NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC3020/CZ4031 DATABASE SYSTEM PRINCIPLES

College of Computing and Data Science

Academic Year 2025/2026 Semester 1

Project 1

Name	Matriculation No.	Individual Contribution
Sky Lim En Xing	U2223731A	Task 1
Gupta Raghav	U2222889A	Task 2 and 3
Mythili Mulani	U2222678G	Task 2 and 3
Minhwan Kim	U2022178H	Task 2 and 3
Chew Wei Hao Kovan	U2322227A	Task 2 and 3

Introduction

Video Presentation : SC3020 Project 1 Video.mp4
Source Code: SC3020 Project 1 Source Code

In this project, our group designs and implements the storage and indexing components of a database management system. This project has been implemented in C.

Our report is structured as follows:

1. Storage component
   In this part, we illustrate the design of our storage component, where data is stored in blocks on the computer disks

2. Indexing Component
   In this part, we describe the implementation of how B+ tree is used and stored on disk, whereas building of B+ tree is done through the method of iteratively inserting records/bulk loading.

3. Deletion Component

   In this part, we describe the implementation of how B+ Tree Search is performed for range deletion of all records where FT_PCT_home > threshold. The system compares B+ Tree search performance against linear scan and provides detailed statistics.

Storage Component

This section illustrates the design of the storage components, each component in our design is listed and explained, alongside with some figures to further illustrate how each part of these storage components interacts with each other.

Data Components

The data components consists of the following objects that illustrates what is being stored:

- Records

  We define a fixed-width packed record format for storing data on disk, and each field is assigned a fixed number of bytes according to its type:
  - INT32: stored as 4 bytes
  - FLOAT: stored as 4 bytes
  - CHAR[10]: stored as 10 bytes (fixed-length string)

  The entire record is packed into a 27-byte buffer for efficient storage and retrieval.

- Block

  Disk operates in blocks(pages), thus block size is fixed to 4096 bytes to simulate real OS pages size. Each block has header (4 byte) + record array (consecutive 27 byte records)

  This unspanned organization design restricts records from crossing block boundaries, allowing simpler access, safer writes, simplifying stores management and allows access by blockID. Used_count in the block header tells us exactly how many valid slots are there.

- Heap File (Database file)

  Sequence of blocks stored in a binary file (data.db) and new blocks appended sequentially during data loading.

  This part puts together records, block, file manager, and buffer pool; it stores the sequence of blocks into a binary file (data.db).

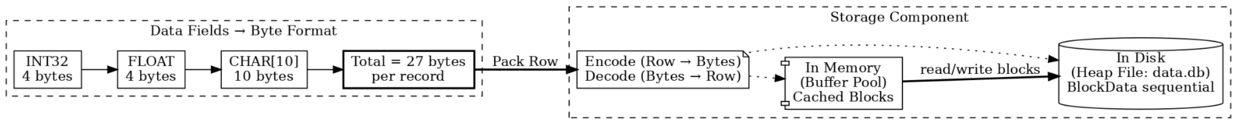

Data Fields → Byte Format
INT32 4 bytes
FLOAT 4 bytes
CHAR[10] 10 bytes
Total = 27 bytes per record
Pack Row
Storage Component
Encode (Row → Bytes) Decode (Bytes → Row)
In Memory (Buffer Pool) Cached Blocks
read/write blocks
In Disk (Heap File: data.db) BlockData sequential

Figure 1: Storage Components Design

In summary, the data component defines the fundamental storage unit of our database: records, blocks, and the heap file. Figure 1 illustrates how fixed-width 27-byte records are encoded into blocks, cached in the buffer pool, and stored sequentially in the heap file (data.db). This predictable structure ensures efficient access, simple slot computation, and reliable disk storage.

<u>Implementation details (for Data Components)</u>:

File	Functions and descriptions
block.c	- **block_capacity_records(record_size)**: Calculates how many records fit into a block, given record size and fixed block size (4096 bytes). - **block_write_record(b, record_size, slot, rec)**: Writes a record into a block at the given slot. - **block_read_record(b, record_size, slot, out)**: Reads a record from a block slot into memory.
schema.c	- **schema_init_default(s)**: Initializes the schema with default fields and computes record size. - **schema_print(s)**: Prints schema details (field names, types, widths, record size). - **encode_row(s, r, dst):** Encodes a row struct into a fixed-width packed record. - **decode_row(s, src, r)**: Decodes a packed record into a row struct. - **parse_header_map(header_line_raw, idx)**: Parses CSV header, normalizes names, and maps fields to schema indexes. - **parse_row_by_index(line_raw, idx, out)**: Parses one CSV row into a row struct based on the header index.
file_manager.c	- **fm_open(fm, path, mode)**: Opens a database file for reading/writing.

