SC3020: Database System Principles

Group 9

Project 1 Report

Name	Matriculation Number
Lee Hao Guang	U2122950A
Leong Hong Yi	U2120932C
Lim Jun Hern	U2120981B
Yap Xuan Ying	U2021297G

* Each group member contributed equally to the project

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY

Table of Content

Introduction

Project Goal

This project is to design and implement the storage and indexing components of a database management system. The entire system was developed using **Java**.

Installation Guide

Please refer to the **Installation Guide.pdf** found in the same folder.

Design and Implementation of Storage Components

Our design includes four primary components for storage: `Record`, `Block`, `Disk`, and `Address`. Each `Record` is stored within a `Block`, and the `Disk` accommodates multiple of these `Blocks`. The `Address` class has been developed to address the absence of pointer features in Java.

Record

In our approach to organising the fields into records, we opted for a **fixed format with fixed length** to make it easier to be interpreted and simplify the execution of various operations. Each `Record` contains nine distinct attributes, which are as follows:

Field Name	Data Type	Size	Explanation
`GAME_DATE_EST`	String	8	• The field in the original file contains dates represented in slashes with variable length. • Each date will be converted into an 8-character string before storing into the database. • We only consider the characters for the size of the string.
`TEAM_ID_home`	int	4	• The fields contain values that do not exceed the maximum int value of 2147483647.
`PTS_home`	int	4	
`FG_PCT_home`	float	4	• The fields contain decimal values that can be accurately represented using float data type without the need for rounding.
`FT_PCT_home`	float	4	
`FG3_PCT_home`	float	4	
`AST_home`	int	4	• The fields contain values that do not exceed the maximum int value of 2147483647.
`REB_home`	int	4	

HOME_TEAM_WINS	boolean	1	• The field contains value either 0/1, which implies True or False.
Assuming the fields can start at **any byte**, Total size of a record: 8 + 7×4 + 1 = **37 bytes**			

> **Note:** Since the size of booleans and characters in Java is not fixed, we assume that both will take up 1 byte each.

To simplify our implementation, we chose not to utilise the getter and setter method approach within the `Record` class. Instead, we allow the attributes to be **directly accessed or modified** by using the `public` access modifiers.

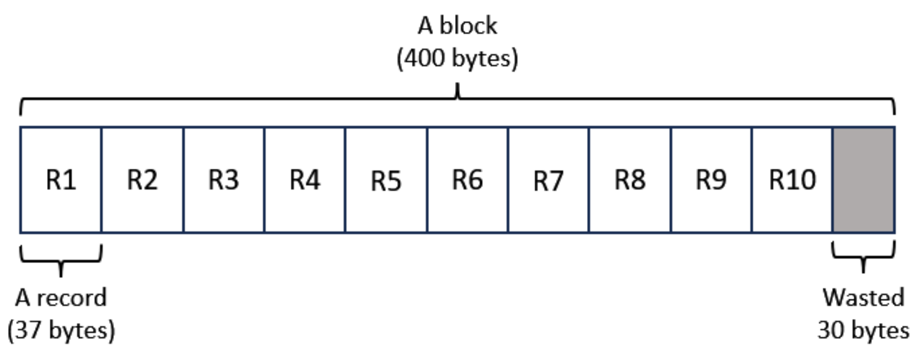
```java
public class Record {
	public static final int size = 37;
	public String GAME_DATE_EST;
	public int TEAM_ID_home;
	public int PTS_home;
	public float FG_PCT_home;
	public float FT_PCT_home;
	public float FG3_PCT_home;
	public int AST_home;
	public int REB_home;
	public boolean HOME_TEAM_WINS;
}
```

Block

Each `Block` contains the following attributes:

Attribute	Description
`maxRecordCount`	• The maximum number of records that a block can hold.
`recordCount`	• Stores the current number of records present in the block.
`records`	• Stores the actual records.

