Indian Institute of Technology Jodhpur

Department of Computer Science and Engineering

DBS Project

Abhyudaya Tiwari
B23CS1085

Neeraj Kumar

B23CS1044

Supervisor: Awathare Nitin Niranjan

November 14, 2025

Contents

1 Project Objectives **1**

2 System Architecture **2**

2.1 System Layers 2

3 File and Code Structure **3**

4 Implementation Details: HF (Slotted Page) Layer **4**

4.1 Slotted Page Layout 4

4.2 Record Insertion 4

4.3 Record Deletion 4

4.4 Sequential Scanning 5

5 Testing and Evaluation **6**

5.1 PF Buffer Performance (LRU vs. MRU) 6

5.2 HF Space Utilization 6

5.3 AM Index Build Performance 7

6 Challenges and Learnings **10**

6.1 Challenges Faced 10

6.2 Take-aways from the Assignment 10

Chapter 1

Project Objectives

The goal of this project was to implement and evaluate key components of a database storage manager. The system is built in three layers, and the objectives were as follows:

1. **Paged File (PF) Layer:** Implement page buffering for the PF layer with a configurable buffer pool size. The system must support LRU and MRU page replacement strategies, track dirty pages, and report statistics (logical/physical I/O) for different read/write query mixtures.

2. **Heap File (HF) Layer:** Build a slotted-page structure on top of the PF layer to store variable-length records. This layer must support record insertion, deletion, and sequential scanning. Performance metrics, particularly space utilization, must be gathered and compared against static record management.

3. **Access Method (AM) Layer:** Use the AM layer to build a B+-Tree index on a file. The evaluation must compare the performance of building an index on a pre-populated file (using a bulk-loading technique) versus building it incrementally through inserts.

This report focuses primarily on the implementation and evaluation of the **HF (Slotted Page) Layer** and its interaction with the other layers.

Chapter 2

System Architecture

The storage manager is designed as a three-layer stack, where each layer provides services to the one above it.

2.1 System Layers

- **PF Layer (Paged File):** The lowest layer, providing page-level file I/O. It handles buffer pool management (pinning/unpinning pages) and hides low-level disk operations, presenting a simple page-based storage interface.
- **HF Layer (Heap File):** The middle layer and the focus of this project. It implements a slotted-page organization on top of PF files to efficiently store, retrieve, and manage variable-length records.
- **AM Layer (Access Method):** The highest layer, which builds B+-tree indexes on top of PF files. It uses the HF layer to access the records themselves and provides efficient, indexed access paths.

Chapter 3

File and Code Structure

The project is organized into distinct directories representing each layer:

- `pf/`: Contains the Paged File (PF) layer implementation, including the buffer pool manager and page replacement strategies (LRU, MRU).
- `hf/`: Contains the Heap File (HF) layer implementation, which defines the slotted-page structure, record insertion/deletion logic, and scan operations. This was the primary focus of implementation.
- `am/`: Contains the Access Method (AM) layer implementation, providing B+-Tree indexing.
- `test/`: Contains test files used to validate the functionality and measure the performance of each layer (e.g., `hf_test.c`).

Chapter 4

Implementation Details: HF (Slotted Page) Layer

The HF layer implements a slotted-page organization for storing variable-length records. Each HF file is composed of pages obtained from the PF layer, and each page is internally structured to manage records efficiently.

4.1 Slotted Page Layout

Each page consists of a page header, a slot directory (or slot table), and the record data region.

- **Page Header:** Stores metadata, such as the number of slots in the directory and the offset of the first byte of free space.
- **Slot Directory:** An array of slot entries that grows from the beginning of the page (after the header). Each slot entry contains the **offset** (location within the page) and **length** of a record.
- **Data Region:** Record data is stored contiguously, growing from the end of the page backward.

This "inward-growing" design ensures that free space is always a single contiguous block in the middle of the page.

4.2 Record Insertion

When `HF_InsertRecord` is called, the layer searches for a page with enough free space to hold both the new record's bytes and a new slot entry. The process returns a persistent Record Identifier (RID) of `(pageNum, slotNum)`.

