Nanyang Technological University

CZ4031 Database System Principles

Assignment 1

Prepared by:

Name	Matriculation Number
Au Yew Rong Roydon	U2021424J
Ahmad Syafiq Bin Ahmad Ghozali	U2022711K
Damien Lee Wen Hao	U2022328A
Chen Wei Yi	U2021480L
Ong Xin Rui Celestine	U2022737G

Introduction

Project Overview:

The aim of this project is to implement a B+ Tree in C++ which supports searching (both search query and range queries), insertion and deletion operations.

Our implementation consists of the following classes:

1. BlockManager: Manages the creation and deletion of new blocks. Handles all the logic related to blocks.
2. Record class: Holds the fields of a record.
3. B+ Tree: Handles the logic of search, insertion and deletion operations of nodes in the tree.
4. Block classes: Consists of B+ tree nodes, recordBlocks and linkedList Blocks

Before diving into the experiments, we shall first explain the attributes and functions in the classes.

****How to run the project**:**

We have included the guide in the readme.

To install, run a C++ compiler with minimum C++11 support to compile main.cpp, then run the produced binary. Assuming g++, run the following command: **g++ main.cpp -O2 -o main**

To run: **.\main.exe**

In the case that some middle output disappears in the terminal (Happened sometimes in vsc terminal, could be due to too many information to print out), do output it out into a text file. That way the full output always comes out correctly. (Use .\main.exe > someTxtFile.txt)

Contributions:

Coding and report writing: **Everyone contributed equally.**

Storage

BlockManager Class:

```
class BlockManager{
private:
    std::queue<unsigned int> deletedIndex;

    std::vector <block *> blockPtrArray;

public:
    const int blkSize = BLOCK_SIZE;
    const char recordsPerBlock = NUM_RECORDS;
    const char keyPerIndexBlock = NUM_KEY_INDEX;
    const char keyPerLinkedList = NUM_LINKED_LIST;
    unsigned long numStorageBlocks = 0;
    unsigned long numTreeBlocks = 0;
    unsigned int getSize() const;
    unsigned int getNumBlocks() const;
    void deleteBlock(unsigned int loc);
    //unsigned int createBlocks(unsigned int numBlocks);
    //unsigned int createBlock();
    unsigned int createRecordBlock();
    unsigned int createIndexBlock();
    unsigned int createLinkedListBlock();
    unsigned int createRecordBlocks(unsigned int numBlocks);
    //unsigned int createIndexBlocks(unsigned int numBlocks);
    block * accessBlock(unsigned int index);
    block * noLogAccessBlock(unsigned int index) const;
    void clearAccessed();

    //Helper functions
    void printRecordBlock(unsigned int recordBlockIndex);

    std::unordered_set<unsigned int> accessedDataBlocks, accessedTreeBlocks;

    std::vector<unsigned int> firstData, firstTree;
};
```

Figure 1 Block Manager

Each Block Manager class comprise of the following attributes and functions:

<u>Attributes:</u>

- blkSize: The size of the blk in bytes (default 200 bytes).
- recordsPerBlock: The number of records in a block (dependant on the block size).
- keyPerIndexBlock: Maximum number of keys in a tree node.
- keyPerLinkedList: Maximum number of record addresses in a LinkedListBlock.
- numStorageBlocks: Total number of storage blocks.
- numTreeBlocks: Total number of tree node blocks + LinkedList Blocks. This also represents the size of the B+ tree.

- deletedIndex: A queue that contains the index of deleted blocks for tracking of soft delete. In the case of creating a new block, the index would be used.
- blockPtrArray: Contains an array of addresses of blocks that have been initialized. The length of this array corresponds to the total number of blocks.
- accessedDataBlocks: An unordered set that contains unique datablocks accessed. Used during search to return searchIO.
- accessedTreeBlocks: An unordered set that contains unique tree node/linkedList blocks accessed. This is used during our search to return searchIO.
- firstData: First 5 unique datablocks accessed.
- firstTree: First 5 unique tree node/LinkedList Blocks accessed.

Functions:

- getSize(): Gets the total size of the B+ tree (number of blocks times block size).
- getNumBlocks(): Gets the total number of blocks.
- deleteBlock(): Deletes the block from memory and stores the position of the block in the blockPtrArray (index) in the deletedIndex queue. This means that the block manager does not delete and shift the other blocks, but rather it empties the deleted block to avoid invalidating all pointers pointing to subsequent blocks. To optimise the usage of space, should a new block be requested for later on, the deleted blocks will be checked and used, if there is any.
- createRecordBlock(), createIndexBlock(), createLinkedListBlock(): Creates a new record/node/linkedList block respectively by first checking if there are any deletedIndex. If present, it will insert at the first deleted index, else it would insert at the end of the blockPtrArray.
- accessBlock(): Returns the address of a particular block (pointer).
- clearAccessed(): Clears the accessedDataBlocks, accessedTreeBlocks, firstData and firstTree.
- printRecordBlock(): Prints all records stored in a RecordBlock.

Block manager will construct the respective blocks it is asked for and a pointer for the block in an array. It will return the index of the array. If given an array, it is able to access the respective block and return the pointer. Block manager manipulates the returned index by shifting them such that 0 is not a valid index, this allows us to use the value 0 as an empty pointer rather than a pointer pointing to the index 0.

Block manager also stores the IOs used since the last time it was cleared. It does so by inserting every accessed block index to an undordered_set, 1 for the records block and 1 for the linked list and index blocks. The unordered_set simply stores unique values, and allows us to count the number of block I/O by checking the size. We also store the first 5 unique records blocks, and first 5 unique index and linked list blocks accessed, in an array in the order they were accessed, allowing us to later check them to find what were the blocks accessed and print their content.

Records Class:

We utilized record serialization to ensure that all records would be of the same length. This is to improve the simplicity of storing the records in the blocks. Since each record is stored at a fixed offset, this allows for easy and fast retrievals of records. Additional memory to indicate the length of variable length fields is also not required. To ensure that not too much space is wasted for padding, we have also checked that there is very low variability between the longest and shortest data of each field.

