NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC3020 Database System Principles

Project 1 (20%)

Report Submission

Group 4

[Source Code Link]

[Video Link]

Prepared By:

Name	Matriculation Number
Lohia Vardhan	U2120105F
Teoh Xi Sheng	U2120456L
Iyer Anushri	U2123122E
Pugalia Aditya Kumar	U2123212D
Mukherjee Tathagato	U2120365K

Table of Contents

1. Introduction

In the era of big data, efficient data storage and retrieval are critical for managing large datasets effectively. This project focuses on the design and implementation of two main components of a database management system: the storage component and the indexing component.

1.1 Assumptions

This project will be implemented using the C programming language and the data block size will be set to 4096 Bytes to ensure consistent results across different devices. We assume that the data fields are in fixed format with fixed length and null data will be excluded from data processing.

2. Task 1: Storage Component Implementation

2.1 Initializing and Storing Data into Disk

We first define the structure of the record, the block and its header (metadata) :

```c
// Record struct definition
typedef struct {
    char GAME_DATE_EST[11];  // Format: DD/MM/YYYY, 11 bytes
    int TEAM_ID_home;        // 4 bytes
    int PTS_home;            // 4 bytes
    float FG_PCT_home;       // 4 bytes
    float FT_PCT_home;       // 4 bytes
    float FG3_PCT_home;      // 4 bytes
    int AST_home;            // 4 bytes
    int REB_home;            // 4 bytes
    int HOME_TEAM_WINS;      // 4 bytes
} Record;

// Metadata structure for a block
typedef struct {
    int block_id;        // Metadata: Block ID
    int record_count;    // Metadata: Number of records in the block
    int record_size;     // Metadata: Size of each record in bytes
} BlockMetadata;

// Block struct definition
typedef struct {
    BlockMetadata metadata;  // Metadata of the block
    Record *records;         // Array of records
} Block;
```

Figure 1. Code snippet for defining record, metadata and block structure

We then read the “games.tx” into the memory using the following steps:

1. Open the file using “fopen” function from C with “rb” read binary mode
2. Allocate memory for storing the Record structures
3. Read each line of the file into memory using “fgets” while using Tab as the delimiter
4. Close the file

The following table shows a general statistics of the records data:

Number of Records scanned (incl. header)	26,652
Number of Records skipped (null)	99
Number of Records to write to disk	26,552

Table 1. Record statistics

The following steps were then executed to write the data from memory into blocks in disk:

1. Open the destination binary file “data.db” using “fopen” function from C with “wb” read binary mode
2. Calculate the size of BlockMetadata
3. Calculate the maximum number of records can fit into a block
4. Calculate the total number of blocks needed to store all records
5. Allocate memory for storing the Block structures
6. Iterate over each record, fill the block and write the block to the destination file
7. Free memory and close the file

2.2 Reading Data from Disk into Memory

To read all the blocks and gather the statistics, the following steps were executed:

1. Open the binary file using “fopen” function from C with “rb” read binary mode
2. Calculate the size of BlockMetadata
3. Calculate the maximum number of records can fit into a block
4. Calculate the total number of blocks needed to store all records and allocate memory for blocks
5. Set the block offset to 0 and seek to the starting block using “fseek”
6. Iterate over each block
7. Allocate memory for records based on the block metadata and read records
8. Close the file

After reading all the blacks from disk, we gathered the following storage statistics:

Size of a Record	44 Bytes
Size of a Block Metadata	12 Bytes
Size of a Block	4,096 Bytes
Total number of Records	26,552 Records
Maximum number of Records per Block	92 Records
Total number of Blocks to store all Records	289 Blocks

Table 2. Storage statistics

We can visualize the storage component with the following diagrams:

Even though the record only occupies 43 Bytes, an additional 1 Byte is added as padding by the compiler. This is to align the struct to a 4-Byte boundary, ensuring that each record in an array is properly aligned in memory.

