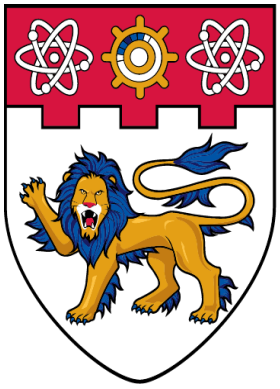

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

**Nanyang Technological University**

**CZ4031 Database System Principles**

**Assignment 1**

**Group 1**

**Chin Zhi Wei - U1821267H**

**Fong Kuo Xin Anthony - U1820958k**

**Luke Chow Tjun Mun - U1822337C**

**Nigel Lee - U1820706L**

# Introduction

## Description

The aim of this project is to demonstrate a B+ Tree implementation in Golang which supports searching, inserting and deleting operations.

## Project Overview

Our implementation consists of the following packages with the corresponding responsibilities:

- BlockManager: Serializes records into byte arrays and controls storage of serialized records.
- BpTree: Implementation of the B+ Tree data structure along with search, insertion and deletion operations done on the B+ Tree.
- Examples: Runs the test experiments according to the assignment, output appropriate metrics.

## Instructions

Clone the repository from Github or download from the zip file submitted. Follow the instructions specified in the README.md. Currently, this project support two ways of executing the source code:

1. Running from Windows executable (Recommended)
2. Running from Docker (Recommended)
3. Building the project manually
    a. Depending on your Operating System, install Go based on the instructions found on https://golang.org/doc/install#install.

Refer to the project README.md for more details on how to run the project. The source code can be found at https://github.com/ticklepoke/CZ4031.

## Storage Component

### BlockManager Structure

The internal structure of a block manager instance is shown in the following struct:

```
/blockmanager/common.go

type BlockManager struct {
      numBlocks    int
      numRecords   int
      blocks       []*[]byte
      blockSet     map[unsafe.Pointer]bool
      hasCapacity  bool
      currentCount int

      // contains the addresses of deleted records
      markedDeleted []*[]byte

      // BLOCKSIZE (100 or 500 bytes)
      BLOCKSIZE int
}
```

Figure 1: Structure of the Block Manager

Each block manager instance comprises of the following components:

- numBlocks: The number of blocks that have been initialized by the blockmanager.
- numRecords: The number of records that have been inserted into the blocks.
- blocks: A slice of the memory addresses of the blocks that have been initialized. This is used for measuring the number of blocks, and thus disk I/Os, that have been accessed during database operations.
- blockSet: A map with keys as addresses of blocks and values as booleans which are set to false initially. When accessing nodes, the respective values are updated to true to indicate a block being excessed.
- hasCapacity: A boolean that determines if a new block needs to be initialized during record insertion. This is set to false when the current block becomes full.
- currentCount: The offset for the current block. This is used to determine the memory address for the insertion of the next record.
- markedDeleted: A slice of pointers to memory addresses that once contained records but have since been deleted. When inserting new records, the block manager first inserts to the memory addresses in this slice before inserting it into the current block.
- blockSize: A constant that indicates the size of the blocks used in the database.

The block manager has the following methods:

- CreateBlock
- DeleteRecord
- PrintRecord
- InsertRecord

### Record Serialization

In order to ensure consistency in data size for each record, we serialize each record in order to store the record as a byte array. Each record attribute is stored at a fixed offset within the array, allowing for easier and faster retrieval subsequently.Furthermore, additional memory to indicate the size of each field is not required (commonly required in variable length fields).

```
const (
      // RECORDSIZE - number of bytes in a Record
      // longest character length for the following fields
      // tconst:        tt11285516   - 10
      // avgRating:     9.9          - 3
      // numVotes       2279223      - 7
      RECORDSIZE = 20

      // RATINGOFFSET - byte slice index for ratings
      RATINGOFFSET = 10

      // VOTESOFFSET - byte slice indes for votes
      VOTESOFFSET = 13
)
```

Figure 2: Maximum length of each attribute

In Figure 2, we can observe the maximum string lengths of each attribute. We use these maximum lengths to derive the offset for each of the attributes when being stored in a record.

```
// serialize fields with fixed offsets
func makeRecord(tconst string, avgRating string, numVotes string) Record {
      val := make([]byte, RECORDSIZE, RECORDSIZE)
      copy(val, []byte(tconst))
      copy(val[RATINGOFFSET:], []byte(avgRating))
      copy(val[VOTEOFFSET:], []byte(numVotes))

      return val
```

```
}
```

Figure 3: Serialization of each record

In Figure 3, we convert tconst, avgRating and numVotes into byte arrays. The 3 values are then concatenated and stored as a contiguous entry in memory.