Calculations for fixed length:

We first looked at the maximum values of each field in the records. This helps determine the maximum size required.

```
print("Max of tconst:", max(data["tconst"]))
print("Max of averageRating:",max(data["averageRating"]))
print("Max of numVotes:",max(data["numVotes"]))
```

```
Max of tconst: tt9916778
Max of averageRating: 10.0
Max of numVotes: 2279223
```

Figure 2 Maximum Values

Each record would then consist of the following:

```cpp
struct fixedPoint{
    unsigned char ones = 0, decimal = 0;
};
```

Figure 3 Fixed Point

```cpp
struct Record{
public:
    fixedPoint avgRating;
    unsigned int numVotes = 0;
    std::string getTconst(){
        char buf[11];
        strncpy(buf, this->tconst, 10);
        buf[10] = '\0';
        return std::string(buf);
    }

    void setTconst(std::string input){
        memset(tconst,0,10);
        strncpy(tconst,input.c_str(),10);
    }

    void printRecord(){
        std::cout<<"----Record details----"<<std::endl;

        printf("Average rating is: %.1f\n",getAverageRating());
        std::cout<<"Num votes is: "<<numVotes<<std::endl;
        std::cout<<"Tconst is: "<<getTconst()<<"\n"<<std::endl;

    }

    //To be used for calculation
    float getAverageRating(){
        return float(avgRating.ones) + float(avgRating.decimal)/10;
    }

private:
    char tconst[10];
};
```

Figure 4 Record Structure

Tconst is represented as a 10 char string as 10 characters is the longest maximum length. We will not be storing the null character for Tconst smaller than 10 characters. **Hence 10 bytes**. avgRatings can be represented by 1 byte for the digit and another for the decimal place. **Total 2 bytes.** numVotes is an unsigned integer, giving us a maximum of 4 trillion. **Total ~ 4 bytes/**

These are represented with the smallest amount of bytes, but at the same time minimising the chances of multiple attributes having to share a byte. This gives us a record size of **16 bytes**. For a block size of **200 bytes** this gives us **200/16 = 12 records per block**, with **8 bytes left over** (for **500** it would be **500/16 = 31 records per block**).

Record Storage in blocks:
Multiple records are then stored together within a block. The records are unsorted and would be stored into the recordsBlock.

Since the size of the node would be similar to the block, we utilized inheritance for the structure of our blocks. All recordBlocks, nodes and linkedList Block would inherit from a superclass called block. Each block contains an unsigned char type that indicates what type of block it is. In Figure 5, it can be seem that we have assigned type 0 for a record block, 1 for non-leaf node, 2 for leaf node, and 3 is for the linked list pointer block, which stores the duplicate keys for each key in the tree.

```
class block{
public:
    unsigned char type; // Check this value before casting your block into the derived classes.
    // 0: recordBlock, 1: tree non-leaf, 2: tree leaf, 3: linked list pointer block
};
```

Figure 5 Block Type

```
#define BLOCK_SIZE 500//200
#define NUM_RECORDS (BLOCK_SIZE-1)/16
#define NUM_KEY_INDEX (BLOCK_SIZE-4-4)/8
#define NUM_LINKED_LIST (BLOCK_SIZE-1-4)/4
```

Figure 6 Definition of terms

To calculate the number of entries we have in each type of blocks we do the following calculations:

- A **record block** stores additionally the type of block, therefore we can store $\frac{blockSize-1}{16}$ records since each record takes up 16 bytes. For 200B this is 12 records, for 500B this is 31 records
- A **treeNodeBlock** stores additionally the type of block, the block index of the parent node, 1 additional pointer on top of $\frac{blockSize-4-1-3}{8}$ key-pointer pairs which are 8 bytes large. For 200B this is 24 keys and 25 pointers, for 500B this is 61 keys and 62 pointers.
- A **linkedListBlock** stores the type of block and the pointer to the next block, hence we can store $\frac{blockSize-1-4}{4}$ pointers which are 4 bytes large. For 200B this is 48 pointers, for 500B this is 123 pointers.

Record Blocks:

This is our implementation of the record block to store our data

```
struct RecordBlock: public block{
    std::array<Record, NUM_RECORDS> records; //Can use records.size()

    char padding[BLOCK_SIZE-(NUM_RECORDS*16+1)]; //Currently there for padding purposes, we can use this for something else later

};
```

Figure 7 Record Block

B+ Tree

Structure

When considering the design of our B+ Tree and the structure of our nodes, we first did some checks for the number of duplicates of our search keys.

We plotted the logarithm of numVotes frequency to observe the frequency of duplicate numVotes and realized that there is a maximum frequency of 81495. With such a high number of duplicate, we had to think of a way to reduce the size of the B+ tree as a frequency of 81495 would mean that duplicate keys would be stretched across multiple nodes.

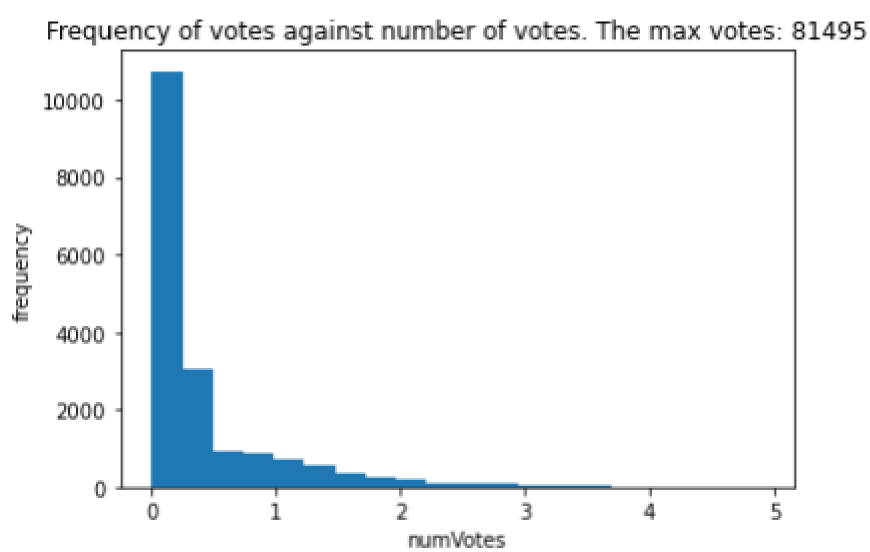

Figure 8 Frequency numVotes

Hence, we decided to proceed with the structure below which incorporates the idea of a normal B+ tree and a linkedList block. The idea is as follows: Unique keys will be stored in the B+ tree as per lecture definitions. However, when it comes to a duplicate key, we would store the address of the duplicate keys in a linkedList block. With the linkedList block only needing to store the record addresses of the duplicate keys and not needing space for the search key (Since all search keys are the same), more space in the block can be utilised to store the record addresses).

This structure would hence allow us to have **1) more number of record addresses to be stored since the search key need not be stored in linkedList blocks, 2) Lesser restructuring of the tree since deletion and insertion of duplicate keys would only affect the linked list blocks which are linked by pointers.** Keys that do not have duplicates would still be as per usual where the pointer in the leaf node points straight to the record. **We do note that search time might be longer due to the LinkedList Structure, but the space is better optimized and insertions and deletions are made easier, since there is lesser restructuring of the tree** (Figure 8).