	- **fm_close(fm)**: Closes the database file. - **fm_read_block(fm, block_id, out)**: Reads a block from disk into memory. - **fm_write_block(fm, block_id, in)**: Writes a block from memory to disk. - **fm_alloc_block(fm, zeroed)**: Allocates and appends a new zeroed block at the end of the file.
heapfile.c	- **hf_create(hf, path, schema, buf_frames)**: Creates a new heap file with given schema and buffer pool size. - **hf_open(hf, path, buf_frames)**: Opens an existing heap file and initializes schema + buffer pool. - **hf_close(hf)**: Flushes buffer pool and closes the heap file. - **hf_records_per_block(hf)**: Returns how many records can fit per block for this schema. - **hf_count_records(hf)**: Counts total number of records stored in the heap file. - **hf_print_stats(hf)**: Prints statistics such as block size, records per block, total blocks, total records, and I/O counts. - **hf_load_csv(hf, csv_path)**: Loads rows from CSV, encodes them into records, and writes them into blocks sequentially. - **hf_scan_print_firstN(hf, limit)**: Scans the heap file and prints the firstN decoded records. - **hf_delete_record(hf, block_id, slot_id):** fetches the block where the record has to be deleted with bp_fetch() based on its block_id and slot_id and loops through the block and replace the current position where the record (to be deleted) with records after it. (deletion)

Controller Components

The controller component consists of the following objects that illustrates how data flows into storage:

- File Manager
  This handles read/write between disk and memory

- Database manager
  Provides following operations:
    - Load Module (load)
      A CSV parser function reads each row from games.txt and maps file headers to schema fields. Each row is then converted into a fixed-width record (27 bytes) using the encode function (Row struct → byte array). These records are packed into blocks, and written to the heap file when a block is full. The process continues until all records are stored.
    - Statistics Module (stats)
      Traverses heap file using buffer pool to calculate and output: record size, total number of records (from dataset), records per block (block size / record size), total number of blocks written.
    - Scan Module (scan)
      Sequentially reads blocks and iterates through records inside each block of the database file.

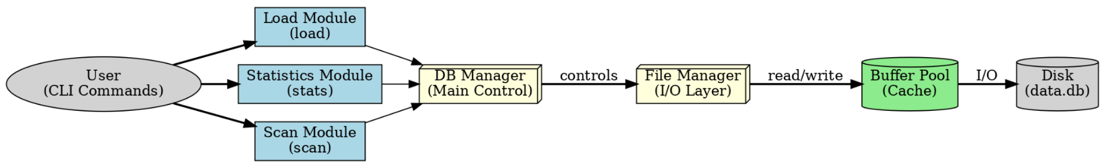


Figure 2: Controller Components Workflow

The controller component acts as the bridge between user operations and physical storage. Figure 2 illustrates how the file manager provides low-level read/write to disk, while the database manager exposes high-level commands (load, stats, scan) that process CSV files, manage heap files, and output useful statistics. Together, these modules allow users to interact with the database system in a straightforward way, while abstracting away the complexity of block management.

<u>Implementation details (for Controller Components)</u>:

File	**Functions and descriptions**

cli.c	- **usage( )**: Prints usage instructions for the command-line interface. - **run_cli(argc, argv)**: Parses command-line arguments and runs corresponding operations (load, stats, scan).
heapfile.c	- **hf_print_stats(hf)**: Prints statistics such as block size, records per block, total blocks, total records, and I/O counts. - **hf_load_csv(hf, csv_path)**: Loads rows from CSV, encodes them into records, and writes them into blocks sequentially. - **hf_scan_print_firstN(hf, limit)**: Scans the heap file and prints the firstN decoded records.
main.c	- **main(argc, argv)**: Entry point; delegates execution to run_cli().

Buffer Pool

The buffer pool acts as a cache between disk and memory, reducing the number of expensive disk I/Os by holding frequently used blocks in memory.

- Buffer Manager

  Maintains a fixed-size in-memory array of blocks, each holding one block.

  On bp_fetch(block_id):
  - If the block is already cached → return it (cache hit).
  - If the block is not cached and an empty frame exists → use it.
  - If the block is not cached and no empty frames → evict a block using LRU.

  On eviction:
  - If the block is dirty, flush it to disk.
  - Otherwise, discard.

- Replacement Policy
  If the buffer is full, it decides which block to evict using the Least recently used policy.

To conclude, the buffer pool provides the efficiency layer of our storage engine. By caching a limited number of blocks in memory and applying the Least Recently Used (LRU) replacement policy, it reduces unnecessary disk I/O and improves performance. Although the logical content of records, blocks, and heap files remains the same, buffer size (--buf N) directly influences the number of reads and writes required. This component highlights the trade-off between memory size and I/O efficiency in database systems.