### Block Storage

A few records are then stored together within a block. The records in the block are stored in an unclustered manner. This is done so for the following reasons:

1. Increase the speed of insertion and deletion of records
2. Ease of implementation of the unclustered index

```
// insert Record into current block if there is sufficient capacity
otherwise
// create and insert into new block
func (b *BlockManager) insertToBlock(newRecord Record) *[]byte {
	var target []byte

	if !b.hasCapacity {
		target = b.createBlock()
	} else {
		target = (*b.blocks[b.numBlocks-1])[b.currentCount*RECORDSIZE:]
		if b.currentCount*RECORDSIZE >= b.BLOCKSIZE-RECORDSIZE {
			b.hasCapacity = false
		}
	}
	copy(target, newRecord)
	b.currentCount++
	return &target
}
```

Figure 4: Logic to handle record storage within blocks

In figure 4, new records would be inserted into blocks. In the event that there is no capacity in existing blocks, a new block would be created.

## B+ Tree Component

### Node Structure

The internal structure of a B+ tree node is shown in the following Golang struct:

```
/bptree/common.go

type Node struct {
      Pointers     []interface{}
      TailPointers []interface{}
      Keys         []int
      Parent       *Node
      IsLeaf       bool
      NumKeys      int
      Next         *Node
}
```

Figure 5: Structure of each B+ tree node

Each node comprises the following components:

- Pointers: A slice of pointers which will hold pointers to either child nodes (in the case of internal nodes) or records (in the case of leaf nodes).
- TailPointers: A slice of pointers only used for the leaf nodes. Each tail pointer will keep track of the last inserted record, for easy insertion of new records in O(1) time. This will only be used for leaf nodes.
- Keys: A slice of keys that will aid in the traversal of the B+ tree.
- Parent: A pointer to the parent for easier node coalescing and splitting operations.
- IsLeaf: A boolean for convenience when checking if a node is a leaf.
- Next: A pointer to the adjacent neighbouring leaf node. This is only used for leaf nodes.

### Duplicate Records Linked List

In order to deal with the presence of multiple keys within the record dataset, we chose to append records with duplicate keys in a linked list fashion. When an insert operation is done, the appropriate insertion location will be found in the leaf node.

If there are no existing records with the same key, a new leaf node entry is made pointing to the record. If there are records with the same key already stored in the B+ Tree, the newly inserted record will be appended to the end of the existing records.

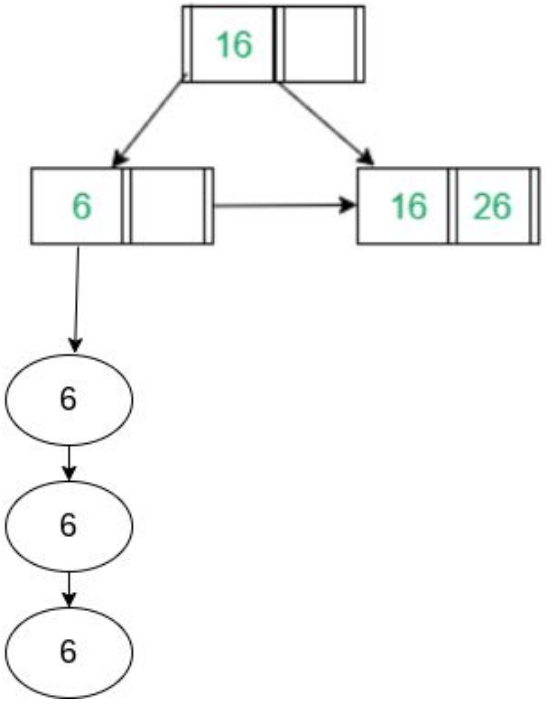


Figure 6: Illustration of linked list with B+ Tree

In figure 6, we append multiple records in a linked list manner from the first record. One advantage of this implementation is that all nodes within a linked list would contain the same key. A search query by key would return the entire linked list.

```
/bptree/common.go

type Record struct {
      Value *[]byte
      Next  *Record
}
```

Figure 5: Sample structure of each record

According to figure 5, each record would contain a serialized byte array consisting of the data for each record, and a pointer to the next record, to allow records to be stored in a linked list.