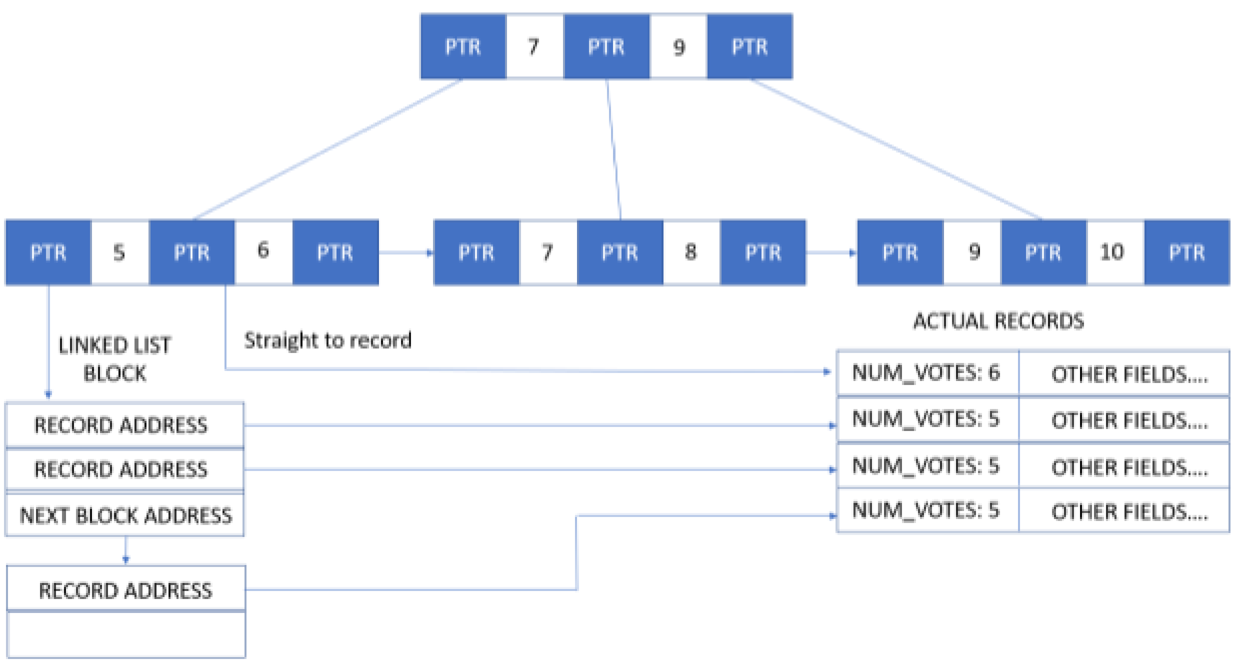

Figure 9 Duplicate Linkedlist

Next, for our tree nodes, they are the same size as blocks. We keep 2 arrays for keys and pointers. For keys, we decided to fill the arrays with 0s, as the minimum for numVotes is 5 and numVotes would never be 0. Hence, 0 represents that no key is present.

	averageRating	numVotes
count	1.070318e+06	1.070318e+06
mean	6.884317e+00	9.588359e+02
std	1.396673e+00	1.592951e+04
min	1.000000e+00	5.000000e+00
25%	6.100000e+00	9.000000e+00
50%	7.100000e+00	2.000000e+01
75%	7.800000e+00	7.800000e+01
max	1.000000e+01	2.279223e+06

Figure 10 Details of AverageRating and numVotes

```
// 200B block
class treeNodeBlock: public block{
public:
    std::array<unsigned int, NUM_KEY_INDEX> key = { 0 }; // Minimum is 5, so we make 0 a special value.
    std::array<Pointer, NUM_KEY_INDEX + 1> ptrs;

    // Returns number of keys
    unsigned int getLength(){
        unsigned int i=0, count=0;
        while(i<key.size()){
            if(key[i] == 0){
                count++;
            }
            i++;
        }
        return key.size()-count;
    }

    unsigned long getParentBlock(){
        return (parentBlock[2] << 16) + (parentBlock[1] << 8) + parentBlock[0];
    }

    void setParentBlock(unsigned long blockNum){
        parentBlock[2] = (blockNum & 0xFF0000) >> 16;
        parentBlock[1] = (blockNum & 0xFF00) >> 8;
        parentBlock[0] = blockNum & 0xFF;
    }
private:
    unsigned char parentBlock[3];
};
```

Figure 11 Tree Node Block

Next, we also created a LinkedListBlock for duplicates. Since the search key of the duplicates are all the same, to save space we just store the record pointer in the LinkedListBlocks without the search key. Each of these Record pointers would be a dense one to the exact record in the recordBlock.

```
class linkedListNodeBlock: public block{
public:
    char padding[BLOCK_SIZE - 4 - 1 - NUM_LINKED_LIST * 4];
    std::array<Pointer, NUM_LINKED_LIST> pointers;
    Pointer nextBlock;
};
```

Figure 12 LinkedList Block

Lastly, we made our own pointer that consists of both Block id and the Record Address. To save space, we made it into a 3 byte pointer. Using the entry (acts like an index), we would be able to retrieve the exact record in the record block. If entry is negative, it would mean it is not pointing to a record block. Hence the entry is only required when accessing a record from a recordBlock.

```
public:
    char entry = -1; // Points to exact record, for dense index leaf nodes
    // We can use a negative value to indicate it does not point to data. Hence we know that it points to a tree

    unsigned long getBlock() const {
        return (block[2] << 16) + (block[1] << 8) + block[0];
    }

    void setBlock(unsigned long blockNum){
        block[2] = (blockNum & 0xFF0000) >> 16;
        block[1] = (blockNum & 0xFF00) >> 8;
        block[0] = blockNum & 0xFF;
    }

private:
    std::array<unsigned char, 3> block = { }; // Points to block. 0 should be a special value.
    // Block is accessed as MSB [2],[1],[0] LSB
    // This is private, we will use getters and setters here.
};
```

Figure 13 Pointer Class

Function

We have 4 vital functions to help us perform the experiments, namely Insert(), Delete(), searchKeys(), searchRangeOfKeys().

1. Insert()

Insert is the function that we used to insert specific data into our B+ Tree and this will be used in our project to populate the database. When we insert, we will check if its a duplicate key. For non-duplicate keys, it would be inserted into the tree nodes. For duplicate keys, they will be inserted into a LinkedListBlock. (Due to the lengthy code, we did not include the screenshot here, as it can be found in the BPlusTree.hpp file)

During insertion, we have to consider a few cases. We split it into two functions: one to handle the insertion in the leaf node level and another for insertion of the internal nodes.

For insertion into leaf nodes, we again had to consider a few cases:

1) **When there is no root node in the tree:** This is at the very start. We keep track of the root address by storing it in the bplustree class. If there is no root node in the tree, we would create a root node, insert the key and ptr into the root node and store the address as the new root node.

2) **For normal insertion in the leaf node if there is space in the leaf node (not a duplicate key).** When there is space in the leaf node we can insert the key and ptr respectively into the leaf node. We would first search where to insert in the leaf node using the **searchBlockToContain function** in our B+ tree. This would return the location to insert the key. After checking that there is no duplicate key, we can just the key and pointer inserted into the leaf node.

3) **For normal insertion in the leaf node if there is space in the leaf node (duplicate key).** When there is an existing key in the leaf node, we would then need to insert it into the LinkedListBlock. Firstly, we check if the LinkedListBlock exists. If it doesn't, we would create it and insert both the old record address and the new duplicate key record address into the LinkedList Block. If a LinkedList Block exists, we would check if the LinkedList Block is full. If the LinkedList Block is not full, we would just add the record address into the Block. Else if it is full, we would recursively move to the next block until we either find a LinkedListBlock with space, or find that all current LinkedListBlocks are full, then we would then create a new LinkedList Block and add the record address in.

