RookDB: Design and Implementation of a Modular Storage Manager

Abstract

This paper presents the architecture and implementation of RookDB, a modular storage manager that combines the core functionality of traditional database storage engines with modern metadata representation techniques. The system is organized into four integrated layers: a catalog layer that stores logical metadata in a human-readable and extensible JSON format, a table layer that manages physical file layout using fixed-size headers and contiguous data pages, a page layer implementing a slotted-page organization with bidirectional growth for efficient tuple management, and a buffer layer that provides an in-memory workspace for data manipulation and bulk loading.

The architecture enforces strong namespace isolation by mapping each database to a dedicated directory, ensuring that no database can access or modify the storage of another, thereby providing a built-in security boundary at the storage layer. Together, these layers form a clean and modular storage hierarchy that decouples metadata handling, file-level organization, and per-page record layout. The resulting system enables predictable page allocation, constant-time page access, efficient free-space utilization, and reduced disk I/O through controlled buffering. This design offers a flexible foundation for future extensions, including advanced compaction strategies, broader data-type support, and further optimizations in storage-level workflows.

1 Introduction

RookDB is a lightweight storage manager designed to isolate and study the fundamental components of a modern database storage engine. Instead of implementing full DBMS functionality, RookDB focuses on the essential mechanisms that govern metadata handling, table organization, page layout, and in-memory buffering. The system emphasizes clarity, modularity, and file-based isolation, with each database stored in its own directory to prevent cross-database interference.

In addition to its architectural simplicity, RookDB provides a practical platform for empirical evaluation of storage behavior. We conducted benchmarks using a 1000K-row CSV dataset to analyze the efficiency of bulk loading under different storage configurations. These experiments compared ingestion performance across variable page sizes, measured the number of I/O operations required during loading, and evaluated the impact of enabling versus disabling the buffer manager. The results highlight how page sizing, buffering strategies, and tuple placement policies directly influence overall ingestion time and system efficiency.

2 System Model and Goals

RookDB runs as a single-threaded, file-backed storage engine that focuses solely on low-level storage behavior. The system omits higher-level DBMS features such as query processing or concurrency control,

concentrating instead on the core mechanisms that govern metadata storage, table organization, page layout, and in-memory buffering.

The primary goals of RookDB are:

- maintain logical metadata in a lightweight JSON-based catalog;
- provide deterministic file organization with per-database directory isolation;
- support efficient tuple storage through a slotted-page format;
- reduce I/O costs via a fixed-size in-memory buffer manager; and
- enable controlled benchmarking of page allocation, I/O behavior, and bulk loading performance.

This simplified model provides a compact environment for studying fundamental storage engine behavior and evaluating design trade-offs.

3 Architecture Overview

The architecture of RookDB follows a modular, layered design that separates logical metadata management from the physical representation of tables and the low-level organization of records within pages. This separation of concerns enforces clear responsibility boundaries, simplifies implementation, and enables individual components to evolve independently.

3.1 Catalog Level

At the highest level, the **Catalog Manager** is responsible for maintaining all logical metadata required for interpreting and organizing the physical storage of the system. RookDB represents catalog information in a hierarchical JSON structure that mirrors the logical namespace of the database. The catalog captures three core concepts: *databases*, *tables*, and *columns*. Each of these components is explicitly encoded in the JSON catalog file, allowing the system to reconstruct all metadata during startup without scanning any physical table files.

The catalog layout follows a nested structure of the form:

```
{
  "databases": {
    "<database_name>": {
      "tables": {
        "<table_name>": {
          "columns": [
            { "name": <colname>, "data_type": <type> },
            ...
          ]
        }
      }
    }
  }
}
```

This representation allows the Catalog Manager to reason about metadata at multiple levels of granularity. For example, the excerpt below illustrates how RookDB registers a logical database named `users`, containing a table `students` with two columns:

```
{
  "databases": {
    "users": {
      "tables": {
        "students": {
          "columns": [
            { "name": "id", "data_type": "INT" },
            { "name": "name", "data_type": "TEXT" }
          ]
        }
      }
    }
  }
}
```

During system initialization, the Catalog Manager loads this JSON file and deserializes it into internal data structures representing databases, tables, and column definitions. Each logical database is mapped to a physical directory within the storage hierarchy, and each table is associated with a specific data file inside its corresponding database directory. This design ensures that schema information and physical storage locations remain tightly synchronized.

The Catalog Manager provides high-level APIs for creating databases, adding tables, and modifying schemas. When a new table is created, the Catalog Manager updates the `tables` map under the appropriate database entry, inserts the column descriptions, and persists the modified catalog back to disk. This tightly controlled update path guarantees atomicity of metadata operations and prevents inconsistencies between in-memory metadata and on-disk catalog state.