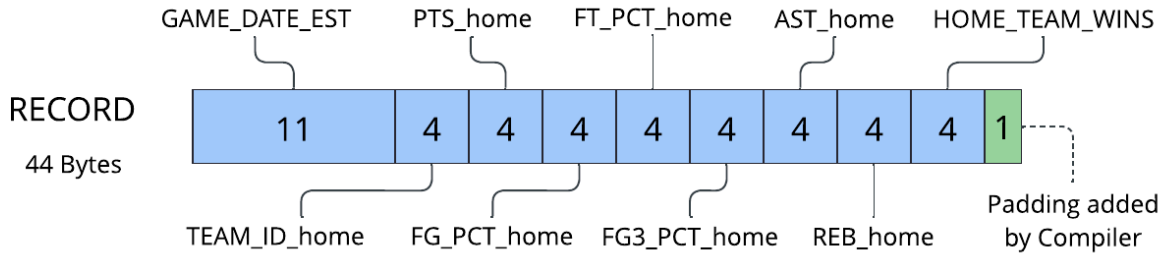

Figure 2. Record structure and size

The block metadata includes a block_id to uniquely identify a block, a record_size and record_count to improve data access efficiency. They are integers and occupy a total of 12 Bytes.

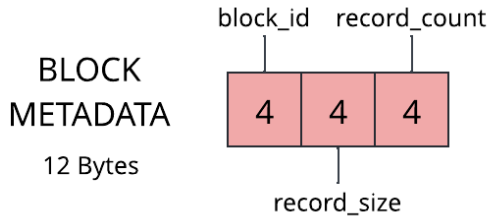

Figure 3. Block metadata structure and size

We can store up to 92 records for each block inclusive of the block header. This will leave the block with 36 Bytes of empty data.

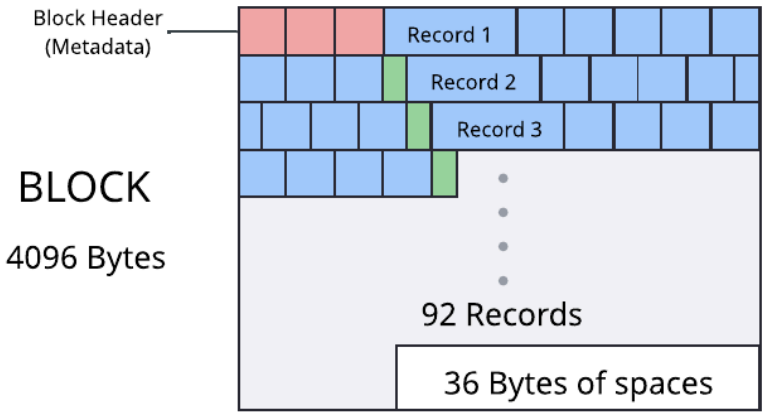

Figure 4. Block structure

Therefore, we require a total of 289 blocks of data to completely store all the records from "games.txt". The last block will only consist of 56 records and 1,260 Bytes of empty data.

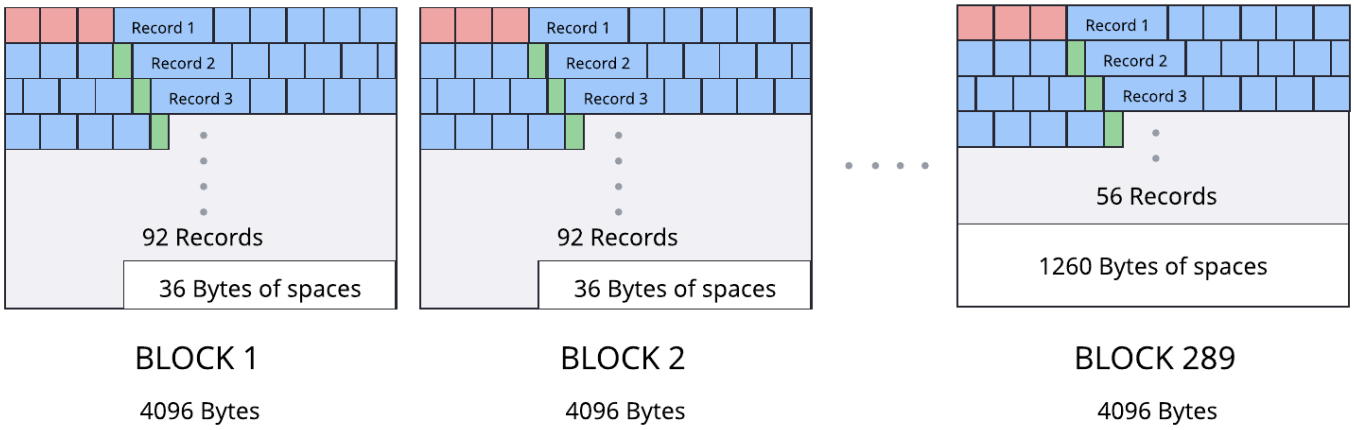

Figure 5. Total data blocks required for "games.txt"

