NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

# CZ4031: Database System Principles

Group 8

Project 1 Report

| Name | Contributions |
|---|---|
| CHIU WEI JIE, REEVES | Experiment 4, Report |
| GUCON NAILAH GINYLLE PABILONIA | BPlusTree, Experiment 1 & 2, Report |
| IVAN PUA JUN HAO | Experiment 5, Report |
| NUR DILAH BINTE ZAINI | Storage, Experiment 3, Report |

**SCHOOL OF COMPUTER SCIENCE AND ENGINEERING**
**NANYANG TECHNOLOGICAL UNIVERSITY**

## Table of Content

## Introduction

### Project Description

The goal of this project is to design and implement the storage and B+ Tree indexing components of a database management system, and incorporate operations such as inserting, deleting, and searching for certain indexes. The chosen language for development is **Java**.

### Installation Guide

Do follow the steps in the Installation Guide.pdf found in the same directory to download our chosen IDE and set up your environment.

## Design and Implementation of Storage Components

The main functionality of the disk is for storing the given dataset in a structured manner. Data is read from the file in sequential order, and inserted into the disk as records. Each record is stored in a block, and the disk contains several of such blocks.

### Record

The dataset contains entries pertaining to three distinct fields:

| Field | Datatype | # of Bytes | Reason for choice |
|---|---|---|---|
| `tconst` | String | 20 | • Fixed size of 10 characters<br>• Each char in Java takes 2 bytes<br>• Total bytes: 2 x 10 = **20 bytes** |
| `averageRatings` | Float | 4 | • Contains decimal of up to 1 dp |
| `numVotes` | Integer | 4 | • Value does not exceed max int value of 2147483647 |
| Total size of record: 20 + 4 + 4 = **28 bytes** | | | |

These fields will be stored in our database as a `Record`. When packing our fields into records, we decided to adopt a **fixed format with fixed length** to make it easier to interpret for operations such as  insertion.

```
public Record(String tconst, float averageRating, int numVotes) {
        this.tconst = tconst;
        this.averageRating = averageRating;
        this.numVotes = numVotes;
}
```

## Block

Each `Block` contains the following attributes:

| Attribute | Explanation |
|---|---|
| `currRecords` | • Store the current number of records in block<br>• Initialised to 0 when new block is created |
| `maxRecords` | • Maximum number of records that can be stored within a block |
| `data` | • To store the actual data of the record upon block initialisation |

**Note:** `Block` used to store records only (Keys are stored in blocks called `Nodes`)

```
public Block(int size){
        this.currRecords = 0;
        this.maxRecords = (int) Math.floor(size / Record.size());
        this.data = new Record[maxRecords];
}
```

We decided to use a **non-clustering index** to help increase efficiency when doing data retrieval from the database.

## Disk

Each `Disk` contains the following attributes:

| Attribute | Explanation |
|---|---|
| `sizeOfDisk` | • Size of Disk |
| `blkSize` | • Size of block (200B) |
| `maxBlkSize` | • Maximum number of blocks that can be created in the disk |
| `blkList` | • List of blocks |
| `countOfRecords` | • Number of records in the disk |

The disk is initialized with the required attributes and functions are set up here to aid in the operations of experiments for display, retrieval, insertion and deletion.

```java
public Disk(int blkSize) {
        this.sizeOfDisk = DISK_SIZE;
        this.blkSize = blkSize;
        this.maxBlkSize = (int) Math.floor(DISK_SIZE / blkSize);
        this.blkList = new ArrayList<>();
        this.countOfRecords = 0;
}
```

## Address

Each `Address` contains the following attributes:

| Attribute | Explanation |
|---|---|
| `blkId` | ● ID of the block |
| `offset` | ● Identify a particular address within a block |

The address is to obtain the ID of the data block given the offset.

```java
public Address(int blkId, int offset) {
        this.blkId = blkId;
        this.offset = offset;
}
```

## Insertion of Records into Blocks

This section specifies our choices for various options (e.g. Spanned or non-spanned; sequencing vs non-sequencing) and our justifications.

