SC3020 / CZ4031
DATABASE SYSTEM PRINCIPLES

Project 1

Group 13	
Sky Lim En Xing	U2223731A
Gupta Raghav	U2222889A
Mythili Mulani	U2222678G
Minhwan Kim	U2022178H
Chew Wei Hao Kovan	U2322227A

Table of contents

01
Storage Component Design

- Data Components
- Controller Components
- Buffer Pool

02
B+tree Components

- Record Extraction & Ordering
- Leaf Node Construction
- Building internal nodes

03
Deletion

- B+ Tree Search
- Comparison Tests
- Deletion function

Task 1: Data Components

Storage format
(Fixed-width packed record format)

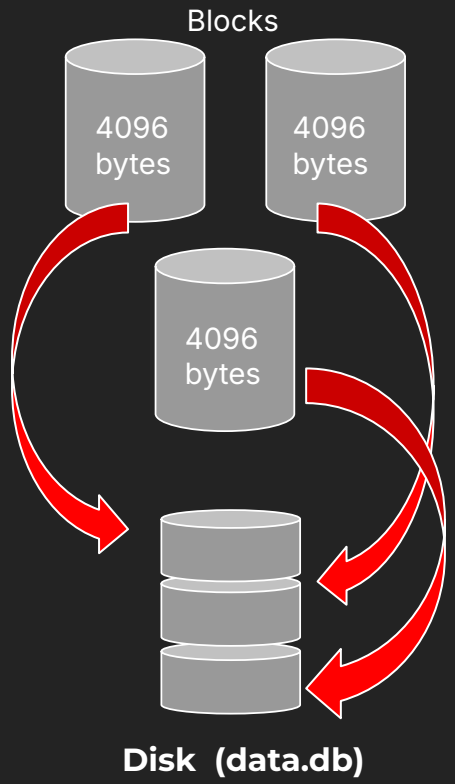
Each Field in game.txt:

INT32 ASSIGNED **4 BYTES**

FLOAT ASSIGNED **4 BYTES**

CHAR[10] ASSIGNED **10 BYTES**

Entire record packed exactly into a 27-byte



Implementation Details (Key Functions)

- **encode(row → bytes):** Packs a Row struct into a 27-byte buffer for storage.
- **block_write_record(block, slot, bytes):** Writes a record into a specific slot in the block.
- **hf_load_csv():** Parses games.txt, encodes rows, fills blocks, and writes them sequentially into the heap file (data.db).

Controller Components

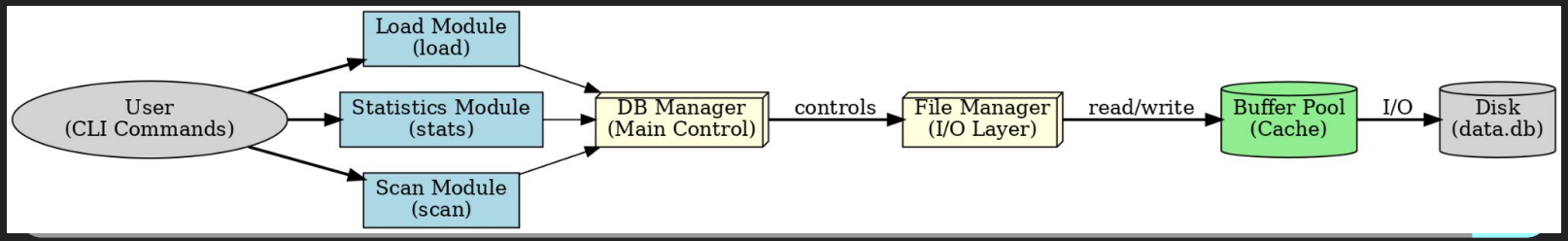
The controller layer acts as the bridge between user commands and physical storage.

- File Manager handles raw read/write between memory and disk.
- Database Manager provides higher-level modules:

  load( ): Reads games.txt, encodes rows into 27-byte records, fills blocks, and writes them to the heap file.

  hf_stats( ): Traverses the heap file and reports statistics such as record size, number of records, records per block, and total blocks.

  hf_scan( ): Sequentially scans all blocks and decodes the stored records for display.



Buffer Pool

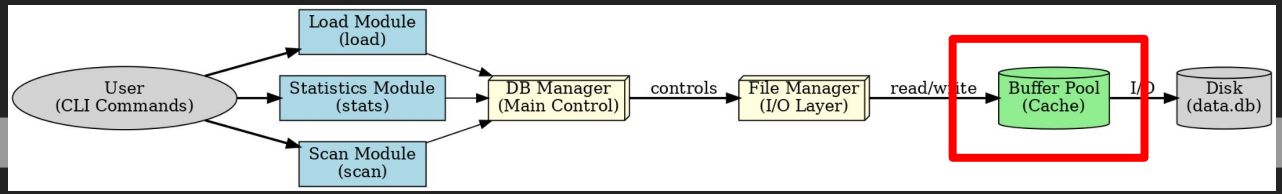
To reduce expensive disk I/Os, we implemented a buffer pool cache.