Considering that a block size is **400 bytes**, and the size of a record is **37 bytes**, each block would be able to hold at most **10 records**, as shown in the diagram below:



> **Note:** Keys for B+ Tree will be stored in another class called `Nodes`.

```
public class Block {
	public static int maxRecordCount =
		(int) Math.floor(Config.BLOCK_SIZE / Record.size);
	int recordCount;
	Record[] records;
}
```

Disk

In our implementation, we configured the capacity of our `Disk` to be **100 MB**. The `Disk` contains the following attributes:

Attribute	Description
`maxBlockSize`	• The maximum number of blocks that the disk can hold.
`blockCount`	• The number of non-empty blocks currently on the disk.
`recordCount`	• The total number of records stored on the disk.
`blocks`	• The list of blocks present on the disk.
`candidateBlocks`	• The list of blocks that become available for insertion after deletion.

Considering that the capacity of our `Disk` is **100 MB**, and the size of a `Block` is **400 Bytes**, a `Disk` would be able to hold at most **250,000 blocks**.

> **Note:** Since the `Nodes` of the B+ tree are stored in the main memory, they will not affect the `blockCount` of the `Disk`.

```java
public class Disk {
	public static final int maxBlockSize =
		(int) Math.floor(Config.DISK_CAPACITY / Config.BLOCK_SIZE);

	int blockCount;
	int recordCount;

	ArrayList<Block> blocks;
	Set<Integer> candidateBlocks;
}
```

Address

Since Java doesn't support pointers, we create an `Address` class to address this issue, which contains the following attributes:

Attribute	Description
`blockID`	• The ID of the block that the pointer is pointing to.
`offset`	• The offset of the record in the block.

To simplify our implementation, we chose not to utilise the getter and setter method approach within the `Address` class. Instead, we allow the attributes to be **directly accessed or modified** by using the `public` access modifiers.

```
public class Address {
	public int blockID;
	public int offset;
}
```

Records Packing

Dealing with Empty Fields

As there are missing values in the `game.txt` file, we have chosen to **exclude these rows** when reading the data since they hold no meaningful information and to reduce any potential impact on our subsequent experiments.

Non-separating

Since all records have a fixed length, we opted not to implement any record separation method as it is deemed unnecessary. Therefore, we chose to adopt a **non-separating** approach in our implementation.

Non-spanned

Since every record has a fixed size, and its size (37 bytes) is smaller than the size of a block (400 bytes), the spanned approach is not mandatory. We decided to pack the records into blocks in a **non-spanned** manner based on the following reasons:

Firstly, using the non-spanned approach, each record is confined to a single block, which simplifies the task of inserting, updating, or deleting records, reducing the complexity of the data management code.

Besides, with the non-spanned approach, we can directly locate and access a record without having to traverse multiple blocks. This reduces the number of blocks required to be accessed for finding one record, making it more efficient in terms of I/O operations and reducing the time taken.

Last but not least, the non-spanned approach is not only easier to implement but will also help us reduce the overall complexity of the data storage and retrieval system, which leads to fewer bugs, easier maintenance, and faster development.

Therefore, we chose to go with the **non-spanned** approach in our implementation.

Non-sequencing

We also decided to pack the records into blocks in a non-sequencing manner based on the following reasons:

Firstly, by using the non-sequencing approach, since the size of the record is fixed, we will be able to insert a record to any available slot in a block, which is faster compared to the sequencing approach where we have to find an appropriate slot to maintain the sequence.

Besides, when using the sequencing approach, we have to shift the records in order to maintain their order after each insertion, update or deletion, leading to larger overhead and introducing more complexity to the code.

Thus, we chose to go with **non-sequencing** in our implementation. This also implies that our indexes will be **non-clustered** indexes.

Blocks Organising

Contiguous Allocation

A database file means the collection of all the blocks storing a table. In our implementation, we opted to arrange the blocks in a contiguous manner to create a database file. We achieved this by utilising Java's `ArrayList` data structure, which arranges elements in a contiguous manner. The reasons are as follows:

Firstly, arranging the blocks contiguously eliminates the need to maintain pointers on each block. This allows us to potentially store more records in each block and simplifies the overall code complexity.