### Non - spanned

We decided to insert the records into blocks in a non-spanned manner. It will be an easier approach and we will be able to limit the number of block accesses. We are required to access only 1 block for a single record. Since the size of the record (28 bytes) is smaller than the size of the block (200 bytes), there will be sufficient space to store at most 7 records in a block.

If the choice was spanned, some records will be spanned over 2 blocks. Hence, we would be required to access 2 blocks for a single spanned record. The number of blocks accessed for the spanned records will be higher than the number of blocks accessed for the unspanned records. We will be required to search for another block after retrieving the 1st half of a record in a block to get the whole record. The implementation will be difficult as we will have to keep track of 2 blocks for a spanned record.

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

### Non - sequencing

Record insertion will be easier as we will be able to insert a record either at the end of the block or an empty slot in a block. The sequence of the records are not required to be maintained. Since the size of the record is of fixed length, we can easily insert new records in any available slots in a block.

Therefore, we chose to go with non-sequencing in our implementation.

## Design and Implementation of B+ Tree Indexing Component

### Node

Each `Node` contains the following attributes:

| Attribute | Explanation |
|---|---|
| `internalNode` | • The Internal Node aka the parent of node (if applicable) |
| `keys` | • An ArrayList is created and used to store the keys of the node |
| `isRootNode` | • Indicates if the node is a Root Node or not |
| `isLeafNode` | • Indicates if the node is a Leaf Node or not |

The `Node` class is used to initialise nodes, key maintenance and identify if a particular value is a Root Node or Leaf Node.

```
public class Node {
    private InternalNode internalNode;
    private ArrayList<Integer> keys;
    private boolean isRootNode;
    private boolean isLeafNode;

    public Node(){
        isRootNode = true;
        isLeafNode = true;
        keys = new ArrayList<>();
    }
    ...
}
```

## LeafNode

Each `LeafNode` contains the following attributes:

| Attribute | Explanation |
|---|---|
| `addresses` | • The "pointers" to the record addresses |
| `nextNode` | • The "pointer" to the next Leaf Node |

Since Java does not make use of pointers, we leverage on the concept of inheritance. Hence, we are able to use functions such as setIsRootNode, setIsLeafNode, etc. that were declared in the `Node` class.

We initialise the Leaf Node to have the attributes isRootNode to false and isLeafNode to true.

```java
public class LeafNode extends Node {
    private ArrayList<Address> addresses;
    private LeafNode nextNode;

    public LeafNode(){
        super();
        setIsRootNode(false);
        setIsLeafNode(true);
        addresses = new ArrayList<Address>();
        setNextNode(null);
    }
    ...
}
```

## InternalNode

Each `InternalNode` contains the following attributes:

| Attribute | Explanation |
|---|---|
| `childNodes` | • The “pointers” to its child nodes |

Since Java does not make use of pointers, we leverage on the concept of inheritance. Hence, we are able to use functions such as setIsRootNode, setIsLeafNode, etc. that were declared in the `Node` class.

Since it is an Internal Node, isLeafNode will be set to false and isRootNode will be set to false.

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

    public InternalNode() {
        super();
        this.setIsLeafNode(false);
        this.setIsRootNode(false);
        childNodes = new ArrayList<Node>();
    }
    ...
}
```

## B+ Tree

The `B+ Tree` contains the following attributes in our scenario:

| Attribute | Explanation |
|---|---|
| `maxNoOfKeys` | • Maximum number of keys is **19** given the size of the pointer is 6 bytes and size of the key is 4 bytes |
| `minNoOfInternalKeys` | • Minimum number of keys in internal node is **9** |
| `minNoOfLeafKeys` | • Minimum number of keys in leaf node is **10** (1 additional pointer point to the next lead node) |
| `noOfNodes` | • Total number of nodes in B+ tree |
| `noOfNodesDeleted` | • Total number of nodes deleted in B+ tree (due to merging of nodes operations) |

**Note:** The search key in a node is based on the `numVotes` attribute.