<u>Implementation details (for Buffer Pool)</u>:

File	**Functions and descriptions**
buffer_pool.c	- **bp_init(bp, fm, capacity)**: Initializes the buffer pool with given capacity and links it to the file manager. - **bp_destroy(bp)**: Flushes all dirty blocks and frees buffer pool memory. - **bp_fetch(bp, block_id)**: Fetches a block into memory; returns from cache if present, otherwise loads from disk (evicts LRU if full). - **bp_mark_dirty(bp, block_id)**: Marks a block as dirty so it will be flushed to disk later. - **bp_flush_all(bp)**: Writes all dirty blocks in the buffer pool back to disk.

Compilation and Execution Instructions for Task 1

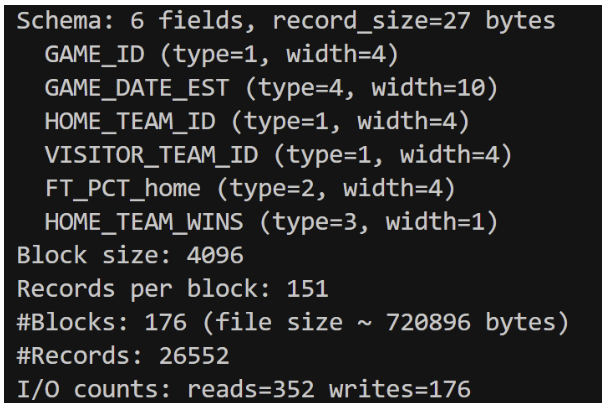
1. First, we compile all the source files using GCC:

   gcc -std=c11 -O2 -Iheader src/*.c -o project_c

   This generates the executable project_c.

2. Next, we run the load command to read data from games.txt, encode each record, and store them sequentially into our binary heap file data.db, using a buffer size of 64:

   ./project_c load games.txt data.db --buf 64



```
Schema: 6 fields, record_size=27 bytes
  GAME_ID (type=1, width=4)
  GAME_DATE_EST (type=4, width=10)
  HOME_TEAM_ID (type=1, width=4)
  VISITOR_TEAM_ID (type=1, width=4)
  FT_PCT_home (type=2, width=4)
  HOME_TEAM_WINS (type=3, width=1)
Block size: 4096
Records per block: 151
#Blocks: 176 (file size ~ 720896 bytes)
#Records: 26552
I/O counts: reads=352 writes=176
```

<u>I/O counts:</u>

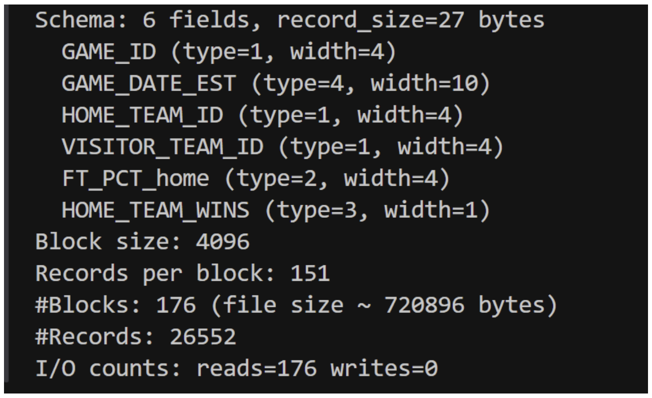
Reads = 352 → Reading from the CSV and managing buffer pool loads.

Writes = 176 → Writing 176 blocks into the heap file on disk.

This command **creates** the database file and stores all the records.

3. After loading, we run the stats command to display key statistics of the heap file — such as record size, records per block, total blocks, and I/O counts:

   ./project_c stats data.db --buf 64



```
Schema: 6 fields, record_size=27 bytes
  GAME_ID (type=1, width=4)
  GAME_DATE_EST (type=4, width=10)
  HOME_TEAM_ID (type=1, width=4)
  VISITOR_TEAM_ID (type=1, width=4)
  FT_PCT_home (type=2, width=4)
  HOME_TEAM_WINS (type=3, width=1)
Block size: 4096
Records per block: 151
#Blocks: 176 (file size ~ 720896 bytes)
#Records: 26552
I/O counts: reads=176 writes=0
```

I/O counts:

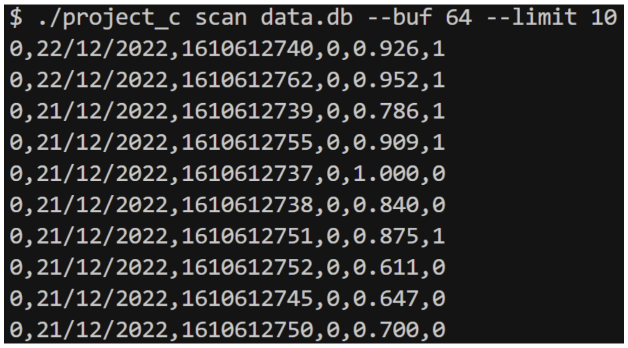
Reads = 176 → One read per block to gather stats.