3.2 Table Level

The **Table Manager** governs the physical organization of tables on disk and provides the abstraction through which higher layers access persistent data. In RookDB, each table is stored as a dedicated file inside the directory corresponding to its parent database. This hierarchical arrangement mirrors the structure of the catalog and ensures a deterministic mapping from logical tables to physical storage locations.

Each table file is composed of two logically distinct regions. The first region is a **fixed-size table header occupying the initial 8 KB** of the file, which stores the **total number of allocated pages**. The second region consists of a **sequence of fixed-size data pages**, each 8 KB in size, which store tuple data and associated slot metadata.

The table header uses a simple format in which the **first four bytes encode the page count**. Because this header resides at a fixed location, the Table Manager can determine the number of pages in constant time without scanning the file. This predictability simplifies page allocation, file extension, and sequential scanning.

When a new table is created, the Table Manager initializes the header by writing an 8 KB block of zeroed bytes and subsequently appends the first data page, establishing the initial storage region for tuples. Each additional data page is appended contiguously to the file, supporting efficient sequential access and append-oriented workloads.

Reading or writing a page is performed by computing the **byte offset determined by the page number**. Since each page has a fixed size of 8 KB, the offset is obtained directly from the page index, enabling constant-time access to arbitrary pages without auxiliary indexing structures.

3.3 Page Level

At the lowest layer of the storage hierarchy, the **Page Manager** is responsible for the organization and manipulation of individual pages within a table file. Each page in RookDB has a fixed size of 8 KB and follows a structured, slotted-page layout that supports efficient free-space management.

Every page begins with a **compact 8-byte page header** that stores two offsets: a **lower pointer**, which marks the end of the ItemId array, and an **upper pointer**, which marks the beginning of the tuple storage region. The space between these two pointers represents the free space available for new tuples. This simple pointer-based design enables the Page Manager to compute free space using constant-time arithmetic and eliminates the need for expensive page scans.

Below the header, the page maintains a growing array of **ItemId entries**. Each entry records the offset and length of a tuple within the page, providing a stable indirection layer between logical tuple positions and their physical locations. These ItemId structures grow upward from the beginning of the page as new tuples are inserted.

Tuple data itself is stored at the opposite end of the page. When a tuple is added, the Page Manager places its byte representation immediately before the current upper pointer and then moves the pointer downward to reflect the consumed space. Because tuples grow downward and ItemIds grow upward, the page naturally organizes free space in the middle, allowing both metadata and data to expand independently until space is exhausted.

Reading and writing pages at the Page Manager level involve operating entirely on in-memory representations. When a page is loaded from disk, its header, ItemId array, and tuple region are reconstructed from the raw byte buffer. Updates—such as tuple insertion, deletion, or metadata changes—are performed directly on the in-memory page, after which the modified buffer is written back to its corresponding byte offset within the table file.

By isolating tuple placement, free-space calculation, and slot directory management within a dedicated Page Manager, RookDB maintains a clear separation between per-page data organization and the table-wide physical layout managed by higher levels. This separation simplifies the design of the storage engine and provides a robust foundation for future enhancements such as tuple deletion, page compaction, and record relocation.

At the page level, the *Page Manager* implements a slotted-page layout consisting of a page header, an array of item identifiers (ItemIds), and tuple data. The layout uses bidirectional growth: ItemIds expand upward from the beginning of the page, while tuples are inserted from the end toward the center. This design enables efficient record insertion and free-space management without requiring immediate compaction.

RookDB exposes these internal components through a set of high-level APIs that govern database creation, table creation, page allocation, and tuple insertion. Each API invocation orchestrates the underlying storage modules, ensuring consistent metadata updates and durable writes.

3.4 Buffer Manager

Above the page layer, RookDB employs a **Buffer Manager** that maintains an in-memory working set of pages to reduce disk I/O and support high-throughput data loading and manipulation. The buffer pool is initialized with a fixed number of preallocated pages, each conforming to the system's slotted-page structure. This design allows the system to reuse page frames without performing repeated memory allocations, thereby improving stability and predictability during sustained workloads.

The Buffer Manager stores two classes of information: a **copy of the table header** and a collection of **in-memory page frames** used to hold data pages. The header frame mirrors the first 8 KB of the table file and is updated as the number of data pages grows. The remaining frames serve as the in-memory staging area for page operations such as tuple insertion, free-space evaluation, and page extension.

When a table is first loaded into memory, the Buffer Manager reconstructs its structure by reading the header page and all existing data pages from disk. Any remaining frames in the pool are initialized as empty pages, ensuring that the buffer pool always maintains a fixed capacity. This layout enables the system to handle workloads in which the logical table size is smaller than the buffer allocation, while also supporting growth as additional data pages are created.