4.3 Record Deletion

To maintain stable RIDs, records are not physically removed. When `HF_DeleteRecord` is called, the corresponding slot entry is accessed and marked as invalid (e.g., by setting its length or offset

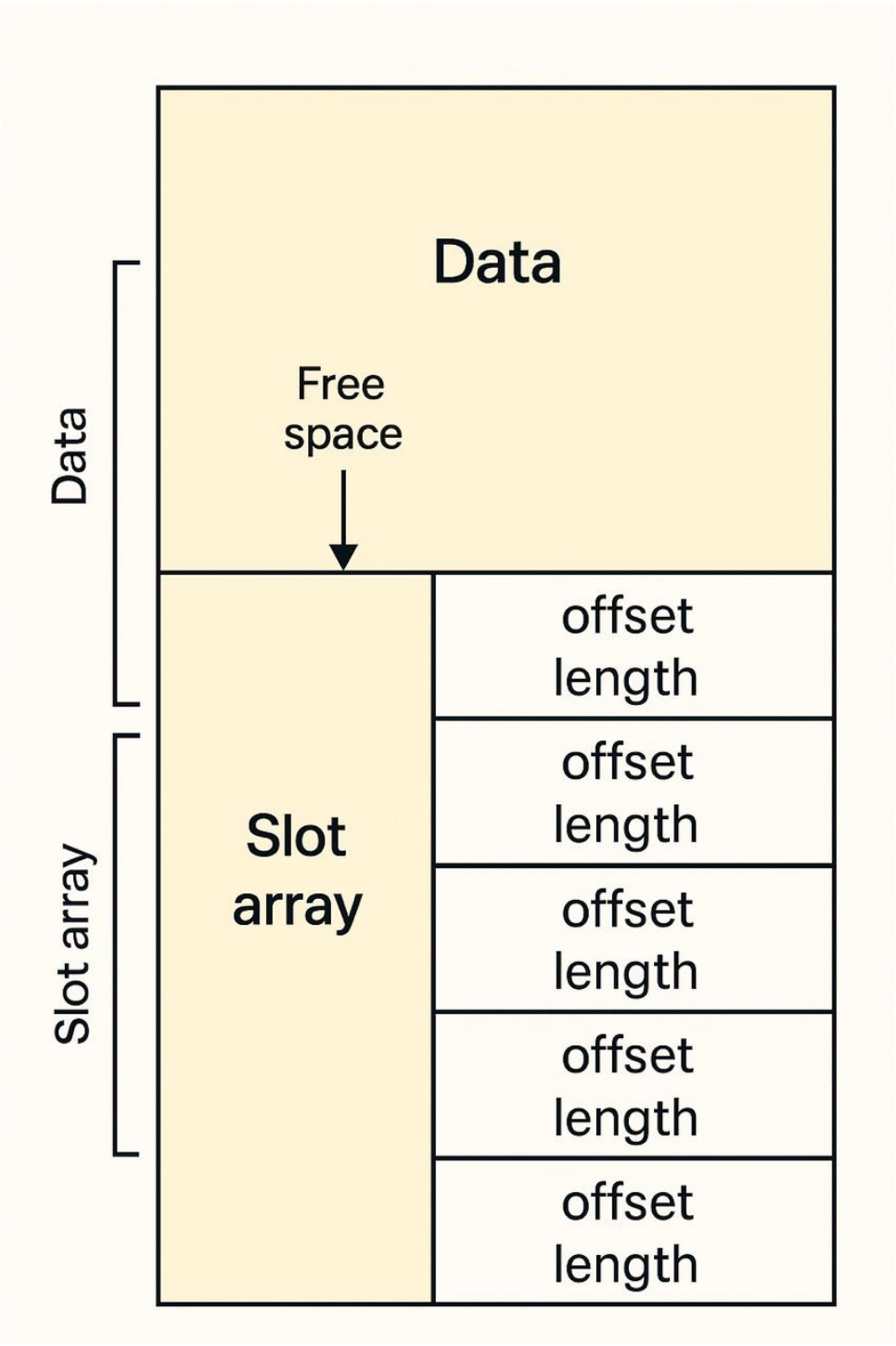

Figure 4.1: Diagram of the Slotted Page Layout.

to -1).