3. Task 2: Indexing Component (B+ Tree) Implementation

After the main driver has read the games.txt file and stored on the disk in blocks, each record contains the FG_PCT_home attribute, which serves as the key for indexing. The following code snippet shows the sorting of records based on the aforementioned attribute:

```
// Sort the records by FG_PCT_HOME
printf("Sorting records by FG_PCT_home...\n");
qsort(records, num_records_to_write, sizeof(Record),
compare_records_by_fg_pct_home);

// Write the sorted records back to the binary file
printf("Writing sorted records to data.db...\n");
if (write_blocks(DATA_FILE, records, num_records_to_write, BLOCK_SIZE) != 0) {
   fprintf(stderr, "Failed to write blocks.\n");
   free(records);
   return EXIT_FAILURE;
}

free(blocks);
free(records);
```

Figure 6. Code snippet for sorting data before creating index

2.1 Structure of B+ Tree Node

Each node in our B+ Tree is represented using a structured format that allows for efficient storage, retrieval, and manipulation of keys and pointers. The node structure is defined as follows:

```
// B+ Tree node structure
typedef struct Node {
    bool is_leaf;       // Indicates if the node is a leaf
    int num_keys;       //  Number of keys in the node
    long parent;        // parent of the B+ tree node
    float keys[N + 1];
    long children[N + 2];
} Node;
```

Figure 7: Structure of B+ Tree Node

1. is_leaf: The boolean field distinguishes between leaf in and internal nodes. If the value is True, the node is a leaf and directly points to data records. Otherwise, it is an internal node that points to child nodes in the tree.
2. num_keys: This field keeps track of the number of keys currently stored in the node. The value of num_keys can vary between 1 and N, where N is the maximum number of keys a node can hold before splitting.
3. parent: This field holds the file offset of the parent node. It helps maintain the hierarchical relationship between nodes.
4. keys[N + 1]: This array stores the keys in the node. The array size is N + 1 to accommodate the possibility of overflow during node insertion, where an additional key might temporarily be held before splitting.
5. children[N + 2]: This array holds pointers to either child nodes (for internal nodes) or data records (for leaf nodes). For internal nodes, it stores the offsets to child nodes. For leaf nodes, it stores the data offsets.

2.1 Initializing B+ Tree

Once the records have been sorted and stored in a binary file, create_bplus_tree() is used to initialize the tree and prepare the Key-Pointer (KP) pairs for insertion. The steps are outlined below, followed by the function in its entirety.

Initialize Variables and Extract KP Pairs:

- For each block in the data file, the function reads the data from the file using the read_block() function, which retrieves a block of records and provides metadata (like total records and record size).
- Inside the above loop, another loop iterates through each record in the block. For each record:
    - The offset is calculated as the current block number multiplied by the block size, plus the record index multiplied by the size of the record.
    - The key, FG_PCT_home, and calculated offset are stored in the KP_Pair array, which are used for insertion into the tree later.

Create the Root Node of the B+ Tree:

- Using create_node(), a node is initialized, which acts as the root node to start with.
- The root node is then written to the index file using write_node_to_disk() at an offset of 0, which means it's placed at the beginning of the index file.
- The variable root_offset, initialized as 0, tracks the offset of the root node. This can change after an insertion of a node.

Insert Keys into the B+ Tree:

- Another loop iterates over all the records. For each key-pointer pair, the insert_into_bplus_tree() function is called to insert the key and its offset. This function manages the insertion into the correct leaf node and handles balancing if necessary, as outlined in the next subsection.

The function returns a root_offset, which points to the beginning of the B+ Tree in the index file. This pointer is used to get statistics about the B+ tree later.

