa la fusión o redistribución de claves si un nodo queda por debajo de su capacidad mínima.

---

4. Gestión de Concurrencia

Para soportar operaciones simultáneas de múltiples hilos, es crucial proteger el acceso a las estructuras de datos compartidas. BonsaiDB utiliza un mecanismo de bloqueo a nivel de motor.

- **Estrategia de Bloqueo**: Se emplea un `std::shared_mutex` en la clase `DatabaseEngine`, implementando un patrón de bloqueo de *lectura-escritura (Reader-Writer Lock)*.
- **Operaciones de Lectura (`find`)**: Adquieren un bloqueo compartido (`std::shared_lock`). Esto permite que múltiples hilos lean la base de datos de forma concurrente, ya que estas operaciones no modifican las estructuras de datos.
- **Operaciones de Escritura (`insert`, `remove`)**: Adquieren un bloqueo exclusivo (`std::unique_lock`). Este bloqueo garantiza que solo un hilo pueda modificar el árbol B+ o las páginas de datos a la vez, previniendo condiciones de carrera y manteniendo la consistencia.

Esta estrategia, aunque limita el paralelismo al serializar todas las operaciones de escritura, es una solución efectiva y robusta para garantizar la seguridad de los hilos (*thread-safety*) sin introducir la complejidad de bloqueos más granulares (a nivel de página o registro), que podrían derivar en `deadlocks`.