4) **For normal insertion in the leaf node if there is no space in the leaf node (duplicate key):** In the case that there is no space in the leaf node, we would still have to check if the key being inserted already exists. In this case, if it already exists, we can just follow the processes above of inserting a duplicate in the respective LinkedListBlock.

5) **For normal insertion in the leaf node if there is no space in the leaf node (not a duplicate key):** If the key inserted is a new unique key and there is no space for insertion we would have to split the keys and respective pointers. Firstly we gather all existing keys and pointers by pairing the keys to

their respective pointers and storing them into an array of std::Pair of keys and pointers, ignoring the last pointer which points to the next leaf node. We then sort this array into ascending order of key value. We split the keys with the following calculations: **Current leaf node** would keep $ceil(\frac{maxKey+1}{2})$ and **new leaf node** would keep $floor(\frac{maxKey+1}{2})$ **.** Since this is the leaf node, the pointers follows the same index as the keys and would be split in the same manner. Finally, we would update the last pointer of the **new node** to point to whichever node the **current leaf node** was previously pointing and **the last pointer** of the **current leaf node** to point to the **new leaf node.** Next we also check if the **current leaf node** is the root. **In that case a new root will be created** with the first pointer **pointing to the current leaf node** and second pointer **pointing to the new leaf node, and the first key** being the **lower bound of the right leaf node.** We then **update the parent index of both nodes** to the **new root node address.** However, if the **the parent of the current leaf node** exists, we then call a recursive function **insertInternal** that handles the insertion of the address of the **newly created leaf node** into the **parent of the current leaf node.**

6) **For inserting of internal nodes (Space available):** When inserting into an internal node, we first have to check if there is space available. If there is, we will collect the pointers of all children nodes and sort them based on the lowest bound of each leaf node. We then write back the pointers into the block, along with the corresponding new lowest bound of the respective blocks into the keys. We also created a **lowestBound** function that finds the **lowest bound of a leaf node** given the **address of an internal node.** The lowest bound function is used to trace and get the lower bound of the **respective leaf node** to be placed as the key **in the internal node.**

7) **For inserting of internal nodes (No space available):** In the event that there is no space available, we would again have to go through the splitting process. This is slightly trickier and hence instead of splitting by keys we handle the splitting using pointers instead. We first copied **all the pointers of the current child node** into a temporary array, and inserted the new pointer **of the new child node** into the array. We then sorted them by their lower boundaries of keys in their respective descendent **leaf nodes.** We then proceed to split the pointers using the following calculation: **Current child node** would be $ceil(\frac{maxKey}{2}) + 1$ and **new child node** would be $floor(\frac{maxKey}{2}) + 1$. We then place the find the lower boundaries of each of the pointer (excluding the first pointer) to update the respective keys. After which, we would also set each of the parent address of the child node following the pointer to the respective **current node or new node addresses.** Similar to the process above, we would check if the **current internal node is a root.** If it is, a new root would need to be created and the left and right pointers of the new root would be the **current internal node and new internal node respectively.** The key would the lower bound of the **new internal node.** The parent addresses would also be updated accordingly to be the **new root** address. However, if a parent node exists for the **current internal node, the insertInternal function would be recursively called to insert the new internal node until a new root is created or there is space in the parent node to insert without splitting the node.**

2. Delete()

Deletion takes in a given key and deletes the corresponding record or linked list of records of the same value at its leaf node. After that, the function traverses the tree from the leaf node to the root node recursively to delete any redundant keys and balance the tree. The deletion is done using three main functions, deleteKey(), which handles deletion at the leaf node, deleteKeyInternal(), which handles the deletion of keys in a node, and a recursive updateParent(), which handles the update of non-leaf nodes after the deletion in the leaf node. A few utility functions were created to support the deletion process.

In the first section of the code as shown in figure. 14, the function deleteKey() checks if the key exists inside the tree. The function takes in the key and traverses through the whole tree depth-first until a leaf node is reached while keeping track of the current node's left and right siblings. After the leaf node is reached, the function attempts to find the given key within the leaf node and returns if the key is not found.

```cpp
// Delete key
cursor->key[index] = 0;
cursor->ptrs[index].entry = -1;

// LB changed hence must update parent
if (index == 0)
{
    // update
    updateParent(cursor->getParentBlock());
}

// Move subsequent keys and pointers 1 position forward
for (int i = index; i < cursor->getLength(); i++)
{
    // currNode->key[i] = currNode->key[i + 1];
    cursor->ptrs[i] = cursor->ptrs[i + 1];
    cursor->key[i] = cursor->key[i + 1];
}

// Check if current leaf less than minimum key
unsigned int minimum = (NUM_KEY_INDEX + 1) / 2;

// Tree is now empty
if (parentIndex == 0 && cursor->getLength() < minimum)
{
    blkManager->deleteBlock(rootNode);
    rootNode = 0;
    std::cout << "No keys left in the tree. Killing tree..." << std::endl;
    return;
}

if (cursor->getLength() < minimum)
{
    mergeLeafNodes(leftIndex, curIndex);
}
```

Figure 14 : deleteKey() finds the node that the key is in and returns if key is not found.

Otherwise, the function calls the block manager to delete the record. In the event that there are multiple records of the same value stored in a linked list, the function iterates through the linked list and deletes all records within the linked list. After the deletion, the function checks the number of keys and if it falls below the minimum, mergeLeafNode() is called. If the node itself is the root and the only key is deleted, then the tree is empty and the tree is deleted. The code for this section is displayed in Figure 14.

MergeLeafNode() is a utility function used to balance out the leaf nodes within the B+ tree. It takes in a leaf node and attempts to borrow a key from a sibling node. It checks whether the sibling exists and if the sibling has enough keys to spare, it transfers a key from the sibling to the current leaf node. In case no siblings can lend a key, the current node merges with the sibling instead, and deleteKeyInternal() is called after merge to update the respective internal nodes. DeleteKeyInternal() would then call MergeParentNodes() which handles the borrowing of a key from a sibling and merging of **internal nodes** through a similar algorithm as the one coded in mergeLeafNode(). The keys within each non-leaf node are then updated recursively via updateParent(). The borrowing and merging for mergeLeafNode() is illustrated in figure 15 and 16. Due to the length we only showed an example for the left leaf node and current leaf node here. Left index here returns the block id of the sibling of the respective node. If there is no left sibling, it would be a 0.

```
// Left sibling can make a transfer
if (leftIndex != 0 && leftNode->getLength() > minimum) {
    //1 2 3
    //A B C next leaf node
    // Make space for the transfer
    for (int i = currNode->getLength()-1; i > 0; i--) {
        currNode->key[i] = currNode->key[i - 1];
        currNode->ptrs[i] = currNode->ptrs[i-1];
    }



    // Transfer key from leftNode to currNode
    currNode->key[0] = leftNode->key[leftNode->getLength() - 1];
    currNode->ptrs[0] = leftNode->ptrs[leftNode->getLength() - 1];


    // Update left leaf
    leftNode->key[leftNode->getLength() - 1] = 0;
    leftNode->ptrs[leftNode->getLength() - 1].setBlock(0);
    leftNode->ptrs[leftNode->getLength() - 1].entry = -1;

    //update parent
    updateParent(currNode->getParentBlock());
    return;
}
```