```java
public class BPTree {
    private static final int POINTER_SIZE = 6;
    private static final int KEY_SIZE = 4;
    Node root;
    int noOfLevels;
    int noOfNodes;
    int noOfNodesDeleted;
    int maxNoOfKeys;
    int minNoOfInternalKeys;
    int minNoOfLeafKeys;

    public BPTree(int sizeOfBlock) {
        maxNoOfKeys = (sizeOfBlock - POINTER_SIZE) / (POINTER_SIZE + KEY_SIZE);
        minNoOfInternalKeys = (int) Math.floor(maxNoOfKeys / 2);
        minNoOfLeafKeys = (int) Math.floor((maxNoOfKeys + 1) / 2);

        root = createRoot();
        noOfNodes = 0;
        noOfNodesDeleted = 0;
    }
    ...
}
```

## Handling Duplicates: Searching keys in B+ tree

Since duplicate keys are allowed in the B+ tree, the duplicated keys are grouped together due to the sorting of keys mechanism in B+ tree. The duplicated keys may span across a multiple of nodes which is shown below:

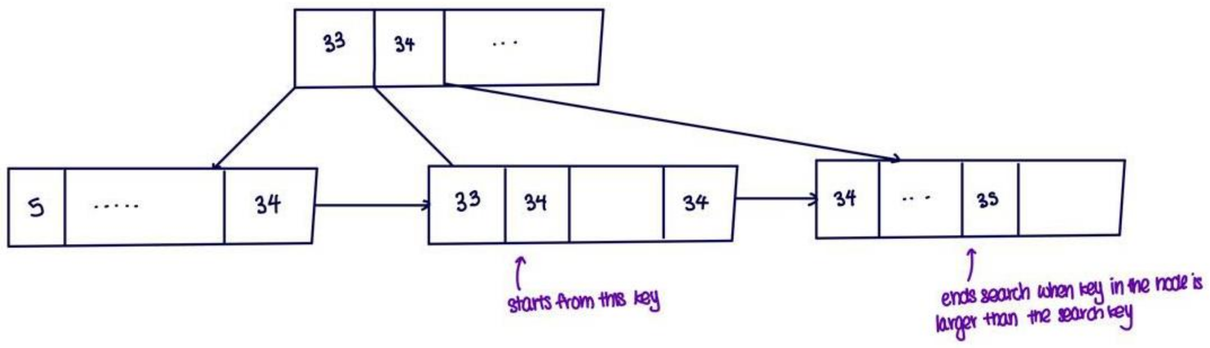
33
34
...
5
.....
34
33
34
34
34
...
35
starts from this key
ends search when key in the node is larger than the search key

Figure 1: Diagram of how our B+ tree handles key search

In figure 1, given the search key is 34, we will retrieve the most left leaf node that contains 34 as the search key. Afterwards, we will scan through each key in the leaf node and retrieve those keys equal to 34. We will then check the next leaf node and will retrieve those keys equal to search key 34 till a key in the leaf node that is larger than the search key is met.

## Experiments

### Experiment 1

Store the data (which is about IMDb movies 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 Table:**

| Requirement | Output |
|---|---|
| Number of Records | 1070318 |
| Size of a Record | 28 |
| Number of Records stored in a Block | 7 |
| Number of Blocks for storing the data | 152903 |

**Terminal Output:**

```
Running Experiment 1...
Num of records: 1070318
Size of a record: 28
Num of records stored in a block: 7
Num of blocks for storing the data: 152903
```

## Experiment 2

Build a B+ tree on the attribute "numVotes" 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 Table:**

| Requirement | Output |
|---|---|
| Parameter n of B+ tree | 19 |
| Number of nodes of B+ tree | 115361 |
| Number of Levels of B+ tree | 6 |
| Content of the root node (only the keys) | [7, 10, 27] |

**Terminal Output:**