Writes = 0 → No data modification, so nothing written back.

This only reads data from the heap file to compute statistics, no new data is written.

4. Finally, we use the scan command to view the first few records stored in data.db:

   ./project_c scan data.db --buf 64 --limit 10



```
$ ./project_c scan data.db --buf 64 --limit 10
0,22/12/2022,1610612740,0,0.926,1
0,22/12/2022,1610612762,0,0.952,1
0,21/12/2022,1610612739,0,0.786,1
0,21/12/2022,1610612755,0,0.909,1
0,21/12/2022,1610612737,0,1.000,0
0,21/12/2022,1610612738,0,0.840,0
0,21/12/2022,1610612751,0,0.875,1
0,21/12/2022,1610612752,0,0.611,0
0,21/12/2022,1610612745,0,0.647,0
0,21/12/2022,1610612750,0,0.700,0
```

This demonstrates that our storage component can successfully load, store, and retrieve data from disk using the buffer pool.

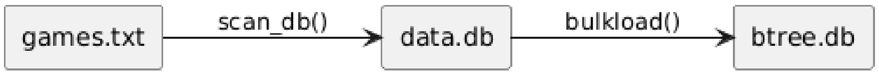
B+ Component

Design Goals

We implement an on-disk B+ tree over attribute **FT_PCT_home** to support range queries and selective deletion. The builder operates **bottom-up** in two phases:

1. **Heap scan & ordering** – extract (key, block_id, slot_id) tuples from the heap file and **totally order** them by key (with stable tiebreakers).
2. **Page materialization** – pack sorted tuples into **leaf pages**, persist them, then iteratively **bulk-load internal levels until** a single root page remains.

This approach yields a reproducible tree (deterministic structure given the same dataset and page constants) and minimizes random I/O during construction by writing pages sequentially.



Node & Page Layout

Each B+ tree page is a fixed-size **4 KB** node with a compact header and a contiguous payload region.

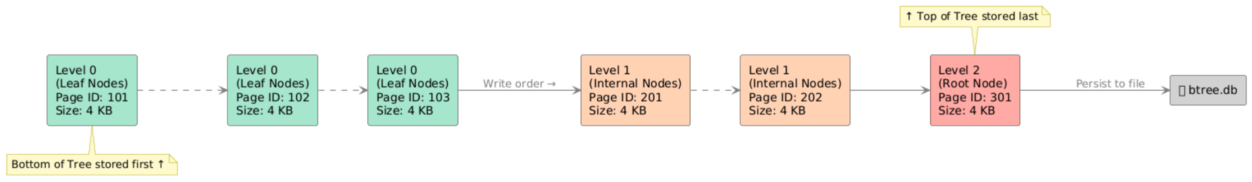
- **Header (11 B):**
  level (1B) | node_id (4B) | key_count (2B) | lower_bound (4B)
- **Leaf payload layout (repeating):**
  [record_pointer (8B : block_id(4B), slot_id(4B))] + [key (4B)]
  The **last 4 bytes** of the payload store the **next-leaf node_id** (for right-sibling chaining).
- **Internal payload layout:**
  P0 (8B) followed by repeating [key (4B), Pi (8B)] entries.
- **Fan-out macros:**
  MAX_LEAF_KEYS = (((NODE_SIZE - NODE_HDR_SIZE) / (KEY_SIZE + RECORD_POINTER_SIZE)) -
  MAX_INTERNAL_KEYS = (((NODE_SIZE - NODE_HDR_SIZE) / (KEY_SIZE + NODE_POINTER_SIZE)) -
  MIN_LEAF_KEYS   = (MAX_LEAF_KEYS + 1) / 2
  MIN_INTERNAL_KEYS = MAX_INTERNAL_KEYS / 2
  MAX_INT_CHILDREN = MAX_INTERNAL_KEYS + 1
- **Endianness & encoding:** nodes are serialized/deserialized via encode_node / decode_node using **little-endian** copies to/from the payload region, ensuring a byte-accurate disk image.

Key routines (bptreenode.c / bptree.h):
node_init, node_write_record_key, link_leaf_node, node_write_node_key, encode_node, decode_node.

File Manager & On-Disk Organization

BtreeFileManager abstracts page allocation and I/O:

- btfm_alloc_node appends a **zero-initialized page** and returns its node_id (page number).
- btfm_write_node seeks to node_id * NODE_SIZE and persists the encoded node.
- Pages are written **by level** during build: **all leaves (level 0) first**, followed by their parents (level 1), and so on until the root. This layout matches the bottom-up pipeline and supports simple sequential verification and efficient top-down reads post-build.