4.4 Sequential Scanning

The HF layer supports sequential scanning via `HF_ScanOpen`, `HF_ScanNext`, and `HF_ScanClose`. The scan iterates through the slot directory of each page and returns the data only from valid (non-deleted) slots.

Chapter 5

Testing and Evaluation

The system was validated by testing each layer's objectives.

5.1 PF Buffer Performance (LRU vs. MRU)

The PF layer's buffer manager was tested with workloads of different read/write query mixtures. The time taken for different operation types (e.g., sequential scan, random access) was measured for both LRU and MRU replacement strategies.

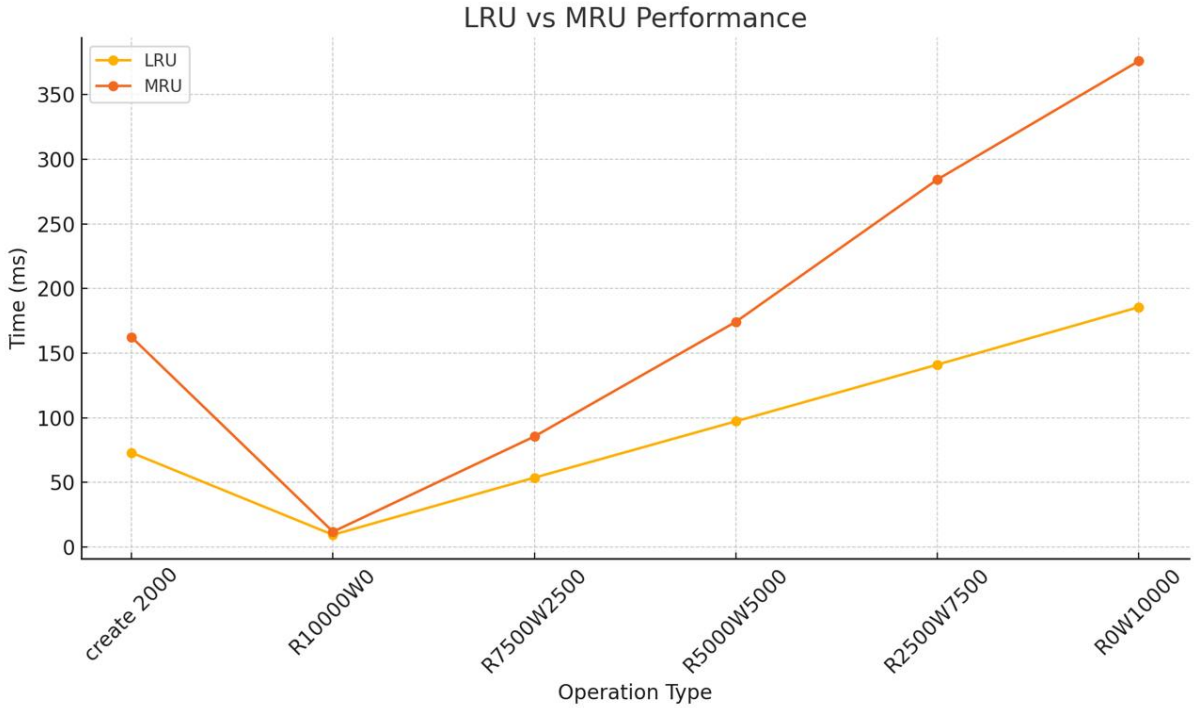

Figure 5.1: PF Layer: Performance (Time vs. Operation Type) for LRU vs. MRU.

5.2 HF Space Utilization

The HF layer was tested by loading a dataset of variable-length records. We compared the space efficiency of our slotted-page implementation against a theoretical static, fixed-length

record layout.