Figure 15 Curr node borrowing

```
// Merge leftNode and currNode
else if (leftIndex != 0 && leftNode->getLength() <= minimum)
{
    // Transfer all keys and pointers from currNode to leftNode
    for (int i = leftNode->getLength()+1, j = 0; j < currNode->getLength(); i++, j++)
    {
        leftNode->key[i] = currNode->key[j];
        leftNode->ptrs[i] = currNode->ptrs[j];
    }
    leftNode->ptrs[leftNode->ptrs.size()-1] = currNode->ptrs[leftNode->ptrs.size()-1];


    // Delete currNode from parent
    unsigned int parentBlockIndex = currNode->getParentBlock();
    blkManager->deleteBlock(currNodeIndex);

    //recursion for parent
    deleteKeyInternal(currNodeIndex, parentBlockIndex);

}
```

Figure 16 left leaf node merging with current leaf node

UpdateParent() is a recursive function that updates the non-leaf nodes of the B+ tree whenever the first key is deleted/changed in the leaf node as this affects the lowest bound in the internal nodes. It recalculates all the keys within the node using the utility function lowestBound() and recursively updates the non-leaf nodes from the second-lowest level to the children of the root node. figure. 16b shows the code for this function.

```
void updateParent(unsigned int currNodeIndex){
    treeNodeBlock *currNode = (treeNodeBlock*) blkManager->accessBlock(currNodeIndex);
    int i = 1;
    while(currNode->ptrs[i].getBlock() != 0 ){
        treeNodeBlock* temp = (treeNodeBlock*)blkManager->accessBlock(currNode->ptrs[i].getBlock());
        currNode->key[i-1] = lowestBound(temp);
        i++;
    }
    if(currNodeIndex != rootNode){
        updateParent(currNode->getParentBlock());
    }
}
```

Figure 16b Update of keys in internal nodes

As our tree node structure stores only the pointer to the right sibling, we require a function to help find the left sibling of the current node, especially in the case where the current node is the leftmost node of its parent. The helper function, findLeftIndex() and findRightIndex() is written with this in mind and takes in the current node to output the block id of its respective left sibling and right sibling.

In the case of merging, the deletion would only end when the tree is balanced.

3. searchKeys()

Our search function traverses the B+ tree by comparing the key that we are searching for with the current node's key starting from the root, all the way to the leaf nodes. From figure 17, we initialise a vector of Records called results, which would be all the result keys that we would be returning. After getting the block that the root node is in, we traverse down the tree by iterating through the internal nodes. During each iteration, we find the correct pointer to follow by checking if the key we are searching for is more than or equal to the current key in each block. Once the key being searched is lesser than the current key, we would access the block in the next internal node layer and go to that layer.

```
//Search function
std::vector<Record> searchKeys(unsigned int key){
    std::vector<Record> results;
    unsigned int curIndex = rootNode;
    treeNodeBlock* curBlock = (treeNodeBlock*) blkManager->accessBlock(rootNode);

    // iterating through internal nodes to get to leaf nodes
    while (curBlock -> type != 2) {  // while it is not a leaf node
        unsigned int i = 0;
        while (i<curBlock->getLength() && key>curBlock->key[i]) {
            i++;
        }
        curIndex = curBlock->ptrs[i].getBlock();
        curBlock = (treeNodeBlock*) blkManager->accessBlock(curIndex);
    }
```

Figure 17 Iterating through internal nodes

Once we have reached a leaf node, we would need to search through the leaf node to find the exact location in that node that the key is at figure 18. This index would be stored in j. To know whether the key does exist in the tree, we compare j with the length of the block because if they are equal to each other, it means that we have iterated through the whole leaf node that the key is supposed to be found in already. Thus, if it is not found in that node, it does not exist in the tree.

```
//Now curIndex and curBlock should be leaf node
unsigned int j = 0;

// search through the leaf node for the key
while (j < curBlock->getLength() && key != curBlock->key[j]) {
    j++;
}

// means that we have iterated through the whole leaf node already == key does not exist
if (j == curBlock->getLength()) {
    std::cout<<"Key does not exist"<<std::endl;
    return results;
}
```

Figure 18 Iterating through the leaf node

Now, we can access the block that the Record containing the key we are searching for. There are 2 cases for this: if there are no duplicates, this block would be a Record Block and we can directly access the record from there based on the index that we have obtained previously Figure 19. The second case would be for duplicates, where we would have to iterate through all the Linked List blocks and access the Record Blocks for each of the Record addresses in the Linked List blocks Figure 19.

```
// accessing Record Blocks
unsigned int curBlockIndex = curBlock->ptrs[j].getBlock();

// directly accessing record blocks, no duplicates
if (blkManager->accessBlock(curBlockIndex) ->type == 0) {
    RecordBlock* curRecordBlock = (RecordBlock*)blkManager->accessBlock(curBlockIndex);
    unsigned int accessIndex = curBlock->ptrs[j].entry;
    results.push_back(curRecordBlock->records[accessIndex]);
}
```

Figure 19 Accessing the Record Block

If we identify that the block is a Linked List block, we iterate through the block and access each Record Block that the record addresses stored points toFigure 20, and store this in the results vector. Since the Linked List block may not be full, we included an IF condition whereby if the Record Block that we are accessing is a 0, it means that there are no more Record Blocks in the Linked List with the same key, thus we would break out of the searching loop.

```
// accessing linked list for duplicates
else if (blkManager -> accessBlock(curBlockIndex) ->type == 3){
    linkedListNodeBlock* linkedList = (linkedListNodeBlock*) blkManager -> accessBlock(curBlockIndex);

    // for first linkedlist block
    for(Pointer record_pointer: linkedList->pointers){
        if(record_pointer.getBlock() == 0){
            break;
        }
        // exact index of the record in the record block
        unsigned int accessIndex = record_pointer.entry;
        // index of the record block
        unsigned int blockIndex = record_pointer.getBlock();
        RecordBlock* curRecordBlock = (RecordBlock*)blkManager->accessBlock(blockIndex);
        results.push_back(curRecordBlock->records[accessIndex]);
    }
```