- The buffer pool maintains an in-memory array of blocks.
- On fetching a block, if it's cached we return it (cache hit). If not, we load it from disk.
- When the buffer is full, we apply the **Least Recently Used (LRU) replacement policy**.
    - If the evicted block is dirty, we flush it back to disk; otherwise, it's discarded.

bp_fetch(block_id): Fetches a block from disk into memory. If already cached, return directly; otherwise, load into an available slot.

bp_evict( ): Evicts the least recently used block when the buffer is full. If the evicted block is dirty, it is flushed back to disk.

bp_flush(block_id): Explicitly writes a dirty block back to disk to ensure persistence.



Task 2: B+tree Component

Implementation Details (Key Functions):

- **scan_db()** — orchestrates heap scanning and key extraction and leaf node creation
- **bulkload()** — builds tree bottom-up from leaf nodes
- **pack_internals()** — partitions children, builds internal nodes
- **btfm_write_node()** — writes 4 KB nodes to disk

Highlights:

- Builds an **on-disk B+ tree** directly from heap file (**BULK LOADING**)
- Each phase materializes sorted data → leaf nodes → internal hierarchy.
- Produces persistent btree.db file for indexed access.

Record Extraction & Ordering

Key Functions:

- **scan_db()** (*build_bplus.c*)
- KeyPointer struct (key, block_id, slot_id)

Process:

1. Traverse each heap block through **buffer pool**.
2. Decode rows → stage as (key, block_id, slot) tuples.
3. Grow array dynamically as new tuples arrive.
4. Perform **total-order sort** on key field (ascending).

Result: Stable input ordering ensures deterministic B+ Tree structure.

Leaf Node Construction

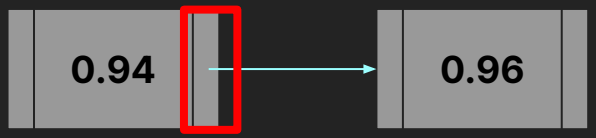
Node format

HEADER **11 BYTES**

INT8 Level **1 BYTES**
INT32 node_id **4 BYTES**
INT16 Key_count **2 BYTES**
FLOAT lower_bound_key **4 BYTES**

BYTES ARRAY

FLOAT key **4 BYTES**
POINTER
INT32 Block id **4 BYTES**
INT slot id **4 BYTES**



Stores the id of the next leaf

Key Functions:

- **node_init(), node_write_record_key(), link_leaf_node(), btfm_write_node()**

Process:

- Create **in-memory leaf nodes** of up to MAX_LEAF_KEYS entries.
- Each node records its **lower-bound key**.
- When a node fills → write to disk immediately.
- Neighbor leaves chained via **sibling pointers** for sequential range access.

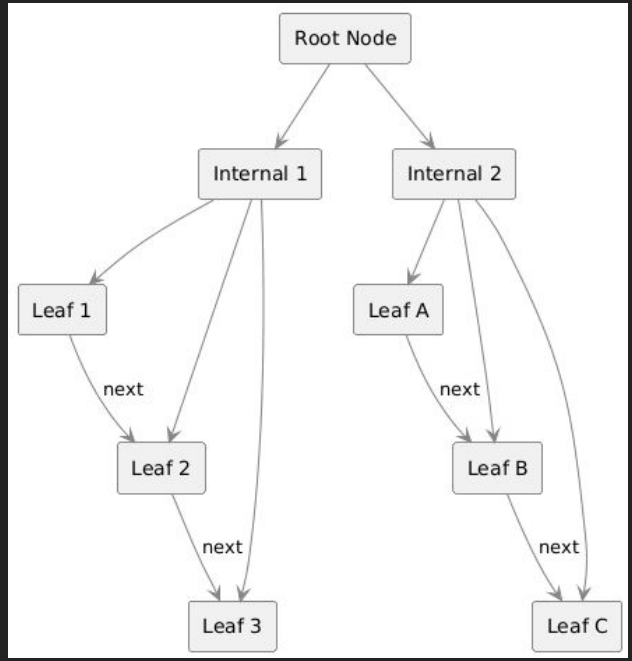
Building Internal Nodes

Key Functions:

- **bulkload()** (*bptree_construct.c:6*)
- **pack_internals()** (*bptree_construct.c:32*)

Process:

1. Consume leaf **lower-bound keys**.
   Group children ≤ MAX_INT_CHILDREN.
2. Build parent nodes inserting **first-key separators**.
3. Handle edge case: borrow redistribution if last node underflows.
4. Repeat until **one root node remains**.