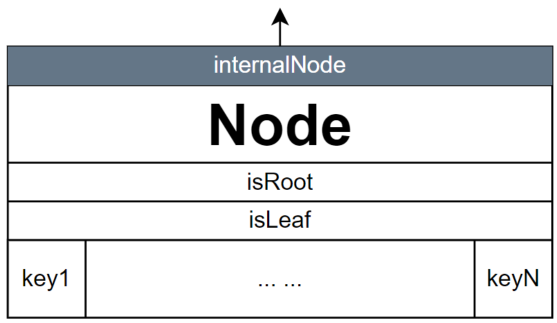
Furthermore, using the contiguous approach, the address of the k^{th} block of the file which starts at block b can easily be obtained as (b+k). This reduces the number of seeks and enhances the efficiency of I/O operations, significantly reducing the time taken to locate a block.

Thus, we decided to implement **contiguous allocation** for the blocks in our project.

Design and Implementation of B+ Tree Indexing Component

Our B+Tree design includes three crucial components: `Node`, `LeafNode`, and `InternalNode` in the relationship as shown below.

Node



Every `Node` has the subsequent four attributes:

Attribute Name	Description
`internalNode`	• Parent of this `Node` (if applicable) for easy traversal when updating the B+Tree.
`keysSet`	• ArrayList of float keys stored in the node.
`isRoot`	• Boolean flag for root indication.
`isLeaf`	• Boolean flag for leaf indication.

The `Node` class serves to initialise nodes, manage `Float` keys, and determine if the `Node` is a `Root` or a `Leaf`.

`internalNode` in the `Node` structure enables quick and efficient retrieval of the parent of this `Node`.

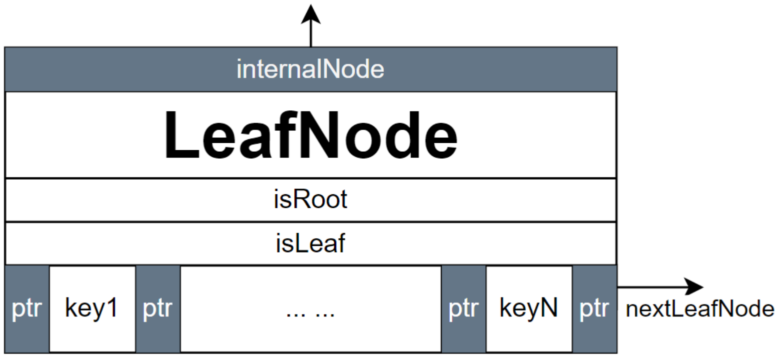
`isRoot` and `isLeaf` flag is introduced to efficiently identify if a `Node` is a root node or a leaf node.

Without `internalNode`, `isRoot` and `isLeaf`, huge overheads will incur as a result of traversing the B+ Tree for parent node, root node and leaf node identification during query, insertion and deletion.

Note: The memory cost for `internalNode`, `isRoot` and `isLeaf` in each `Node` is **considered** when calculating the maximum number of keys in each of the `Node`.

```
public class Node {
	private InternalNode internalNode;
	private ArrayList<Float> keysSet;
	private boolean isRoot;
	private boolean isLeaf;
}
```

LeafNode



Every `LeafNode` has the following attributes beside attributes that are inherited from `Node` class:

Attribute Name	Description
`addressesSet`	• ArrayList of "pointers" to the linked list of addresses of records with the same key.
`nextLeafNode`	• "Pointer" to the next `LeafNode`.

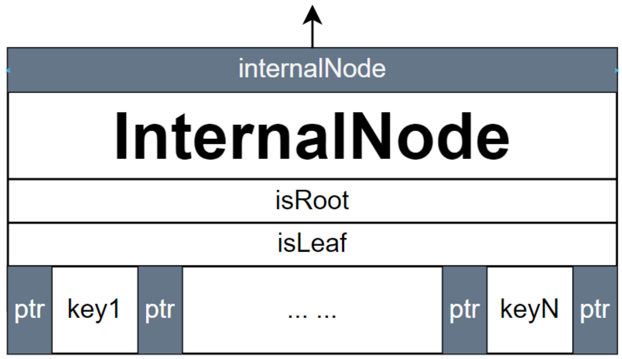
Inheritance concepts in Java are utilised such that we are able to reuse the methods implemented in the parent `Node` class while ensuring low coupling between `LeafNode` and `InternalNode`.