2.2 Inserting Node into B+ Tree

Asides from the splitting and balancing of nodes, the insertion of nodes is handled by insert_into_bplus_tree(), as outlined below:

1. Initialize Variables and Read the Root Node

- Read the root node from the index file and assign it to current_node. Its offset is stored in current_offset.

2. Traverse the Tree to Find the Leaf Node

- Traverse the B+ Tree until it reaches a leaf node.
- Update current_offset and reads current_node from the correct child until it finds the appropriate leaf node for the key.

3. Insert the Key into the Leaf Node

- Find where to insert the new key by shifting existing keys to the right if needed.
- Inserts the key and its data offset into the leaf node and updates current_node.num_keys.

Finally, the function writes the updated leaf node back to the index file using write_node_to_disk() and handles overflows by calling **split_node()**, as outlined in the next subsection.

2.3 Splitting Nodes and Balancing

The function split_node() handles when a node overflows, including promoting keys to parent nodes and creating new nodes if necessary. The steps are outlined below:

1. Create a New Node to Store Half of the Keys

- A new node (new_node) is created using createNode(), which has the same type (is_leaf) as the current node.
- The number of keys for new_node is set based on whether it is a leaf or internal node.
- The number of nodes (num_nodes) is incremented to keep track of the total nodes in the B+ Tree.

2. Move Half of the Keys and Children to the New Node

- **If the current node is a leaf:**
    - Keys and their corresponding children (record offsets) from the second half of the current node are moved to new_node.
    - The middle key (middle_key) is set to the first key of the new_node (i.e., the first key that was moved to the new node).
- **If the current node is an internal node:**
    - Keys and child pointers from the second half of the current node are moved to new_node.
    - The middle key is set as the key at the position (ceil(N/2) + 1) from the current node.
    - The last child pointer in the current node is also transferred to new_node after all keys are moved.

3. Write the New Node to the Index File

- The newly created node (new_node) is written to the index file using create_bplus_tree_node(), and its offset (new_node_offset) is stored.
- If the current node is a leaf, the pointer to the next leaf (children[N]) is updated to point to new_node and the pointer to the next leaf for the current node is set to the new_node.

4. Handle Parent Node Update

- **If Splitting the Root Node:**
  - A new root is created with one key (middle_key) and two child pointers, pointing to current_node and new_node.
  - The new root is written to the index file, and root_offset is updated to point to this new root.
  - The parent pointers of both current_node and new_node are updated to point to the new root.
- **If Not Splitting the Root Node:**
  - The middle key is promoted to the parent node.
  - The parent node is updated by inserting the middle key and adding a pointer to the new_node.
  - If the parent node overflows, it is split recursively.

5. Write Updated Nodes to the Index File

- The current node (current_node) is updated to reflect the number of keys it now has (half of the original).
- The current node and new node are written back to the index file.
- Finally, the newly created node (new_node) is freed from memory.

The split_node function ensures that the B+ Tree maintains its balance when a node becomes full, either by creating a new root (if splitting the root) or by promoting a middle key to an existing parent. It helps maintain the structure of the B+ Tree by dividing keys between nodes and ensuring parent-child relationships are correctly updated.

2.4 Indexing Statistics

N in B+ Tree	338
# of Nodes in Tree	157
# of Levels	2
Content of Root Node	0.325 0.338 0.348 0.354 0.359 0.364 0.367 0.370 0.373 0.376 0.378 0.381 0.383 0.385 0.388 0.389 0.391 0.393 0.395 0.397 0.398 0.400 0.400 0.402 0.404 0.405 0.407 0.408 0.410 0.411 0.412 0.413 0.415 0.416 0.417 0.418 0.419 0.420 0.422 0.423 0.424 0.425 0.426 0.427 0.429 0.429 0.430 0.431 0.432 0.433 0.434 0.435 0.436 0.437 0.438 0.439 0.440 0.441 0.442 0.443 0.444 0.444 0.446 0.447 0.447 0.448 0.449 0.451 0.451 0.452 0.453 0.455 0.455 0.456 0.457 0.458 0.459 0.460 0.461 0.462 0.462 0.463 0.464 0.465 0.466 0.467 0.468 0.469 0.470 0.471 0.471 0.472 0.473 0.474 0.475 0.476 0.477 0.478 0.479 0.481 0.481 0.482 0.483 0.484 0.486 0.487 0.488 0.488 0.489 0.490 0.493 0.494 0.494 0.494 0.495 0.500 0.500 0.500 0.500 0.500 0.505 0.506 0.506 0.506 0.507 0.511 0.512 0.513 0.514 0.516 0.518 0.519 0.520 0.522 0.524 0.525 0.527 0.529 0.531 0.533 0.536 0.538 0.541 0.543 0.545 0.548 0.551 0.554 0.558 0.562 0.566 0.571 0.579 0.588 0.602