Figure 20 Accessing Linked List Block

In Figure 21, we have another WHILE loop to loop through the Linked List blocks for a particular key, for the case when there is more than one Linked List block. At the start of each loop, we will obtain the index of the next Linked List block based on the current block. We repeat the same steps as above in Figure 20. At the end, when we have retrieved all the Records with the key, we will return the Results vector.

```
            // if there are more than one linkedlist block
            while(linkedList->nextBlock.getBlock() != 0){
                //Switch to next linkedlist block
                unsigned int nextLinkedListIndex = linkedList->nextBlock.getBlock();
                linkedList = (linkedListNodeBlock*) blkManager -> accessBlock(nextLinkedListIndex);
                for(Pointer record_pointer: linkedList->pointers){
                    if(record_pointer.getBlock() == 0){
                        break;
                    }
                    unsigned int accessIndex = record_pointer.entry;
                    unsigned int blockIndex = record_pointer.getBlock();
                    RecordBlock* curRecordBlock = (RecordBlock*)blkManager->accessBlock(blockIndex);
                    results.push_back(curRecordBlock->records[accessIndex]);
                }
            }
        }
        return results;
}
```

Figure 21 Accessing Linked List Block

4. searchRangeOfKeys()

To find the keys that are present within a specific range, we have created a search range function that would return a vector of Records that fall within the range of the lower and upper bounds specified. Figure 22 shows the iteration through the internal nodes to reach the leaf nodes. This is similar to the beginning of the Search function, except that we are now comparing the current key to the Lower Bound to find the first record which is in the range, whereby we iterate through the internal node when the Lower Bound is more than the current key. This would allow us to find the first key that is larger than or equals to the Lower Bound. We then go to the next block where the smallest key in the range belongs to.

```
//Search range function
std::vector<Record> searchRangeOfKeys(int LowerBound, int UpperBound) {
    std::vector<Record> results;
    unsigned int curIndex = rootNode;
    treeNodeBlock* curBlock = (treeNodeBlock*) blkManager->accessBlock(rootNode);

    while (curBlock -> type != 2) {  // while it is not a leaf node
        unsigned int i = 0;

        // search until the point where the lower bound of the range is found, if not keep iterating
        while (i < curBlock->getLength() && curBlock->key[i] < LowerBound) {
            i++;
        }
        curIndex = curBlock->ptrs[i].getBlock();
        curBlock = (treeNodeBlock*) blkManager->accessBlock(curIndex);
    }
```

Figure 22 Iterating through internal nodes

To ensure that there are keys in the specified range, we checked if the first key in the current leaf node is larger than the Upper Bound. If it is larger, then no such keys exist and we return an empty vector, as shown in Figure 23.

```
// Now curIndex and curBlock should be leaf node
// iterate through leaf node
unsigned int j = 0;
if(curBlock->key[0]>UpperBound){
    std::cout<<"Keys in range do not exist"<<std::endl;
    return results;
}
```

Figure 23 No such key in the range

As shown in Figure 24, the external WHILE loop checks for the first key in every leaf node that we are accessing. If the first key is greater than the Upper Bound, we have exceeded the range and have completed the search.

Next, we iterate through each leaf node until the Upper Bound since we are incrementing it. Next, when we found another pointer within this range, we then check the type of the block that the current key points to. There are 2 scenarios here, one would be a Record Block in the case that there are no duplicates and the other would be a Linked List block if there are duplicates, which is similar to the Search function above.

```
// Now curIndex and curBlock should be leaf node
// iterate through leaf node
unsigned int j = 0;
// find the index of the first key that falls in the range
while (curBlock->key[0] <= UpperBound) {
    j = 0;
    while(curBlock->key[j]< LowerBound){
        j++;
    }

    while(LowerBound<curBlock->key[j]){
        LowerBound+= 1;
    }
    //20000 20001 30001//30003 30005 //30006 40001
    while (LowerBound <= curBlock->key[curBlock->getLength()-1] && LowerBound <= UpperBound) {
        while(j < curBlock->getLength()){
            if(LowerBound == curBlock->key[j]){
                unsigned int curBlockIndex = curBlock->ptrs[j].getBlock();
                // directly accessing record blocks, no duplicates
                if (blkManager->accessBlock(curBlockIndex) ->type == 0) {
                    RecordBlock* curRecordBlock = (RecordBlock*)blkManager->accessBlock(curBlockIndex);
                    unsigned int accessIndex = curBlock->ptrs[j].entry;
                    results.push_back(curRecordBlock->records[accessIndex]);
                }
```

Figure 24 Iterating through leaf nodes based on the Lower and Upper Bound

From Figure 25, we follow the same process as the Search function when dealing with Linked List blocks as well, whereby we iterate through the Linked List block and go to each Record Address to access the Record with key that falls in the range. If there is more Linked List block for a particular key, we repeat the same process for the rest of the blocks.

```
// accessing linked list for duplicates
else if (blkManager -> accessBlock(curBlockIndex) ->type == 3){
    linkedListNodeBlock* linkedList = (linkedListNodeBlock*) blkManager -> accessBlock(curBlockIndex);

    // for first linkedlist block
    for(Pointer record_pointer: linkedList->pointers){
        if(record_pointer.getBlock() == 0){
            break;
        }
        // exact index of the record in the record block
        unsigned int accessIndex = record_pointer.entry;
        // index of the record block
        unsigned int blockIndex = record_pointer.getBlock();
        RecordBlock* curRecordBlock = (RecordBlock*)blkManager->accessBlock(blockIndex);
        results.push_back(curRecordBlock->records[accessIndex]);
    }

    // if there are more than one linkedlist block
    while(linkedList->nextBlock.getBlock() != 0){
        //Switch to next linkedlist block
        unsigned int nextLinkedListIndex = linkedList->nextBlock.getBlock();
        linkedList = (linkedListNodeBlock*) blkManager -> accessBlock(nextLinkedListIndex);
        for(Pointer record_pointer: linkedList->pointers){
            if(record_pointer.getBlock() == 0){
                break;
            }
            unsigned int accessIndex = record_pointer.entry;
            unsigned int blockIndex = record_pointer.getBlock();
            RecordBlock* curRecordBlock = (RecordBlock*)blkManager->accessBlock(blockIndex);
            results.push_back(curRecordBlock->records[accessIndex]);
        }
    }
}
```

Figure 25 Accessing the actual Record

From Figure 26, after iterating through the current leaf node, we need to continue checking the subsequent leaf nodes for keys that lie within the range. Thus, we access the next leaf node and repeat the same process as above. At the end, we will obtain a vector of Records storing the results of the search and we will return this vector.

```
                }
            }
            j+=1;
        }
        LowerBound+=1;
        break;
    }
};
j = 0;
curIndex = curBlock->ptrs[(curBlock->ptrs.size())-1].getBlock();
curBlock = (treeNodeBlock*) blkManager->accessBlock(curIndex);

}
return results;
```

Figure 26 Accessing the next leaf node

Experiments:

Experiment 1:

The following statistics as reported from our project:

- The number of blocks
- The size of database (in terms of MB)

Block Size	Number of blocks	Size of Database
200 B	89194	17838800 B = 17.01MB
500 B	34527	17263500 B = 16.46 MB

Validation:

Total number of records: 1070318

200B can store 12 records, therefore 89194 blocks

500B can store 31 records, therefore 34527 blocks

200 B

```
------Exercise 1----------
Number of blocks used by storage is: 89194
Size used by storage is: 17838800B
```

500 B

```
------Exercise 1----------
Number of blocks used by storage is: 34527
Size used by storage is: 17263500B
```

Experiment 2:

We have inserted the attribute "NumVotes" sequentially in our B+ tree and the following statistics are the reported results for these four categories:

1. The parameter n of the B+ tree (number of max keys)
2. The number of nodes of the B+ tree
3. The height of the B+ tree (which excludes duplicated nodes)
4. The content of root node and its first child node

Block Size	N	Number Of Nodes	Height Of B+ Tree
200 B	24	29066	4
500 B	61	15487	3

The parameter N is calculated:

Verification:

```
In [3]:  # Check distinct values in numVotes
         len(data["numVotes"].unique())

Out[3]:  18072
```

Total unique values to be inserted in B+ tree: 18072
Lower bound of height for 200B (Assuming each node fully filled): 4
Upper bound of height for 200B (Assuming only minimum keys of 12): 4
Hence the height has to be 4.