As this is class represents the leaf node, we initialise the `LeafNode` to have attributes `isRoot = false` and `isLeaf = true`.

> **Note:** Initialization of `LeafNode` is done under the assumption that `LeafNode` is strictly not the root of the B+ Tree. In the case where this `LeafNode` also serves as the root of the B+ Tree, `setIsRootNode` will be invoked to change the boolean value of `isRoot` to `true`.

```
public class LeafNode extends Node {
	private ArrayList<ArrayList<Address>> addressesSet;
	private LeafNode nextLeafNode;
}
```

InternalNode



Every `InternalNode` has the subsequent attribute beside the attributes that are inherited from the `Node` Class:

Attribute Name	Description
`childNodesSet`	• ArrayList of `Node` serving as "pointers" to its child nodes.

As Java does not employ pointers, we utilised the object references of `Node` class to represent the "pointers" to the child nodes.

Inheritance concepts in Java are utilised such that we are able to reuse the methods implemented in the parent `Node` class while ensuring low coupling between `LeafNode` and `InternalNode`.

As this class represents the non-leaf node, we initialise the `InternalNode` to have attributes `isRoot = false` and `isLeaf = false`.

> **Note:** Initialization of `InternalNode` is done under the assumption that this `InternalNode` is not the root of the B+ Tree. In the case where this `InternalNode` also serves as the root of the B+ Tree, `setIsRootNode` will be invoked to change the boolean value of `isRoot` to `true`.

```
public class InternalNode extends Node {
	private ArrayList<Node> childNodesSet;
}
```

BPTree

The `BPTree`, representing the B+ tree, contains the following attributes in our scenario:

Attribute	Explanation
`root`	• The root node.
`numLevels`	• The current height of the B+ tree.
`numNodes`	• The current total number of nodes in B+ tree.
`maxKeys`	• Other than keys and pointers, each node also stores two additional boolean flags and one extra pointer to the parent. • The size of the pointer is **8 bytes.** • The size of the key is **4 bytes.** • The size of boolean is **1 byte.** • The maximum number of keys is calculated as: $\lfloor 400 - 2\times8 - 2\times1 / (8+4) \rfloor$ = **31.**
`minInternalKeys`	• The minimum number of keys in internal node is: $\lfloor 31/2 \rfloor$ = **15**.
`minLeafKeys`	• The minimum number of keys in leaf node is: $\lfloor 32/2 \rfloor$ = **16**.

Note: The search key in a `Node` is based on the `FG_PCT_home` attribute.

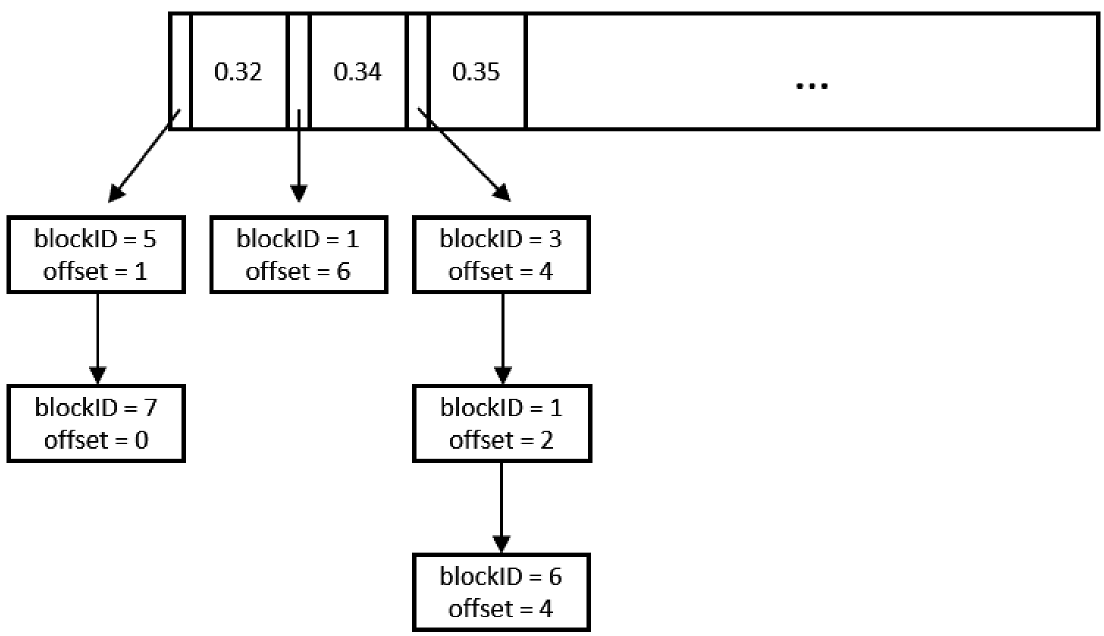
```java
public class BPTree {
    private static final int POINTER_SIZE = 8;
    private static final int KEY_SIZE = 4;
    private static final int BOOL_SIZE = 1;
    Node root;
    int numLevels;
    int numNodes;
    int maxKeys;
    int minInternalKeys;
    int minLeafKeys;

    public BPTree(int blkSize) {
        maxKeys = (blkSize - 2 * POINTER_SIZE - 2 * BOOL_SIZE) / (POINTER_SIZE +
KEY_SIZE);
        minInternalKeys = (int) Math.floor(maxKeys / 2);
        minLeafKeys = (int) Math.floor((maxKeys + 1) / 2);

        root = new LeafNode();
        numLevels = 1;
        numNodes = 1;
        root.setIsRootNode(true);

        numNodes = 0;

    }
}
```