Table 3. Indexing Result Statistics

4. Task 3: Evaluation on B+ Tree

4.1 Implementation of B+ Tree Search

```
// Traverse to the leaf level
while (!(current_node.is_leaf)) {
   int i = 0;
   while (i < current_node.num_keys && lower > current_node.keys[i]) i++;
   fseek(index_file, current_node.children[i], SEEK_SET);
   fread(&current_node, sizeof(Node), 1, index_file);
   num_index_nodes_accessed++;
}
int ind = -1;
// Search within the leaf nodes for keys in the specified range
for (int i = 0; i < current_node.num_keys; i++) {
   if (current_node.keys[i] ⩾ lower && current_node.keys[i] ⩽ upper) {
       ind = i;
       break;
   }
}
while (ind ⩽ current_node.num_keys){
   if (ind == current_node.num_keys){
       if (current_node.children[N] == -1) break;
       fseek(index_file, current_node.children[N], SEEK_SET);
       fread(&current_node, NODE_SIZE, 1, index_file);
       ind = 0;
       num_index_nodes_accessed++;
   }
   else{
       if(current_node.keys[ind] > upper){
           break;
       }
       block_num = current_node.children[ind] / BLOCK_SIZE;
       record_num = (current_node.children[ind] % BLOCK_SIZE) / sizeof(Record);
       if (block_num ≠ current_block_num){
           read_block(filename, &block, block_num, &max_records_per_block,
&total_records, &record_size);
           current_block_num = block_num;
           num_blocks_accessed++;
       }
       result.records[result.count] = block.records[record_num];
       result.count++;
       ind++;
       if (result.count == capacity){
           capacity *= 2;
           result.records = (Record *) realloc(result.records, sizeof(Record) *
capacity);
       }
   }
}
```

Figure 8: Main logic for B+ search

We followed the following steps to implement the B+ tree Search:

1. Set the current node to be the root node.
2. While the current node is not a leaf node do:
    a. Find the smallest key that is larger than the lower bound of the range query and set the current node = the child of the key.
3. Loop through the current leaf node to find the smallest key that is larger or equal to the the lower bound and set the index of that key as the current index.
4. Loop through the indexes in the leaf nodes to retrieve records till the smallest key larger than the upper bound of the range query is found.

4.2 Implementation of Brute-Force Linear Search

In this method, each record in each block is examined to check if FG3_PCT_home value falls within the range. The time complexity is O(n) where n is the total number of blocks.

We followed the following steps to implement the Brute-force Linear Search:

1. Allocate space for the resulting records.
2. Loop through all used blocks and read block from file via the read_block function.
3. Loop through all the records in the block based on the record count stored in block header.
4. If record falls in required range, it is added to the result record array. If the resulting record count is more than the space allocated, dynamically allocate more capacity.

```
SearchResult result;
result.records = NULL;
result.count = 0;

float sum_FG3_PCT_home = 0.0;
int capacity = 100;
result.records = (Record *)malloc(sizeof(Record) * capacity);
if (!result.records) {
   perror("malloc");
   fclose(data_file);
   return result;
}
int max_records;
int total_records;
int record_size;

for (int i = 0; i < num_blocks; i++) {
   Block block;

   read_block(data_filename, &block, i, &max_records, &total_records,
&record_size);

   for (int j = 0; j < block.metadata.record_count; j++) {
       Record rec = block.records[j];
       if (rec.FG_PCT_home ⩾ lower && rec.FG_PCT_home ⩽ upper) {
           if (result.count ⩾ capacity) {
               capacity *= 2;
               result.records = (Record *)realloc(result.records,
sizeof(Record) * capacity);
               if (!result.records) {
                   perror("realloc");
                   fclose(data_file);
                   SearchResult empty = {0, NULL};
                   return empty;
               }
           }
           result.records[result.count++] = rec;
           sum_FG3_PCT_home += rec.FG3_PCT_home;
       }
   }
}
```