↑ Top of Tree stored last
Level 0 (Leaf Nodes) Page ID: 101 Size: 4 KB
Level 0 (Leaf Nodes) Page ID: 102 Size: 4 KB
Level 0 (Leaf Nodes) Page ID: 103 Size: 4 KB
Write order →
Level 1 (Internal Nodes) Page ID: 201 Size: 4 KB
Level 1 (Internal Nodes) Page ID: 202 Size: 4 KB
Level 2 (Root Node) Page ID: 301 Size: 4 KB
Persist to file
btree.db
Bottom of Tree stored first ↑

Phase 1: Heap Scan & Stable Ordering

Entry point: scan_db(HeapFile *hf)

1. **Scan through buffer pool.** For each block b in data.db, fetch via bp_fetch, iterate used slots, and decode_row into a Row struct.
2. **Staging tuples.** Append (key = r.ft_pct_home, block_id = b, slot_id = s) into a dynamically grown array (ensure_capacitydoubles capacity).
3. **Total order.** After the full pass, execute qsort(entries, …, compare_key_pointer), which orders by (key, block_id, slot_id). This ensures a **stable ascending key order** with deterministic tie-breaks, a prerequisite for balanced page packing.

Phase 2: Leaf Construction (Bottom Level)

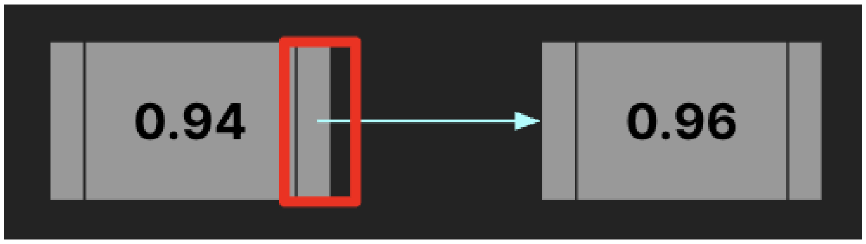
Within scan_db:

1. **Initialize first leaf.** Allocate a new page via btfm_alloc_node, then node_init(curr, /*level*/ 1, node_id) for leaves (in this codebase, level uses 1 for leaves; internally we treat increasing integers as upper levels).
2. **Fill until full.** For each sorted entry, call node_write_record_key(curr, key, block_id, slot_id).
    - On the **first** insert, lower_bound is set to the leaf's first key.
    - If the call returns error (leaf full), finalize the leaf:
        - Record its lower_bound into a side array,

        - Allocate new_id for the **next** leaf,
        - link_leaf_node(curr, new_id) to chain siblings,
        - btfm_write_node to persist,
        - node_init(next, 1, new_id) and **retry** the current tuple on the fresh page.
3. **Close the run.** For the final partially filled leaf, record its lower_bound, set right-sibling to UINT32_MAX, and btfm_write_node to persist.

Products of this phase:

- A sequential run of **leaf pages** on disk, each with a lower_bound and a valid right-sibling pointer.
- An array lower_bound_array[] listing the first key of every leaf in **leaf order**; this becomes the input to internal bulk loading.



Phase 3: Bulk-Loading Internal Levels

Entry points: bulkload and pack_internals (bottom-up)

Inputs

- child_list: for the first internal level, populated from lower_bound_array[] where each element is (key = leaf.lower_bound, node_id = leaf_page_id).
- MAX_INT_CHILDREN and MIN_INTERNAL_KEYS determine grouping and underflow handling.

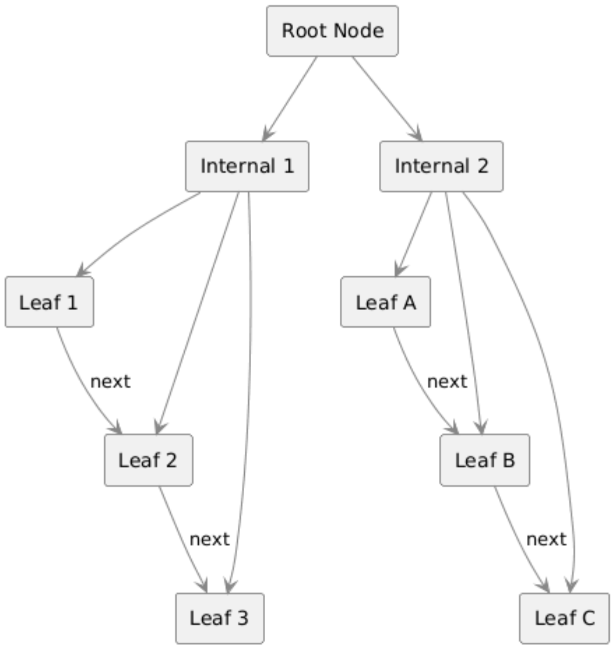
Algorithm (per level)

1. **Partition children into parents.** pack_internals(child_list, node_count, level, total_nodes, &parent_count, parent_list, fm) produces **parent nodes** at the next higher level by writing separator keys (first_key of each child ≥ 1) and child pointers in internal-page format:
    - If node_count < MAX_INT_CHILDREN: **single parent** – write one internal node containing all children (root construction case).
    - If node_count % MAX_INT_CHILDREN == 0: **exact groups** – create num_nodes = node_count / MAX_INT_CHILDRENinternal pages, each consuming a contiguous run of children.