Handling Duplicate Keys

As duplicate keys are allowed in the B+ tree for this project, we need to implement some extra mechanisms to handle this situation. For our group, we decided to handle records with duplicate keys by appending them in a linked list fashion. This approach simplifies the search process and reduces the number of B+ tree levels.

Instead of storing the addresses of the records in those leaf nodes, we store pointers to the linked list that contains the addresses of records with the same key. If there are records with the same key already stored in the B+ tree, we will simply append the new record at the end of the linked list for that key.

The following figure illustrates how we handle the duplicates in the leaf node:



In the figure above, when the search key is 0.35, we simply traverse down to the leaf node and scan the entire linked list associated with key 0.35 to retrieve addresses of all related records. For simplicity, when calculating the parameter n for B+ tree, we will only count **the cost for the pointer to the linked list (8 bytes)** instead of considering the cost of storing all elements in the linked list, assuming that the elements are stored elsewhere.

Experiments

Experiment 1

Store the data (which is about NBA games and described in Part 4) on the disk (as specified in Part 1) and report the following statistics:

- the number of records
- the size of a record
- the number of records stored in a block
- the number of blocks for storing the data

Results:

Requirement	Result
Number of records (Ignoring records with empty fields)	26552
Size of a record	37
Number of records stored in a block	10
Number of blocks for storing the data	2656

Terminal Output:

```
The number of records: 26552
The size of a record: 37
The number of records stored in a block: 10
The number of blocks for storing the data: 2656
```

Experiment 2

Build a B+ tree on the attribute "`FG_PCT_home`" by inserting the records sequentially and report the following statistics:

- the parameter n of the B+ tree
- the number of nodes of the B+ tree
- the number of levels of the B+ tree
- the content of the root node (only the keys)

Results:

Requirement	Result
Parameter n of the B+ tree	31
Number of nodes of the B+ tree	15
Number of levels of the B+ tree	2
Content of the root node (only the keys)	[0.304, 0.323, 0.349, 0.373, 0.398, 0.429, 0.453, 0.472, 0.488, 0.512, 0.529, 0.548, 0.581, 0.61, 0.632]

Terminal Output:

```
The parameter n of the B+ tree: 31
The number of nodes of the B+ tree: 15
The number of levels of the B+ tree: 2
The content of the root node (only the keys): [0.304, 0.323, 0.349, 0.373, 0.398, 0.429, 0.453,
 0.472, 0.488, 0.512, 0.529, 0.548, 0.581, 0.61, 0.632]
```

Experiment 3

Retrieve games with "FG_PCT_home" equal to 0.5 and report the following statistics:

- the number of index nodes the process accesses
- the number of data blocks the process accesses
- the average of "FG3_PCT_home" of the records that are returned
- the running time of the retrieval process (please specify the method you use for measuring the running time of a piece of code)
- the number of data blocks that would be accessed by a brute-force linear scan method (i.e., it scans the data blocks one by one) and its running time (for comparison)

Method for measuring running time:

We utilised `System.nanoTime()` to obtain the most accurate system timer reading in nanoseconds. To calculate the runtime, we subtract the time taken before the experiment begins from the time measured after the experiment has ended. We also converted the result to milliseconds for better readability.

Method for measuring data blocks accessed:

Since the B+ tree is stored in the main memory, we omitted it from our count of accessed data blocks. Moreover, we only increase the number of data blocks accessed if the i+1th record is located in a different block in comparison to the ith record.

Results:

Requirement	Result
Number of index nodes the process accesses	2
Number of data blocks the process accesses	732
Average of "FG3_PCT_home" of the records that are returned	0.391
Running time of the retrieval process	1 ms

Number of data blocks that would be accessed by a brute-force linear scan method and its running time	2656, 2 ms

Terminal Output:

```
B+ tree
-------------------------------------------------------------------
The number of index nodes accessed: 2
The number of data blocks accessed: 732
The average of "FG3_PCT_home" for Exp 3: 0.39120033
The running time of the retrieval process: 1 ms

Brute-force Linear Scan
-------------------------------------------------------------------
The number of data blocks accessed: 2656
The running time of the retrieval process: 2 ms
```

Experiment 4

Retrieve games with the attribute "`FG_PCT_home`" from 0.6 to 1, both inclusively and report the following statistics:

- the number of index nodes the process accesses
- the number of data blocks the process accesses
- the average of "`FG3_PCT_home`" of the records that are returned;
- the running time of the retrieval process
- the number of data blocks that would be accessed by a brute-force linear scan method (i.e., it scans the data blocks one by one) and its running time (for comparison)

Results:

Requirement	Result
Number of index nodes the process accesses	4
Number of data blocks the process accesses	238
Average of "FG3_PCT_home" of the records that are returned	0.5026
Running time of the retrieval process	0 ms
Number of data blocks that would be accessed by a brute-force linear scan method and its running time	2656, 1 ms

Terminal Output:

```
B+ tree
------------------------------------------------------------------
The number of index nodes accessed: 4
The number of data blocks accessed: 238
The average of "FG3_PCT_home" for Exp 4: 0.5025545
The running time of the retrieval process: 0 ms

Brute-force Linear Scan (Range)
------------------------------------------------------------------
The number of data blocks accessed: 2656
The running time of the retrieval process: 1 ms
```

Experiment 5

Delete games with the attribute "`FG_PCT_home`" below 0.35 inclusively, update the B+ tree accordingly, and report the following statistics:

- the number of nodes of the updated B+ tree
- the number of levels of the updated B+ tree
- the content of the root node of the updated B+ tree(only the keys)
- the running time of the process
- the number of data blocks that would be accessed by a brute-force linear scan method (i.e., it scans the data blocks one by one) and its running time (for comparison)

Results:

Requirement	Result
Number of nodes of the updated B+ Tree	12
Number of levels of the updated B+ Tree	2
Content of the root node of the updated B+ tree(only the keys)	[0.373, 0.398, 0.429, 0.453, 0.472, 0.488, 0.512, 0.529, 0.548, 0.581, 0.61, 0.632]
Running time of the deletion process	1 ms
Number of data blocks that would be accessed by a brute-force linear scan method and its running time	2656, 2 ms

Terminal Output:

```
The number of nodes of the B+ tree: 12
The number of levels of the B+ tree: 2
The content of the root node (only the keys): [0.373, 0.398, 0.429, 0.453, 0.472, 0.488, 0.512,
 0.529, 0.548, 0.581, 0.61, 0.632]
The running time of the deletion process is: 1 ms

Brute-force Linear Scan (Delete)
------------------------------------------------------------
The number of data blocks accessed: 2656
The running time of the deletion process is: 2 ms
```