Lower bound of height for 500B (Assuming each node fully filled): 3
Upper bound of height for 500B (Assuming only minimum keys of 12): 3
Hence the height has to be 3.

The content of root node and first child node

Block Size	Content of root node and first child node
200 B	Content of root node: -- Block id is: 93976 Parent id is: 0 Type is: 1 Keys: 5254 18365 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 89387 93975 102502 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Content of 1st child node: -- Block id is: 89387 Parent id is: 93976 Type is: 1 Keys: 300 600 951 1347 1699 1926 2151 2510 2922 3250 3579 3836 4081 4340 4599 4922 0 0 0 0 0 0 0 0 Ptrs: 89198 89386 89944 89523 90264 90980 95858 89807 91866 90719 93421 91566 106860 92810 104983 90281 97671 0 0 0 0 0 0 0 0
500 B	Content of root node: -- Block id is: 35962 Parent id is: 0 Type is: 1 Keys: 1618 3024 4596 7000 11555 18122 31173 78494 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 34541 35961 43003 36661 40242 37618 44734 38444 40766 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Content of 1st child node: -- Block id is: 34541 Parent id is: 35962 Type is: 1 Keys: 59 91 135 188 225 267 307 340 395 451 501 557 602 652 685 728 782 837 879 917 977 1008 1040 1089 1148 1194 1240 1293 1350 1386 1421 1457 1496 1546 1584 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 34528 34554 34652 34575 34664 34822 34540 34976 34707 34902 34599 35004 34791 35128 34674 35294 34854 35235 34564 35595 34875 35254 36201 34692 35466 34891 35707 35150 35575 34658 36646 35545 36750 35112 35761 37150 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Experiment 3:

During this experiment we were tasked to retrieve movies with “NumVotes” equal to 500 and the following statistics were reported

1. The number and the content of Index Nodes the process access
2. The number and the content of Data Blocks the process accesses
3. The average of “averageRating’s” of the records that are returned

Block Size	No. Index Nodes Access (Unique)	No. of Data Blocks Access (Unique)	Avg “averageRating” returned	Total SearchIO (Number of blocks accessed)
200 B	7	110	6.73812	117
500 B	4	110	6.73182	114

Verification:

Average of average rating is as shown below which matches.

B Plus tree search

In [88]:
```
numVotes500 = data[data["numVotes"] == 500]
```

In [89]:
```
numVotes500
```

Out[89]:

	tconst	averageRating	numVotes
3595	tt0013674	7.0	500
9018	tt0024561	6.8	500
11826	tt0028277	7.7	500
22779	tt0041956	6.5	500
27186	tt0047361	7.3	500
...	...	...	...
1043265	tt8960572	6.0	500
1059269	tt9454484	8.7	500
1062934	tt9614612	7.7	500
1064691	tt9680914	7.1	500
1069708	tt9895050	9.0	500

110 rows × 3 columns

In [90]:
```
numVotes500["averageRating"].mean()
```

Out[90]: 6.731818181818184

<u>Content of Index blocks accessed</u>

200 B

```
------Exercise 3----------

Total Number of Index Nodes accessed (Unique): 7

First 5 Unique Index Nodes accessed:
93976||89387||89386||90315||89951||
------------------------------------------
Block id is: 93976
Parent id is: 0
Type is: 1
Keys:
5254 18365 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Ptrs:
89387 93975 102502 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

------------------------------------------
Block id is: 89387
Parent id is: 93976
Type is: 1
Keys:
300 600 951 1347 1699 1926 2151 2510 2922 3250 3579 3836 4081 4340 4599 4922 0 0 0 0 0 0 0 0
Ptrs:
89198 89386 89944 89523 90264 90980 95858 89807 91866 90719 93421 91566 106860 92810 104983 90281 97671 0 0 0 0 0 0 0 0

------------------------------------------
Block id is: 89386
Parent id is: 89387
Type is: 1
Keys:
316 332 354 367 382 403 418 432 451 465 478 491 503 523 544 561 575 588 0 0 0 0 0 0
Ptrs:
89320 89681 89460 89243 90030 89606 89370 90047 89637 89219 90111 89628 90315 89405 89765 89586 90133 89302 90970 0 0 0 0 0 0

------------------------------------------
Block id is: 90315
Parent id is: 89386
Type is: 2
Keys:
491 492 493 494 495 496 497 498 499 500 501 502 0 0 0 0 0 0 0 0 0 0 0 0
Ptrs:
89548 90228 90087 90283 90496 90104 90014 90079 89604 89951 89708 90207 0 0 0 0 0 0 0 0 0 0 0 0 89405

------------------------------------------
Block id is: 89951
Type is: 3
Record addresses:
300 752 986 1899 2266 2271 2556 2650 2758 3334 3818 3941 4759 4799 4966 5330 5967 6075 6473 7341 7415 8615 9193 10071 11251 11402 14113 14697 15142 15592 15605 18066 18943 19843 21980 22060 23536 24461 24656 24658
24662 24664 26068 26459 26460 26747 29194 29300
Next linkedList Block Pointer: 101737
```