   - **Borrowing case:** if the remainder would make the **final parent underflow** ((node_count % MAX_INT_CHILDREN) - 1 < MIN_INTERNAL_KEYS), redistribute from the penultimate group to ensure both last two parents satisfy MIN_INTERNAL_KEYS. The code computes num_borrow and borrow_from to split across the boundary, then writes two well-formed parents.
2. **Write and promote.** Each created parent is btfm_write_node'd immediately and also summarized into a **new**parent_list of (lower_bound, node_id) pairs for the next iteration upward.
3. **Repeat** until parent_count == 1; the last produced node is the **root**.

Correctness notes:

- **First-key separators**: for each parent, we store P0 once (the pointer before the first key), then (key_i, P_i) for i = 1..k. Keys are taken as the **first keys of children 1..k** (child 0 is covered by P0).
- **Balance guarantees**: the grouping logic plus the borrowing path ensures each internal node respects MIN_INTERNAL_KEYS (except possibly the root, which can be smaller), maintaining B+ tree invariants.



Diagnostics & Parameters

At the end of scan_db, we print:

- **n (fan-out parameter):** reported as MAX_LEAF_KEYS + 1 (records per leaf + 1) for readability in the assignment's terms.

- **Total leaf nodes** and **total nodes (incl. root)** as counted during construction.
- **Number of levels** (in this implementation, two or more depending on dataset size and page constants).

These values verify both the storage macros and the builder. They also facilitate comparisons with linear scan and with deletion operations later in the report.

Implementation Details (Key Functions)

Node & encoding (bptreenode.c / bptree.h)

- node_init(Node *n, uint8_t level, uint32_t node_id) — zeroes payload, sets header, sentinel lower_bound = -1.0f.
- node_write_record_key(Node *n, float key, uint32_t block_id, int slot) — appends [ptr, key], sets lower_bound for first insert, increments key_count.
- link_leaf_node(Node *n, uint32_t next_node_id) — writes right-sibling id into the last 4B of payload.
- node_write_node_key(Node *n, float key, uint32_t child_node_id) — for internals, emits P0 on first call, then appends (key, P_i).
- encode_node / decode_node — serialize/parse headers and payload bytes.

Builder orchestration (build_bplus.c / build_bplus.h)

- scan_db(HeapFile *hf) — extracts/sorts tuples, builds and persists leaves, then calls bulkload.
- bulkload(float *lower_bound_array, int child_count, BtreeFileManager *fm, int *total_nodes) — iteratively ascends levels using pack_internals.
- pack_internals(...) — creates internal nodes from child_list, handling **exact grouping** and **borrowing** underflow fix-ups; writes each node through btfm_write_node.

Persistence (file_manager_btree.c / .h)

- btfm_open/close/sync — manage the btree.db handle.
- btfm_alloc_node — append new 4 KB page, returning its node_id.
- btfm_write_node / btfm_read_node — page-aligned I/O based on node_id.

Summary

Our B+ tree builder performs a **bottom-up**, disk-friendly construction: it **sorts** the heap's key space once, **materializes leaves** as full 4 KB pages with sibling links, and then **bulk-loads** the internal hierarchy using **first-key separators** with explicit handling for boundary underflows. Pages are stored **in level**

order (leaves → parents → root), enabling simple verification and efficient traversals. This design, combined with strict page macros and deterministic ordering, yields a robust on-disk index compatible with the subsequent **range search** and **deletion** tasks in this project.

Execution Instructions for Task 2

1. This command creates the bplus tree and reports the following statics :
   the parameter n of the B+ tree; the number of nodes of the B+ tree; the number of levels of the B+ tree; thecontent of the root node (only the keys)

   ./project_c build_bplus data.db

Output :