Figure 9: Main logic for brute-force search

4.3 Search Results

Index Nodes Accessed	42
# of Data Blocks Accessed	76
Average of "FG3_PCT_home" Records Returned	0.421
Run Time (B+ Search)	0.000898 seconds
# of Data Blocks Accessed (Brute Force Linear Scan)	289
Run Time (Brute Force Linear Scan)	0.007439 seconds

Table 4. Search Result Statistics

The run time is determined using the clock() function. This is calculated as follows with average_FG3_PCT_home.

```
*average_FG3_PCT_home = (result.count > 0) ? (sum_FG3_PCT_home / result.count) : 0.0;
clock_t end = clock();
*time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

return result;
```

Figure 10: Run time, Average of FG3_PCT_home code

4.4 Comparison between B+ tree search and Brute Force Linear Scan Performance

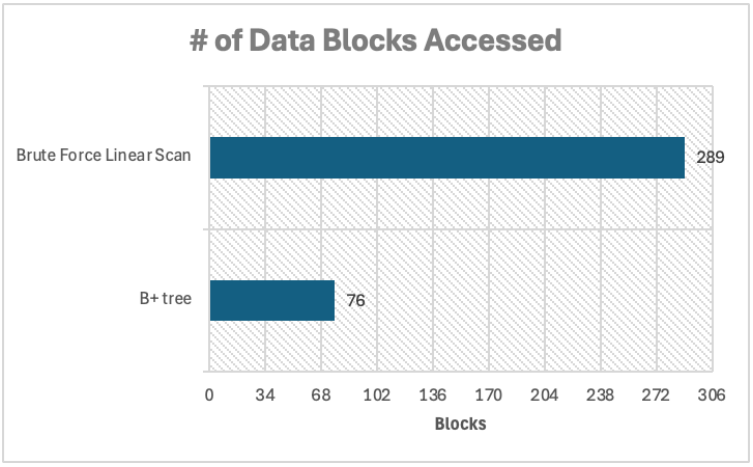

Figure 11: Comparison in # of Data Block Accessed

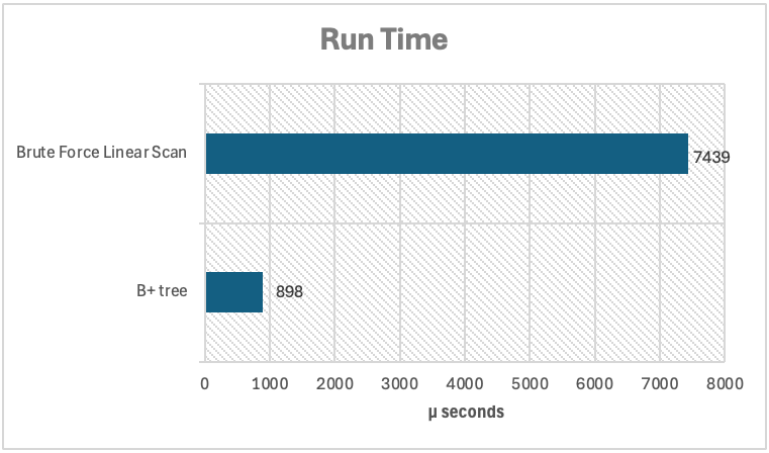

Figure 12: Comparison in run time (microseconds)

As shown in *Figure 11*, the number of blocks accessed by the brute-force linear scan method is significantly larger i.e. **380% larger** than the number of data blocks accessed by the B+ tree. The results match the time complexity described earlier as Number of Data Blocks Accessed in brute-force linear scan = Total Number of Data Blocks (*Table 2*). In comparison, as B+ tree narrows the search space to perform a search query, the block accesses required is lower and the overall I/O cost is lower.

This is also reflected in the runtime comparison in *Figure 12* where the Brute Force Linear Scan is approximately **828% greater** than the B+ tree search owing to a lower number of data blocks being accessed.