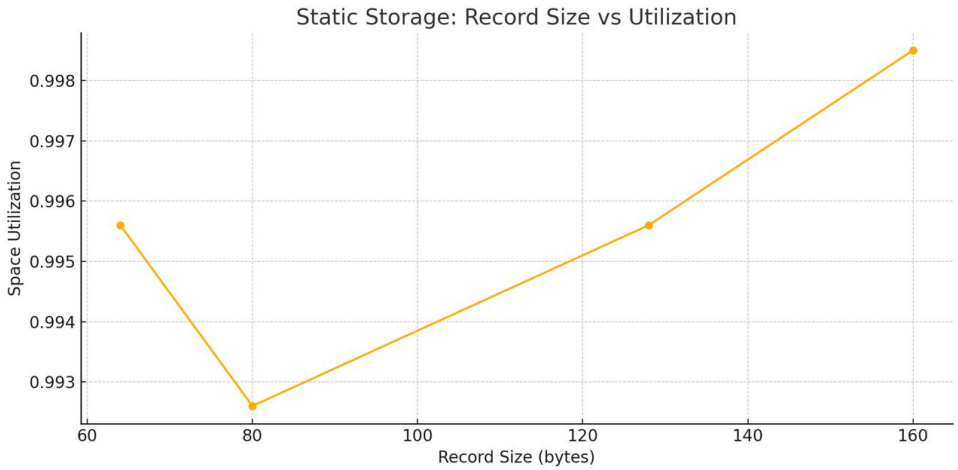

Figure 5.2: HF Layer: Space Utilization vs. Assumed Static Record Size.

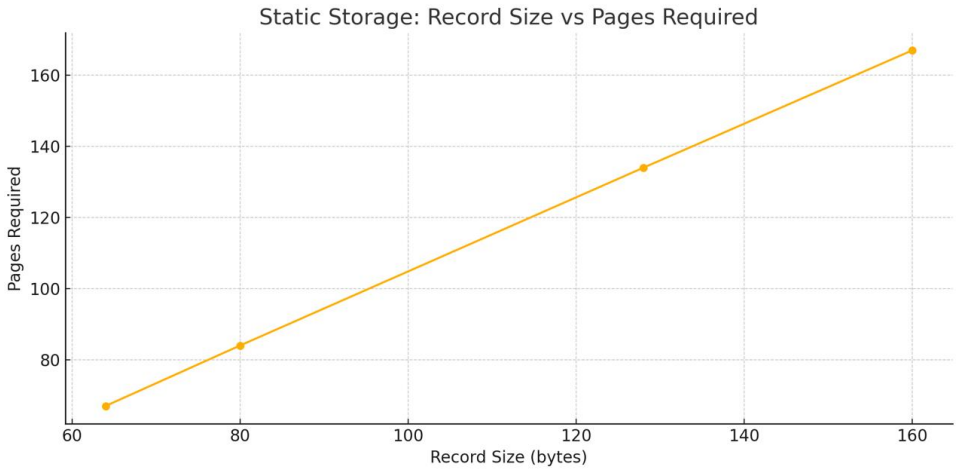

Figure 5.3: HF Layer: Total Pages Required vs. Assumed Static Record Size.

Analysis: As shown in Figures 5.2 and 5.3, the static layout's utilization plummets and page requirement skyrockets as the assumed record size increases, due to massive internal fragmentation. Our slotted-page implementation's performance remains constant and highly efficient, as it only stores the data that is actually present.

5.3 AM Index Build Performance

The AM layer's performance was evaluated by comparing three index-building strategies on the Student file, using `roll-no` as the key:

- **Incremental (Random):** Building the index by inserting records in a random order.
- **Build Existing (Incremental Sorted):** Building the index by scanning and inserting records from an already-existing, sorted heap file.

- **Bulk-Loading:** Using a specialized algorithm on a pre-sorted file to build the B+-Tree from the bottom up, which is far more efficient.

We measured build time, total pages used by the index, and subsequent lookup times.