‘N’	340	
Number of nodes	80	
Levels	2	
Content of root node	key index 1: value : 0.50 key index 2: value : 0.55 key index 3: value : 0.57 key index 4: value : 0.59 key index 5: value : 0.60 key index 6: value : 0.61 key index 7: value : 0.62 key index 8: value : 0.63 key index 9: value : 0.64 key index 10: value : 0.65 key index 11: value : 0.65 key index 12: value : 0.66 key index 13: value : 0.67 key index 14: value : 0.67 key index 15: value : 0.68 key index 16: value : 0.68 key index 17: value : 0.69 key index 18: value : 0.69 key index 19: value : 0.70 key index 20: value : 0.70 key index 21: value : 0.70 key index 22: value : 0.71 key index 23: value : 0.71 key index 24: value : 0.71 key index 25: value : 0.72 key index 26: value : 0.72 key index 27: value : 0.73	key index 40: value : 0.77 key index 41: value : 0.77 key index 42: value : 0.77 key index 43: value : 0.78 key index 44: value : 0.78 key index 45: value : 0.79 key index 46: value : 0.79 key index 47: value : 0.79 key index 48: value : 0.79 key index 49: value : 0.80 key index 50: value : 0.80 key index 51: value : 0.80 key index 52: value : 0.81 key index 53: value : 0.81 key index 54: value : 0.81 key index 55: value : 0.82 key index 56: value : 0.82 key index 57: value : 0.82 key index 58: value : 0.83 key index 59: value : 0.83 key index 60: value : 0.83 key index 61: value : 0.84 key index 62: value : 0.84 key index 63: value : 0.85 key index 64: value : 0.85 key index 65: value : 0.86 key index 66: value : 0.86

	key index 28: value : 0.73 key index 29: value : 0.73 key index 30: value : 0.74 key index 31: value : 0.74 key index 32: value : 0.74 key index 33: value : 0.75 key index 34: value : 0.75 key index 35: value : 0.75 key index 36: value : 0.76 key index 37: value : 0.76 key index 38: value : 0.76 key index 39: value : 0.76	key index 67: value : 0.86 key index 68: value : 0.87 key index 69: value : 0.88 key index 70: value : 0.88 key index 71: value : 0.89 key index 72: value : 0.89 key index 73: value : 0.90 key index 74: value : 0.91 key index 75: value : 0.92 key index 76: value : 0.94 key index 77: value : 0.95 key index 78: value : 1.00

Deletion Components

Required Files	**Functionality**
bptree_delete.c	*Core search and deletion logic*
cli.c	*Line interface and orchestration*
file_manager_btree.c	*B+ tree file I/O operations*
heapfile.c	*Heap file operations for record deletion*

Command Usage

- ./project.c delete_bplus <database_file> <threshold>

Parameters

- arg[1] = “delete_bplus”
- arg[2] = “data.db”
- arg[3] = “0.9”, this is the threshold (FT_PCT_home) that we are focusing on for deletion

Execution Flow

1. Search Phase: Find records to delete using B+ tree search
2. Performance Comparison: Run B+ tree vs linear scan comparison (3 runs each)
3. Deletion Phase: Actually delete records and rebuild B+ tree
4. Statistics Display: Show updated B+ tree structure

Core Data Structures

Struct/Classes	Functionality

RecordLocation	A struct/class that encapsulates the block_id, slot_id and key value (FT_PCT_home) value of the record that should be **deleted**
	
	typedef struct {
	uint32_t block_id; // Data block containing the record
	uint16_t slot_id; // Slot within the block
	float key_value; // FT_PCT_home value
	} RecordLocation;

SearchResult	A struct/class that represents the output of the statistic required. It stores variables such as records, count, capacity, index_nodes_accessed, leaf_node_accessed and total_key_value.
	
	typedef struct {
	*RecordLocation *records; // Dynamic array of records to delete*
	size_t count; // Number of records found
	size_t capacity; // Array capacity
	uint32_t index_nodes_accessed; // Internal nodes accessed
	uint32_t leaf_nodes_accessed; // Leaf nodes accessed
	double total_key_value; // Sum for average calculation
	double search_time_ms; // Search duration
	} SearchResult;

Constants	**Size**	**Description**
RECORD_POINTER_SIZE	8	Block ID (4) + Slot ID (4, cast to uint16_t)
KEY_SIZE	4	Float key size
[record_pointer][key]	8 + 4 = 12	B+ tree node size
Record pointer: [block][slot]	4 + 2 = 6	Node header size

Leaf Entry Format: [record_pointer(8)][key(4)] = 12 bytes per entry

Implementation Details

Helper Functions	**Technical Purpose**
ensure_records_capacity (SearchResult *result, size_t needed)	**Purpose**: Dynamically resize the “records” array when needed **Algorithm**: 1. Check if current capacity ≥ needed 2. If insufficient, calculate new capacity (start with 256, then double)