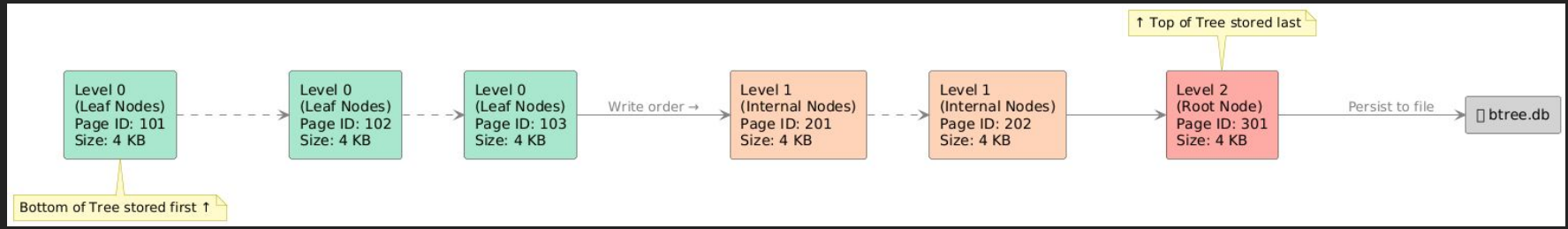
Persistent & Layout of B+ Tree

Key Functions:

- **BtreeFileManager** (*file_manager_btree.c*)
- **encode_node()** / **decode_node()** (*bptree_node.c*)

Structure:

- Each B+ tree node is written as a **4 KB page** on disk.
- **Page header fields:**
  • Level • Node ID • Key Count • Lower-Bound Key
- **Pages are stored in ascending order of levels — leaf (level 0)** pages written first, then their parent (level 1), and so on until the root.



Task 3: Deletion

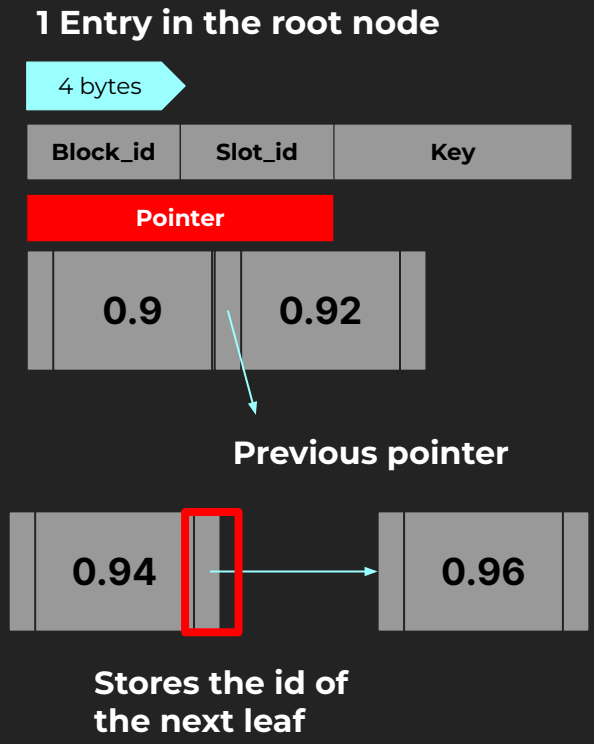
(Linear Search vs B+tree Search)

Implementation Details (Key Functions)

- bptree_range_search(btree db heap file, threshold, SearchResult)
- linear_scan_search(db heap file, threshold, SearchResult)
- run_comparison_test()

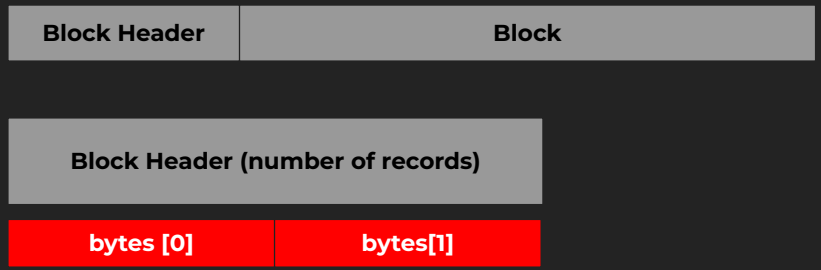
bptree_range_search function

- Find the root node first by looping through bptree.db **until the max height (height = 2)** is found.
- Loop through each entry of the root node;
- Locate the leaf node to travel to: Comparing key value > threshold
- Iterate through the leaf node and find the first key value > threshold
- Stores the records that should be deleted in the array “records” in SearchResult
- Go to the next leaf node with get_next_leaf_id() and **repeat**.