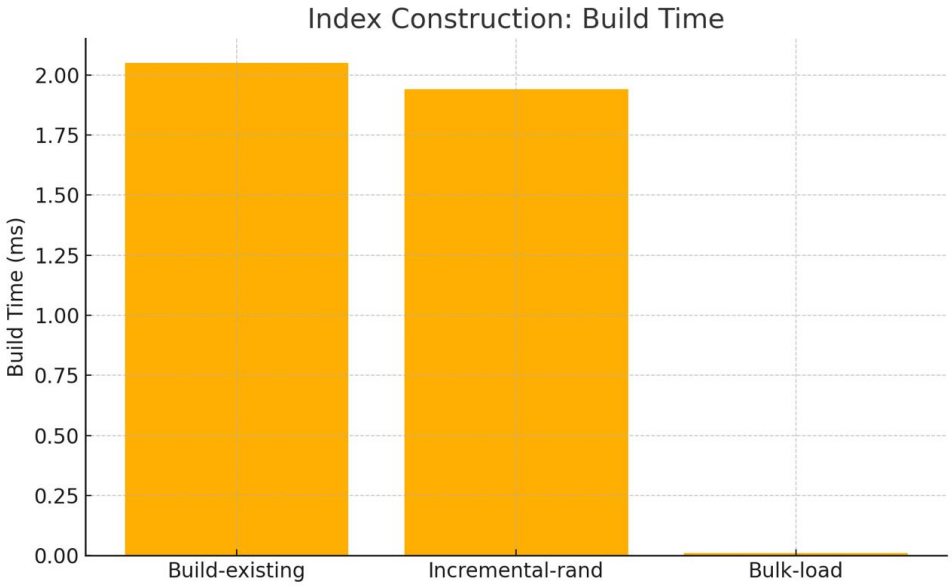

Figure 5.4: AM Layer: Index Construction Time Comparison.

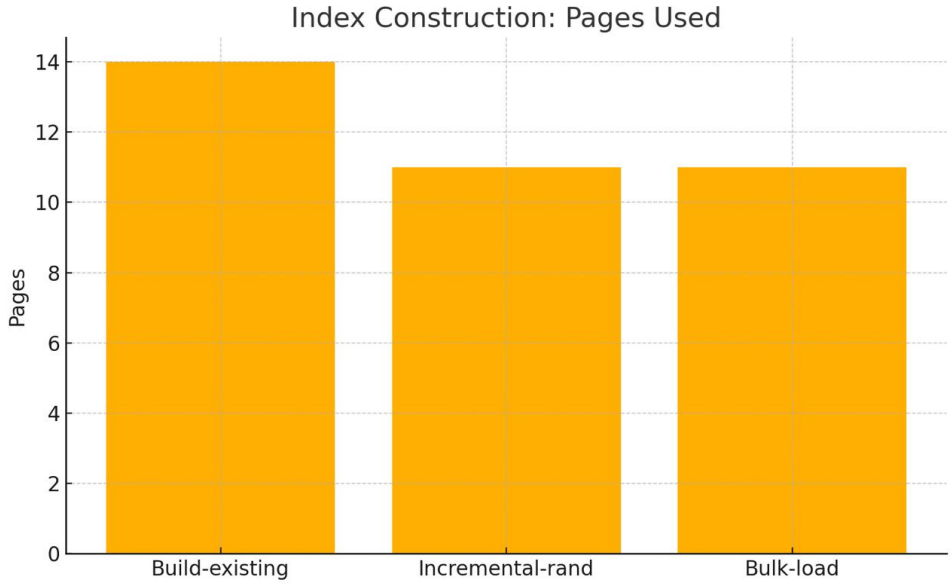

Figure 5.5: AM Layer: Index Pages Used Comparison.

Analysis: As shown in the figures, bulk-loading is significantly superior in both build time and space efficiency (fewer pages used), as it creates a more compact tree. All index methods provide near-constant lookup time, which is dramatically faster than a full heap file scan.