	3. Reallocate memory using realloc() 4. Update capacity field
read_leaf_key(const Node* leaf, int entry_index)	**Purpose**: Extract key value from a leaf node entry **Math rationale:** $Offset = i * (Pointer\ Size + Record\ Size) + Pointer\ Size$ $where\ i \in 0, \ldots, n$ **Algorithm**: *offset = entry_index * (RECORD_POINTER_SIZE + KEY_SIZE) + RECORD_POINTER_SIZE;* *memcpy(&key, &leaf-> bytes[offset], KEY_SIZE);*
read_leaf_record_pointer (const Node *leaf, int entry_index, uint32_t *block_id, uint16_t *slot_id)	**Purpose**: Extract record location from a leaf node entry **Math rationale:** $Offset = i * (Pointer\ Size + Record\ Size)$ $where\ i \in 0, \ldots, n$ **Algorithm**: *offset = entry_index * (RECORD_POINTER_SIZE + KEY_SIZE);* *memcpy(block_id, &leaf-> bytes[offset], 4);* *memcpy(&slot_temp, &leaf-> bytes[offset + 4], 4);* **slot_id = (uint16_t)slot_temp;*
get_next_leaf_id(Node *leaf)	**Purpose**: Get the next leaf node ID from the current leaf's link (stored in the last 4 bytes of the leaf node) **Math rationale:** $Offset = Total\ Size\ of\ a\ Node - Node\ Header\ size - 4$ **Algorithm**: *offset = (NODE_SIZE - NODE_HDR_SIZE) - 4;* *uint32_t next_id;* *memcpy(&next_id, &leaf-> bytes[offset], 4);*

Main Functions	**Technical Purpose**
bptree_range_search(char *btree_filename, float min_key, SearchResult *result)	**Purpose**: Search for all records with key > min_key using B+ tree index and store into “records” array **Algorithm**: 1. Find Root Node: Scan nodes to find highest level (root) 2. Navigate to First Leaf: Traverse internal nodes to find starting leaf node 3. Leaf Scanning:

	◦ Read each leaf node sequentially ◦ For each entry with key > min_key, store in "records" ◦ Follow next-leaf pointers until end of chain 4. Statistics Collection: Track node accesses and timing **Key Optimizations**: ● Starts scanning from the first **relevant** leaf that contains the first record where key > min_key (not from beginning) ● Uses linked-list structure of leaves for efficient sequential access
bptree_perform_deletion(char *db_filename, SearchResult *result)	**Purpose**: Delete records from heap file and rebuild B+ tree **Algorithm**: 1. Sort Records: Order by block_id and slot_id (descending) to avoid slot shifting 2. Delete Records: Use hf_delete_record() for each location 3. Rebuild Index: Call scan_db() to reconstruct B+ tree 4. Statistics: Report deletion count and success **Critical Detail**: Deletion order prevents slot shifting issues in heap file blocks. By deleting in reverse order (highest slot numbers first within each block), we ensure that: 1. **Within each block**: We delete from the end backward, so shifting doesn't affect the slot IDs of records we still need to delete 2. **Across blocks**: We process higher-numbered blocks first, then lower-numbered blocks
linear_scan_search(char *db_file, float min_key, SearchResult *result)	**Purpose**: Brute force search through entire database for comparison **Algorithm**: 1. Open Database: Access heap file directly 2. Block-by-Block Scan: ◦ Fetch each block using bp_fetch() ◦ Read block metadata to get record count ◦ Decode each record and check FT_PCT_home value 3. Result Collection: Store qualifying records in SearchResult 4. Performance Tracking: Count data blocks accessed **Block Record Count Extraction**: *int used = block_used_count(cur); // Reads little-endian from bytes[0:1]* **Example**: *bytes[0] = x34* *bytes[1] = x12* *bytes[0] | bytes[1] << 8 = x1234 (little-endian)*

run_comparison_tests()	**Purpose**: Execute performance comparison between B+ tree and linear scan **Algorithm**: 1. Multiple Runs: Execute each search method 3 times 2. Average Calculation: Compute mean performance metrics 3. Comparison Metrics: ○ Time speedup: linear_time / btree_time ○ I/O reduction: linear_blocks / btree_nodes **Math rationale:** $Time\ Improvement\ = \frac{averaged\ time\ for\ linear\ scan}{averaged\ time\ for\ b+tree}$ $I/O\ reduction\ = \frac{averaged\ no.\ of\ block\ accessed\ linear}{averaged\ no.\ of\ block\ accessed\ b+tree}$

Performance Metrics/Statistics

B+ Tree Search Output	**Linear Scan Output**
● Records found ● Internal nodes accessed (typically 1 for height-2 tree) ● Leaf nodes accessed ● Total nodes accessed ● Average key value ● Search time (ms)	● Records found ● Data blocks accessed (entire database) ● Average key value ● Search time (ms)

Installation Steps

1. This project requires a C++ compiler. Mingw, clang, or gcc are recommended.
2. Download the source code folder from the provided URL and unzip the downloaded .zip file and open a command prompt or terminal at the unzip location.
3. Run the following command lines in the terminal:
   - **gcc -std=c11 -O2 -Iheader src/*.c -o project_c**
   - **./project_c load games.txt data.db**
   - **./project_c stats data.db**
   - **./project_c delete_bplus data.db 0.9**