linear_scan_search function

- Loop through each record through indexing up to the index value “used” in a “**sequential flooding fashion**”
- **In each loop, compare if the current record’s key > threshold,**
- **Store it in the “records” array in SearchResult.**
- Repeat until no blocks or records.



run_comparison_test function

- Loop 3 times for bptree_range_search()
- Store each statistics in the array with type SearchResult
- Loop 3 times for linear_scan_search()
- Store each statistics in the array with type SearchResult
- Compute the average time taken for both scans

```
=== COMPARISON RESULTS ===
B+ Tree Search (average of 3 runs):
  Time: 0.357 ms
  Nodes accessed: 8
  Records found: 1778

Linear Scan (average of 3 runs):
  Time: 1.031 ms
  Blocks accessed: 176
  Records found: 1778

Performance Improvement:
  Time speedup: 2.88x
  I/O reduction: 22.00x

Proceeding with deletion of 1778 records...
Successfully deleted 1778 records from database.
Rebuilding B+ tree index...
```

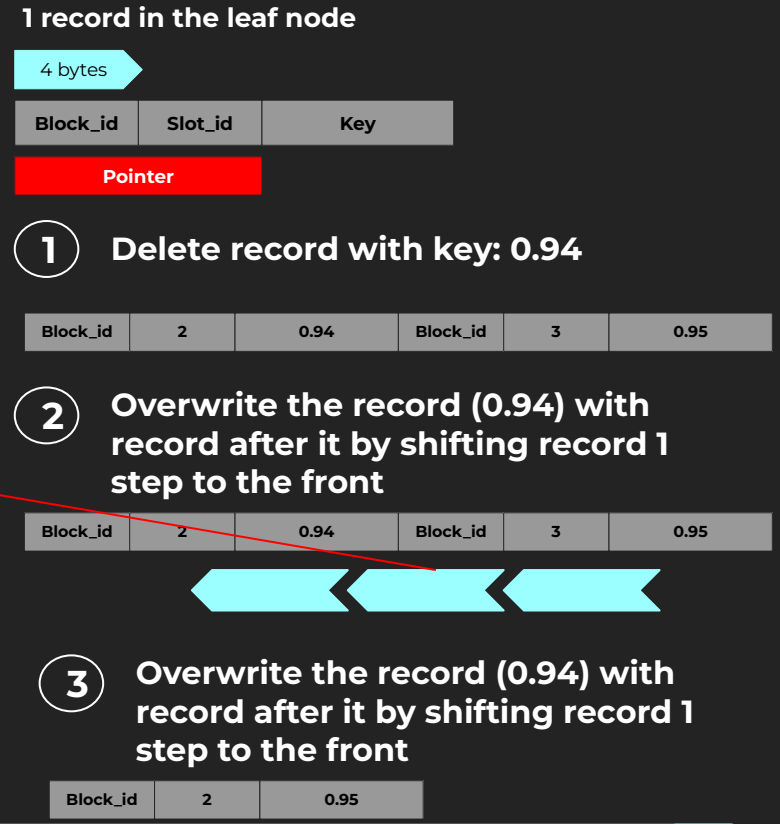
Task 3: Deletion

Implementation Details (Key Functions)

- hf_delete_record (db heap file, block_id, slot_id)
- bptree_perform_deletion(db heap file, SearchResult)

hf_delete_record function

- For each record that should be deleted in the array “records” in object SearchResult
- Get the block based on the record’s block_id from buffer pool
- Using slot_id of the current record, increment it by 1 to get the next record after it.
- Loop through and overwrite the deleted record at its position with the records after it by shifting all the records position to the left

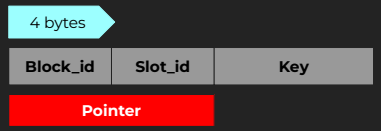


bptree_perform_deletion

- Array in SearchResult, “records” that store the records that should be deleted
- **Selection sort** the order of the records in **descending** order based on block_id or slot_id
- Once sorting is done, **hf_delete_record()** to delete each record based on the block_id and slot_id for identification.

- **Selection sort to prevent complicated shifting during deletion**
- **hf_delete_record() shifts the records to the left, altering each record’s slot_id - 1 in the process.**
- **Next deletion will target the wrong record since the slot_id has changed.**
- **Sorting slot_id in descending order will prevent this**

1 record in the array “records”



1 **Descending order after sort, delete slot_id {5,3}**

5	0.98	4	0.97	3	0.96

2 **hf_delete_record() will shift the records (0.97, 0.96) to the left but because slot_id + 1 is 6, it will just be “empty” record, so the other records’ slot_id stays as it is {4,3} and {5} will be empty**

5	0.98	4	0.97	3	0.96

3 **Updated**

Garbage	Garbage	4	0.97	3	0.96

Thank You !!