The Buffer Manager also orchestrates the process of loading external data sources, such as CSV files, directly into in-memory pages. During this process, each row is serialized according to the table schema and inserted into the first page with sufficient free space. If a page becomes full, the manager automatically advances to the next page, initializing new pages as needed. Once the data load completes, the buffer updates the header to reflect the new **page count** and writes the modified header and all used data pages back to disk. This pipeline approach ensures that large data imports can be processed efficiently while preserving the physical layout expected by the storage engine.

By serving as an intermediary between disk-resident table files and in-memory operations, the Buffer Manager provides a controlled environment for page allocation, modification, and persistence. Its design lays the groundwork for future enhancements such as replacement policies, pin–unpin semantics, concurrency control, and the integration of a full-fledged buffer pool manager.

4 Design Extensions: Extents and Variable Page Sizes

4.1 Variable Page Sizes

To evaluate how page size influences bulk-loading performance, we benchmarked RookDB using multiple page configurations ranging from 4,KB to 64,KB. Larger pages can accommodate more tuples per page, thereby reducing the total number of page allocations and disk writes during ingestion. Conversely, smaller pages increase the number of page frames required and typically result in higher I/O volume.

Page Size	Load Time(s)	Pages Used
4 KB	2.930	5408
8 KB	2.864	2689
16 KB	2.852	1346
32 KB	2.832	672
64 KB	2.783	337

Table 1: Bulk-loading performance for a 1000K-row CSV under different page sizes.

4.2 Extent-Based Allocation

An **extent** is a contiguous group of pages (e.g., 8, 16, or 32 pages) that are allocated as a single unit, rather than growing the table one page at a time. Allocating larger extents reduces the number of allocation operations and helps maintain physical locality on disk or SSD, which is especially beneficial for large sequential scans and bulk inserts. Instead of adding pages one at a time, the system grows the table by allocating the next full extent. This keeps data physically contiguous on disk, improves locality during scans, and reduces the overhead of repeated allocation calls. As data fills the current extent, the next extent is allocated and appended to the file, allowing large inserts or sequential workloads to proceed with fewer interruptions.

Extent Size (No. of Pages)	**Load Time(s)**	Extents Count
1	2.869	2869
8	2.856	338
16	2.847	170
32	2.858	86
64	2.891	44

Table 2: Bulk-loading performance for a 1000K-row CSV under different extent sizes with 8KB page size.

5 Implementation

RookDB is implemented in Rust, chosen for its strong guarantees around memory safety, predictable resource management, and efficient handling of low-level byte operations.

RookDB begins by creating an empty `catalog.json` file if none exists. This file, stored in a dedicated global metadata directory, contains the full logical description of all databases, tables, and their columns. The file is initialized as an empty JSON object with no databases, ensuring the system starts from a clean and consistent metadata state.

All physical storage is organized under a fixed root directory. When a new database is created, RookDB adds an entry to `catalog.json` and creates a corresponding directory inside the storage root. Tables are then stored as individual `.dat` files inside their database directory, giving each table its own isolated and predictable location on disk.

When a table is created, the system updates the catalog, writes the modified JSON back to disk, and initializes the table's data file with a fixed-size header that records the number of allocated pages. This tight coupling between metadata and directory layout ensures that RookDB always maintains a one-to-one relationship between logical schema entries and physical storage files, providing clarity, safety, and full namespace isolation across databases.

6 Benchmarks

To evaluate the performance characteristics of RookDB, we conducted a series of benchmarks focused on bulk-loading behavior during CSV ingestion. All experiments used a dataset of 1,000K rows containing two attributes (an integer and a fixed-length text field). The system was configured with an 8 KB page size and a single-threaded ingestion pipeline.

The primary goal of this evaluation was to measure the impact of the Buffer Manager on load performance. In the *unbuffered* configuration, every tuple insertion triggers an immediate page write to disk. In contrast, the *buffered* configuration performs all tuple operations in memory and writes pages to disk only after ingestion completes, reducing overall I/O frequency.

Both experiments were executed under identical conditions, with the only difference being whether the Buffer Manager was enabled. Load time was measured using wall-clock duration from the start of CSV ingestion to the completion of all writes.

Configuration	**Load Time (s)**
Without Buffer Manager	10.664
With Buffer Manager	2.864

Table 3: CSV ingestion performance for 1000K rows with and without the Buffer Manager.

The results indicate a substantial improvement when buffering is enabled. The buffered configuration is

3.7x faster than the unbuffered approach, a difference primarily attributable to reduced disk I/O. Because the unbuffered pipeline forces a write per tuple insertion, it amplifies I/O costs significantly. In contrast, the buffer-based design aggregates tuple operations into in-memory pages and performs only a small number of sequential writes at the end of the workload.