### Number of Keys

The default number of keys in the node is controlled by the package level variable:

```
/bptree/common.go

var (
      ...
      // N is the maximum number of children of an internal node
      N = 4
      …
)
```

Figure 7: Setting the number of keys as a global variable

The number of keys can be overridden by the bptree constructor:

```
/main.go

t := bptree.NewTree(6, 100)


/bptree/common.go

func NewTree(n, blocksize int) *Tree {
      N = n
      b := blockmanager.InitializeBlockManager(blocksize)
      return &Tree{BlckMngr: &b}
}
```

Figure 8: Setting the number of keys within the constructor

In our implementation, we used N to represent the maximum number of children of an internal node. This equates to:

- N-1 maximum number of keys in an internal node and leaf node
- N maximum number of pointers in an internal node and leaf node
- Floor(N-1 / 2) minimum number of keys in an internal node
- Floor(N + 1 / 2) minimum number of keys in a leaf node

# Experiments

## Experiment 1

We store the data on the disk and report the following statistics:

1) the number of blocks
2) the size of database

| Block Size | Number of Blocks | Size of Database |
|---|---|---|
| 100 B | 214064 | 21406.4 KB |
| 500 B | 42813 | 21406.5 KB |

## Experiment 2

We build a B+ tree on the attribute "averageRating" by inserting the records sequentially and report the following statistics:

1) the parameter n of the B+ tree
2) the number of nodes of the B+ tree
3) the height of the B+ tree, i.e., the number of levels of the B+ tree
4) the root node and its child nodes (actual content)

Please refer to experiment2.log for the complete logs.

| Block Size | n | Number of Nodes | Height of B+ tree | Contents of root nodes and its child nodes |
|---|---|---|---|---|
| 100 B | 4 | 46 | 3 | Refer to Figure 9 |
| 500 B | 4 | 46 | 3 | Refer to Figure 9 |

Note: The root of the B+ tree is of height 0

```
5.5 |

2.0 2.8 3.7 4.6 | 6.5 7.4 8.2 8.9 |

1.3 1.5 1.8 | 2.3 2.5 | 3.1 3.4 | 3.9 4.2 | 4.8 5.2 | 5.7 6.1 | 6.9 7.1 | 7.6 8.0 | 8.5 8.7 | 9.1 9.3 9.6 9.8 |

1.0 1.1 1.2 | 1.3 1.4 | 1.5 1.6 1.7 | 1.8 1.9 | 2.0 2.1 2.2 | 2.3 2.4 | 2.5 2.6 2.7 | 2.8 2.9 3.0 | 3.1 3.2 3.3 | 3.4 3.5 3.6 | 3.7 3.8 | 3.9 4.0 4.1 | 4.2 4.3 4.4 4.5 | 4.6 4.7 | 4.8 4.9 5.0 5.1 |
```

```
5.2 5.3 5.4  | 5.5 5.6  | 5.7 5.8 5.9 6.0  | 6.1 6.2 6.3 6.4  | 6.5 6.6 6.7 6.8  | 6.9 7.0  | 7.1 7.2 7.3 | 7.4 7.5  | 7.6 7.7 7.8 7.9  | 8.0 8.1  | 8.2 8.3 8.4  | 8.5 8.6  | 8.7 8.8  | 8.9 9.0  | 9.1 9.2  | 9.3 9.4 9.5  | 9.6 9.7  | 9.8 9.9 10.0  |
```

Figure 9: B+ tree contents after insertion when block size is 100 B

## Experiment 3

We retrieve the attribute “tconst” of those movies with the “averageRating” equal to 8 and report the following statistics:

1) the number and the content of index nodes the process accesses
2) the number and the content of data blocks the process accesses
3) the attribute “tconst” of the records that are returned

| Block Size | 100 B | 500 B |
|---|---|---|
| Number of Index Nodes accessed | 4 | 4 |
| Number of Data blocks accessed | 24204 | 19797 |
| Content of Data blocks accessed | Refer to experiment3.log | Refer to experiment3.log |
| Attribute “tconst” of records returned | Refer to experiment3.log | Refer to experiment3.log |

