Technical Architecture and Design of BonsaiDB

Author: John Fajardo
Version: 1.1
Date: July 11, 2025

---

1. Introduction and Motivation

BonsaiDB is a lightweight storage engine developed in `C++17`, conceived as a project to explore the fundamentals of database systems. Unlike robust commercial solutions or embedded databases like *SQLite*, the main goal of BonsaiDB is to demonstrate a deep understanding of key concepts such as memory management, disk access, indexing data structures (B+ Tree), and concurrent programming.

This document details the engine's internal architecture, the disk storage model, the B+ tree index design, and the concurrency strategies implemented to ensure data integrity in a multithreaded environment.

---

2. Disk Storage Model

Persistent storage is the foundation of any database engine. In BonsaiDB, disk access is optimized through a paging model and a fixed-size record format, managed by the `FileManager` class.

2.1. Paging

The database file (`.db`) is not manipulated byte by byte. Instead, it is organized into fixed-size blocks called **pages**.

- **Page Size**: Each page has a fixed size of **4096 bytes (4 KB)**, defined by the `PAGE_SIZE` constant. This is an industry standard, as it often aligns with the operating system's page size, optimizing I/O operations.
- **Page Management**: The `FileManager` class centralizes the responsibility of reading (`readRawPage`) and writing (`writeRawPage`) these pages. It also manages the allocation of new pages (`allocatePage`) when space runs out.

2.2. Record Format

To simplify serialization and space calculation, records use a fixed-size structure defined in the `Record` class.

- **Structure**: A record consists of the following fields:

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

- **Fixed Size**: The total size of a serialized record is constant, allowing predictable calculation of available space in each data page.

2.3. Serialization and Deserialization

Converting the `Record` structure to a byte format for storage on disk is a critical process.

- **Technique Used**: Serialization is performed via low-level memory copies using `memcpy`. This approach is extremely efficient for fixed-layout data structures.
- **Process**: The `Record::serialize` and `Record::deserialize` methods copy the bytes of each structure member directly between the object and a buffer (`std::vector<char>`).
- **Portability Note**: This serialization method is highly performant but not portable between architectures with different *endianness* or memory alignment rules. For a production system, a neutral serialization format would be required (e.g., Protocol Buffers, FlatBuffers, or manual field-by-field serialization).

---

3. B+ Tree Index Design

Performing a linear search across all pages to locate a record is inefficient. BonsaiDB implements a **B+ Tree** as the primary indexing structure to accelerate key-based (`id`) searches.

3.1. Node Structure

The tree consists of nodes (`BPlusNode`), each stored in a 4 KB page. Each node contains:

- `is_leaf`: A boolean specifying if the node is a leaf (points to data) or an internal node (points to child nodes).
- `keys`: A vector storing the keys (`id` of the records).
- `children_page_ids`: (Internal nodes only) A vector of page IDs pointing to child nodes.
- `data_page_ids`: (Leaf nodes only) A vector of page IDs pointing to data pages where records reside.
- `next_leaf_id`: (Leaf nodes only) A pointer to the next leaf node in the sequence, enabling efficient range scans.

3.2. Core Algorithms

The `BPlusTree` class implements the logic to manipulate the tree structure.

- **Search (`search`)**: The algorithm traverses from the root to the leaves. In each internal node, it uses the key to determine which child node to visit next. The process ends at a leaf node, where the key and corresponding `page_id` are located.
- **Insertion (`insert`)**: Insertion always occurs in a leaf node. If the new key exceeds the node's capacity (defined by `BPLUS_TREE_ORDER`), the node is split into two siblings:
  - *Node Splitting*: The node is split into two sibling nodes. The median key is "promoted" to the parent node to act as a separator. This splitting process can propagate recursively up to the root, potentially increasing the tree's height.
- **Removal (`remove`)**: Removal locates the key in a leaf node and deletes it. The current design is simplified: the key and its data pointer are removed from the node, but the tree does not implement key merging or redistribution if a node falls below its minimum capacity.

---

4. Concurrency Management

To support simultaneous operations from multiple threads, it is crucial to protect access to shared data structures. BonsaiDB uses an engine-level locking mechanism.

- **Locking Strategy**: A `std::shared_mutex` is used in the `DatabaseEngine` class, implementing a *reader-writer lock* pattern.
- **Read Operations (`find`)**: Acquire a shared lock (`std::shared_lock`). This allows multiple threads to read the database concurrently, since these operations do not modify the data structures.
- **Write Operations (`insert, remove`)**: Acquire an exclusive lock (`std::unique_lock`). This lock ensures that only one thread can modify the B+ tree or data pages at a time, preventing race conditions and maintaining consistency.

This strategy, while limiting parallelism by serializing all write operations, is an effective and robust solution to ensure thread-safety without introducing the complexity of finer-grained locks (at page or record level), which could lead to deadlocks.