7 Pros and Cons of the RookDB Design

The design of RookDB offers several clear advantages rooted in simplicity, modularity, and predictable storage behavior. At the same time, the system intentionally omits certain features to maintain its focus on low-level storage mechanisms, which introduces trade-offs in flexibility, scalability, and robustness. This section summarizes the key strengths and weaknesses of the current implementation.

7.1 Pros

- **Modular, Layered Architecture:** The clear separation between the catalog, table, page, and buffer layers simplifies reasoning about each component and makes the system easy to extend.
- **Human-Readable JSON Catalog:** Using JSON for metadata provides transparency, simplifies debugging, and allows schema updates to be performed easily without specialized tooling.
- **Namespace Isolation via Directory Structure:** Mapping each database to its own directory provides a strong storage-level boundary that prevents accidental cross-database interference.
- **Fixed-Size Headers and Pages:** Predictable 8 KB page and header sizes allow constant-time offset computation and efficient random access without auxiliary indexing structures.
- **Slotted-Page Layout:** Bidirectional growth of ItemIds and tuple data provides efficient free-space management and stable tuple addressing, forming the foundation for compaction and deletion features.
- **In-Memory Buffer Manager:** The buffer layer significantly reduces disk I/O during bulk ingestion, as demonstrated by the $3.3\times$ speedup compared to unbuffered loading.

7.2 Cons

- **Missing Tuple Deletion and Compaction:** Without free-space reclamation, pages may become fragmented over time, reducing storage efficiency.
- **Single-Threaded Execution Model:** The absence of concurrency control limits throughput and prevents multi-user or parallel workloads.
- **No Durability Mechanisms:** Without write-ahead logging or crash-recovery features, the system does not guarantee persistence against failures.
- **No Buffer Replacement Policy:** The buffer manager assumes enough in-memory space to hold all active pages, which is not scalable for larger datasets.
- **Full Catalog Rewrite on Updates:** Because the entire JSON catalog is rewritten for each metadata change, performance may degrade as the number of databases and tables grows.
- **Limited Data Types and No Variable-Length Attributes:** The current schema system supports only basic types such as `int` and fixed-size `text`. However, it can be extended to support additional data types by enhancing the existing type-handling infrastructure and adding corresponding serialization.

8 Limitations and Future Work

RookDB is intentionally minimal, designed to isolate and study low-level storage behavior rather than to function as a full database system. As a result, several limitations remain. The current storage engine lacks

support for tuple deletion, page compaction, and free-space reclamation, meaning that once data is written to a page, its physical layout cannot be reorganized without rewriting the entire file. The slotted-page format provides the infrastructure for these features, but additional logic is required to manage fragmentation and reclaim unused space.

The system also omits any form of concurrency control or write-ahead logging. All operations assume a single-threaded execution model and do not provide durability guarantees in the presence of crashes. The absence of a buffer-pool replacement policy limits scalability, as the current buffer manager assumes that all active pages fit within the fixed in-memory allocation. Similarly, the catalog is re-written in its entirety for each metadata change, which is acceptable for small workloads but may become inefficient as the number of tables and databases grows.

Several promising directions exist for expanding the system. At the storage layer, future work includes implementing tuple deletion, in-page compaction, and background cleaners to restore free space. Extending the Buffer Manager with pin–unpin semantics and a replacement strategy such as LRU or CLOCK would enable experiments under memory pressure. Adding variable-length attribute support, overflow chains, and indexing structures would allow more realistic workloads to be executed. At the allocation level, adopting extent-based file growth and experimenting with different extent sizes would provide deeper insight into sequential layout and write amplification.

Beyond the storage engine, integrating a lightweight query executor, adding concurrency control mechanisms, and introducing a log-based recovery model would move RookDB closer to a fully featured DBMS while retaining its educational and experimental value. These extensions would enable richer benchmarking across metadata updates, transactional semantics, and multi-query workloads. Overall, the system provides a clean foundation on which to build more advanced storage and management features.

9 Conclusion

Building RookDB allowed us to understand how the core components of a storage engine work together and how design choices at the catalog, table, page, and buffer layers directly influence performance. Implementing features such as fixed-size pages, slotted-page layout, and an in-memory buffer clarified how real DBMSs manage metadata, organize records on disk, and reduce I/O during bulk ingestion. The benchmarks showed that buffering has a substantial impact on load time, emphasizing the importance of minimizing fine-grained disk writes. Overall, the project was highly helpful in understanding DBMS storage engine internals and provided a clear foundation for exploring more advanced techniques in future work.

A Source Code Repository

The full RookDB project is available on GitHub: https://github.com/hemanth-sunkireddy/RookDB.