Please refer to experiment3.log for the content of blocks accessed and the “tconst”s of the records returned

```
Index Node 1 [[5.500000]]
Index Node 2 [[6.500000 7.400000, 8.200000 8.900000]]
Index Node 3 [[7.600000 8.000000]]
Leaf Node 4 [[8.000000 8.100000]]
```

Figure 10: Content of nodes accessed with block size set to 100 and 500B

## Experiment 4

We retrieve the attribute “tconst” of those movies with the attribute “averageRating” from 7 to 9, both inclusively and report the following statistics:

1) the number and the content of index nodes the process accesses
2) the number and the content of data blocks the process accesses

3) the attribute “tconst” of the records that are returned

| Block Size | 100 B | 500 B |
|---|---|---|
| Number of Index Nodes accessed | 16 | 16 |
| Number of Data blocks accessed | 184415 | 42252 |
| Content of Data blocks accessed | Refer to experiment4.log | Refer to experiment4.log |
| Attribute “tconst” of records returned | Refer to experiment4.log | Refer to experiment4.log |

```
Index Node 1 [[5.500000]]
Index Node 2 [[6.500000 7.400000 8.200000 8.900000]]
Index Node 3 [[6.900000 7.100000]]
Leaf Node 4 [[6.900000 7.000000]]
Leaf Node 5 [7.100000 7.200000 7.300000]
Leaf Node 6 [7.400000 7.500000]
Leaf Node 7 [7.600000 7.700000 7.800000 7.900000]
Leaf Node 8 [8.000000 8.100000]
Leaf Node 9 [8.200000 8.300000 8.400000]
Leaf Node 10 [8.500000 8.600000]
Leaf Node 11 [8.700000 8.800000]
Leaf Node 12 [8.900000 9.000000]
Leaf Node 13 [9.100000 9.200000]
Leaf Node 14 [9.300000 9.400000 9.500000]
Leaf Node 15 [9.600000 9.700000]
Leaf Node 16 [9.800000 9.900000 10.000000]
```

Figure 11: Content of nodes accessed with block size set to 100 and 500B

## Experiment 5

We delete those movies with the attribute “averageRating” equal to 7, update the B+ tree accordingly, and report the following statistics:

1) the number of times that a node is deleted (or two nodes are merged) during the process of the updating the B+ tree
2) the number nodes of the updated B+ tree
3) the height of the updated B+ tree
4) the root node and its child nodes of the updated B+ tree

| Block Size | Number of times node is deleted | Number of Nodes | Height of B+ tree | Contents of root nodes and its child nodes |
|---|---|---|---|---|
| 100 B | 0 | 46 | 3 | Refer to experiment5.log |
| 500 B | 0 | 46 | 3 | Refer to experiment5.log |

```
5.5 |

2.0 2.8 3.7 4.6 | 6.5 7.4 8.2 8.9 |

1.3 1.5 1.8 | 2.3 2.5 | 3.1 3.4 | 3.9 4.2 | 4.8 5.2 | 5.7 6.1 | 6.8 7.1 | 7.6 8.0 | 8.5 8.7 | 9.1 9.3 9.6 9.8 |

1.0 1.1 1.2 | 1.3 1.4 | 1.5 1.6 1.7 | 1.8 1.9 | 2.0 2.1 2.2 | 2.3 2.4 | 2.5 2.6 2.7 | 2.8 2.9 3.0 | 3.1 3.2 3.3 | 3.4 3.5 3.6 | 3.7 3.8 | 3.9 4.0 4.1 | 4.2 4.3 4.4 4.5 | 4.6 4.7 | 4.8 4.9 5.0 5.1 | 5.2 5.3 5.4 | 5.5 5.6 | 5.7 5.8 5.9 6.0 | 6.1 6.2 6.3 6.4 | 6.5 6.6 6.7 | 6.8 6.9 | 7.1 7.2 7.3 | 7.4 7.5 | 7.6 7.7 7.8 7.9 | 8.0 8.1 | 8.2 8.3 8.4 | 8.5 8.6 | 8.7 8.8 | 8.9 9.0 | 9.1 9.2 | 9.3 9.4 9.5 | 9.6 9.7 | 9.8 9.9 10.0 |
```

Figure 12: B+ Tree contents after deletion