```
Running Experiment 2...
The parameter n of the B+ tree: 19
The No of nodes of the B+ tree: 115361
The No of levels of the B+ tree: 6
The content of the root node (only the keys):
[7, 10, 27]
```

## Experiment 3

Retrieve those movies with the “numVotes” equal to 500 and report the following statistics:

- the number of index nodes the process accesses
- the number of data blocks the process accesses
- the average of “averageRating’s” 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:**
Made use of System.nanoTime() method to return the most precise system timer in nanoseconds.
The runtime is obtained by taking the current time (after experiment function has ran) and subtracting the startTime (before the experiment runs) and dividing the output by 1000000 to
obtain the ms (better timing readability)

**Results Table:**

| Requirement | Output |
|---|---|
| Number of Index Nodes the process accesses | 17 |
| Number of Data Blocks the process accesses | 127 |
| Average of “averageRating’s” of the records that are returned | 6.731818214329806 |
| Running time of the retrieval process | 3 ms |
| Number of data blocks that would be accessed by a brute-force linear scan method and its running time | 152903, 59 ms |

**Terminal Output:**

```
Running Experiment 3...

B+ tree
------------------------------------------------------------------
Total no of index nodes accesses: 17
Total no of data block accesses: 127
The running time of the retrieval process is 3 ms
The average rating of the records that numVotes = 500 is
6.731818214329806

Brute-force Linear Scan
------------------------------------------------------------------
Total no of data block accesses (brute-force linear scan method):
152903
The running time of the retrieval process (brute-force linear scan
method) is 59 ms
The average rating of the records that numVotes = 500 (brute-force
linear scan method) is 6.731818214329806
```

## Experiment 4

Retrieve those movies with the attribute “numVotes” from 30,000 to 40,000, 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 “averageRating’s” 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 Table:**

| Requirement | Output |
|---|---|
| Number of Index Nodes the process accesses | 75 |
| Number of Data Blocks the process accesses | 1028 |
| Average of “averageRating’s” of the records that are returned | 6.727911862221244 |
| Running time of the retrieval process | 14 ms |
| Number of data blocks that would be accessed by a brute-force linear scan method | 152903 |
| Running time using brute-force linear scan | 51 ms |
| Average of “averageRating’s” of brute-force linear scan | 6.727911862221244 |

**Terminal Output:**

```
Running Experiment 4...

B+ tree
------------------------------------------------------------------
Total no of index nodes accesses: 75
Total no of data block accesses: 1028
The running time of the retrieval process is 14 ms
The average rating of the records where numVotes from 30000 - 40000
is 6.727911862221244

Brute-force Range Linear Scan
------------------------------------------------------------------
Total no of data block accesses (brute-force linear scan method):
152903
The running time of the retrieval process (brute-force linear scan
method) is 51 ms
The average rating of the records where numVotes from 30000 - 40000
using (brute-force linear scan method) is 6.727911862221244
```

## Experiment 5

Delete those movies with the attribute “numVotes” equal to 1,000, update the B+ tree accordingly, and report the following statistics:

- the number 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 Table:**

| Requirement | Output |
|---|---|
| Number of Nodes of the updated B+ Tree | 115357 |
| Number of Levels of the updated B+ Tree | 6 |
| Content of the Root Node of the updated B+ tree(only the keys) | [7, 10, 27] |
| Running time of the process | 2 ms |
| Total no of data blocks that would be accessed by a brute-force linear scan method and its running time | 152903, 56 ms |
| Total no of data block accessed to **delete a record** by a brute-force linear scan method | 42 |

**Assumption made:**
It was not explicitly mentioned in the question whether “number of data blocks that would be
accessed by a brute-force linear scan method” refers to <u>all</u> accesses to the data blocks **OR** total number of data blocks accessed to <u>delete a record</u>. Hence, both values are displayed in the table above and the terminal output below.

**Terminal Output:**

```
Running Experiment 5...

B+ tree
------------------------------------------------------------------
No of nodes deleted: 4
The running time of the deletion process is 2 ms
The parameter n of the B+ tree: 19
The No of nodes of the B+ tree: 115357
The No of levels of the B+ tree: 6
The content of the root node (only the keys):
[7, 10, 27]

Brute-force Linear Scan
------------------------------------------------------------------
Total no of data block accesses (brute-force linear scan method):
152903
Total no of data block accessed to delete a record (brute-force
linear scan method): 42
The running time of the deletion process is (brute-force linear scan
method) is 56 ms
```