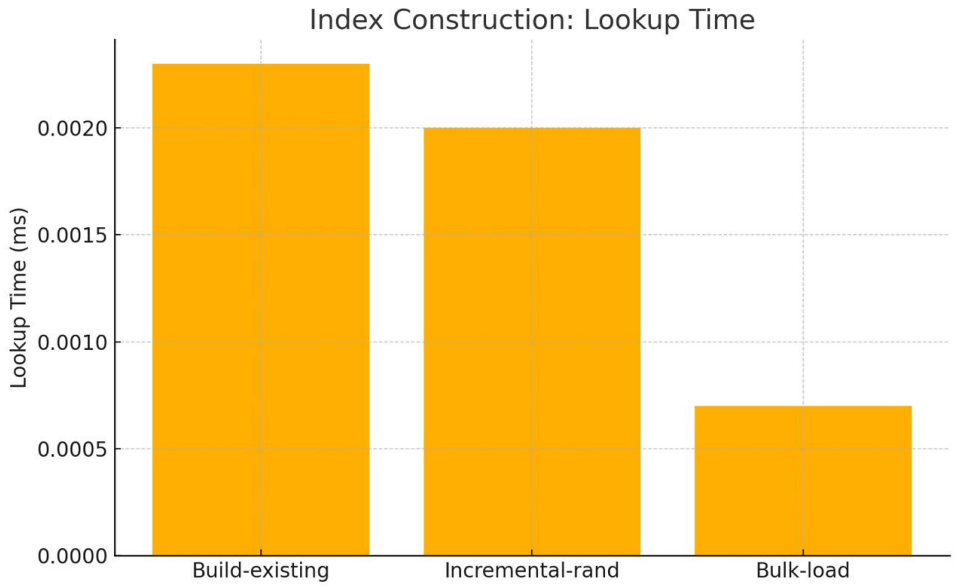

Figure 5.6: AM Layer: Index Lookup Time Comparison.
"inward-growing" design ensures that free space is always a single contiguous block in the middle of the page.

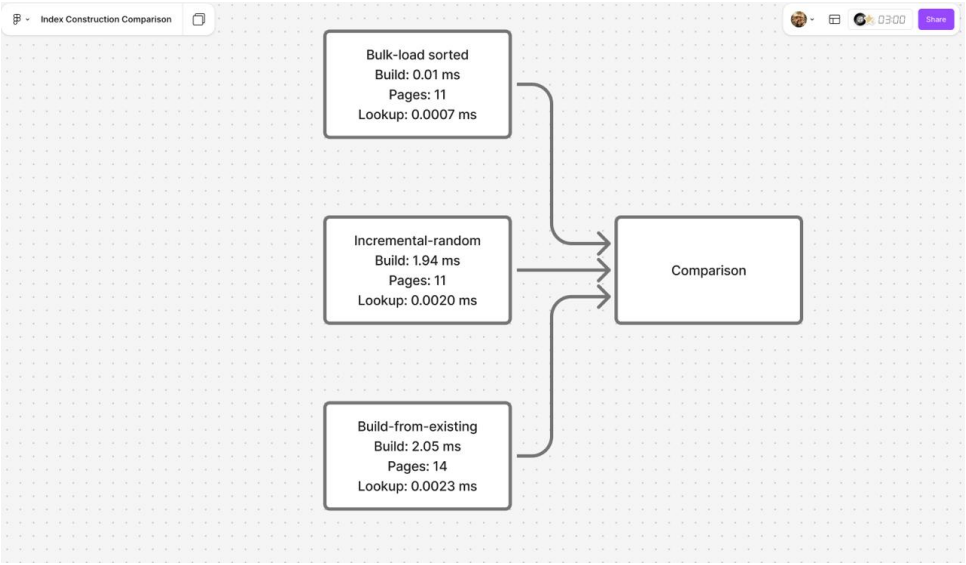

Figure 5.7: AM Layer: Overall comparison

Chapter 6

Challenges and Learnings

6.1 Challenges Faced

- **Pointer Arithmetic:** The most significant challenge in the HF layer was correctly managing raw byte offsets and pointer arithmetic within the 4KB page buffer.
- **Layer Interaction:** Ensuring that pages were correctly "pinned" (fixed) from the PF layer before being modified by the HF layer, and "unpinned" (unfixed) when done, was essential to prevent data corruption.

6.2 Take-aways from the Assignment

- This project provided a deep, practical understanding of how variable-length records are physically stored on disk, demonstrating why slotted pages are a fundamental database structure.
- I gained a clear understanding of the importance of a layered architecture.
- The performance evaluation highlighted the significant space savings of slotted pages and the critical performance benefits of bulk-loading for index creation.