500 B

```
Total Number of Index Nodes accessed (Unique): 4

First 5 Unique Index Nodes accessed:
35962||34541||34599||35167||
------------------------------------------
Block id is: 35962
Parent id is: 0
Type is: 1
Keys:
1618 3024 4596 7000 11555 18122 31173 78494 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Ptrs:
34541 35961 43003 36661 40242 37618 44734 38444 40766 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

------------------------------------------
Block id is: 34541
Parent id is: 35962
Type is: 1
Keys:
59 91 135 188 225 267 307 340 395 451 501 557 602 652 685 728 782 837 879 917 977 1008 1040 1089 1148 1194 1240 1293 1350 1386 1421 1457 1496 1546 1584 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Ptrs:
34528 34554 34652 34575 34664 34822 34540 34976 34707 34902 34599 35004 34791 35128 34674 35294 34854 35235 34564 35595 34875 35254 36201 34692 35466 34891 35707 35150 35575 34658 36646 35545 36750 35112
35761 37150 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

------------------------------------------
Block id is: 34599
Parent id is: 34541
Type is: 2
Keys:
451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 0 0 0 0 0 0 0 0 0 0 0
Ptrs:
34841 34848 35282 35206 35157 35710 35031 35529 35556 35063 35589 35191 35623 35326 34973 34811 35427 34710 35098 35148 35346 35171 35069 35009 34959 35598 35464 34884 34702 35320 34890 35424 35192 34729 35943 35258
34962 35419 35134 35742 34819 35398 35281 35437 35604 35291 35224 35276 34870 35167 0 0 0 0 0 0 0 0 0 0 0 35004

------------------------------------------
Block id is: 35167
Type is: 3
Record addresses:
116 291 382 735 877 880 990 1026 1068 1291 1478 1526 1843 1858 1923 2064 2310 2352 2506 2842 2871 3335 3559 3899 4356 4414 5464 5690 5862 6036 6041 6994 7333 7682 8509 8540 9111 9469 9544 9545 9547 9548
10091 10242 10243 10354 11301 11342 12293 16197 17183 17801 17964 18001 18028 18289 18832 18958 19155 19289 19511 19555 19605 19798 19937 20042 20738 21467 22337 22382 22853 23603 23795 23889 24476 24625
24688 24759 24784 25034 25110 25446 25634 26089 26267 26659 26898 27081 27314 27402 27466 27760 29792 29823 30130 30151 30297 31287 31321 31900 31932 31999 32238 32453 32677 33654 34170 34289 34345 34507 0 0 0 0 0 0 0 0 0 0 0 0 0
Next linkedList Block Pointer: 0
```

Content of Data blocks accessed

200 B	Total Number of Data Blocks accessed (Unique): 110 First 5 Unique Data Blocks accessed: 300||752||986||1899||2266|| --- For full content of data blocks accessed refer to our Logs
500 B	Total Number of Data Blocks accessed (Unique): 110 First 5 Unique Data Blocks accessed: 116||291||382||735||877|| --- For full content of data blocks accessed refer to our Logs

*As per requirements we are only reporting the first 5 unique index nodes and datablocks.

Verification:
For **200B: (Divide by 12 since 12 records in each block) For first 5 datablock accessed)**
Block id: **3595/12 =** 300
Block id: **9018/12 =** 752
Block id: **11826/12 =** 986
Block id: **22779/12 =** 1899
Block id: **27186/12 =** 2266

For **500B: (Divide by 31 since 31 records in each block) For first 5 datablock accessed)**
Block id: **3595/31 =** 116
Block id: **9018/31 =** 291
Block id: **11826/31 =** 382
Block id: **22779/31 =** 735
Block id: **27186/31 =** 877

In [89]:
```
numVotes500
```

Out[89]:

	tconst	averageRating	numVotes
3595	tt0013674	7.0	500
9018	tt0024561	6.8	500
11826	tt0028277	7.7	500
22779	tt0041956	6.5	500
27186	tt0047361	7.3	500
...	...	...	...
1043265	tt8960572	6.0	500
1059269	tt9454484	8.7	500
1062934	tt9614612	7.7	500
1064691	tt9680914	7.1	500
1069708	tt9895050	9.0	500

110 rows × 3 columns

Experiment 4:

Similar to experiment 3 we are tasked to retrieve movies with the attribute "numVotes" from 30,000 to 40,000 and the statistics that will be reported are:

1. The number and the content of index nodes the process accesses
2. The number an the content of data blocks the process accesses
3. The average of "averageRating's" of the records that are returned

Block Size	No. Index Nodes Access	No. of Data Blocks Access	Average of average ratings	Total SearchIO
200 B	110	932	6.72791	1042
500 B	76	911	6.72791	987

Verification:

Average of average rating is as shown below which matches.

Bplus Tree Range Search Checking

Average of average ratings of all records: 6.1087 (100,000)records loaded

```
In [59]: new = data.loc[(data['numVotes'] >= 30000) & (data['numVotes'] <= 40000)]

In [60]: new["averageRating"].mean()
Out[60]: 6.727911857292764

In [61]: new.sort_values(by="numVotes")
Out[61]:
```

	tconst	averageRating	numVotes
32968	tt0054167	7.7	30022
10652	tt0026778	7.9	30034
65199	tt0091828	5.6	30037
773789	tt3361792	6.8	30041
572980	tt1456941	6.2	30049
...	...	...	...
616056	tt1698641	6.2	39951
83906	tt0114287	6.9	39963
226387	tt0421729	4.7	39979
567389	tt1423995	5.4	39988
958452	tt6742252	7.5	39996

953 rows × 3 columns

Content of Index blocks accessed

200 B

```
------Exercise 4----------

Total Number of Index Nodes accessed (Unique): 110

First 5 Unique Index Nodes accessed:
93976||102502||103298||116561||90578||
```

```
------------------------------------------
Block id is: 93976
Parent id is: 0
Type is: 1
Keys:
5254 18365 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Ptrs:
89387 93975 102502 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

------------------------------------------
Block id is: 102502
Parent id is: 93976
Type is: 1
Keys:
19839 21658 23439 25863 28769 32062 36904 39772 42764 49775 56815 62974 70213 77882 86611 99298 112772 129366 155251 195826 248283 392268 0 0
Ptrs:
91189 105039 94208 108923 92962 103298 93974 98009 117172 92119 101714 94120 111467 97632 113662 92949 109325 97429 110993 93905 105825 96539 102747 0 0

------------------------------------------
Block id is: 103298
Parent id is: 102502
Type is: 1
Keys:
29024 29136 29268 29353 29603 29818 29949 30034 30246 30402 30576 30672 30766 30888 31073 31254 31353 31524 31636 31727 31894 0 0 0
Ptrs:
91935 97981 117122 93057 107669 96174 106251 116561 90578 106304 95357 93080 112384 106133 93836 102410 92506 98497 93968 113114 96318 110174 0 0 0

------------------------------------------
Block id is: 116561
Parent id is: 103298
Type is: 2
Keys:
29949 29954 29956 29959 29962 29974 29975 29978 29982 29988 29996 30022 0 0 0 0 0 0 0 0 0 0 0 0
Ptrs:
68569 58617 116838 59446 19544 7917 111727 13964 71855 67434 58617 2748 0 0 0 0 0 0 0 0 0 0 0 0 90578

------------------------------------------
Block id is: 90578
Parent id is: 103298
Type is: 2
Keys:
30034 30037 30041 30049 30053 30056 30078 30081 30085 30090 30136 30144 30149 30158 30168 30175 30177 30195 30206 30221 30240 0 0 0
Ptrs:
888 5434 64483 47749 6736 6407 98820 58373 55351 4506 33152 47446 61202 96152 87233 86420 4778 51988 63288 4746 6925 0 0 0 106304
```

500 B

```
------Exercise 4----------

Total Number of Index Nodes accessed (Unique): 76

First 5 Unique Index Nodes accessed:
35962||44734||40367||40901||39440||
------------------------------------------
```

	`` -- Block id is: 35962 Parent id is: 0 Type is: 1 Keys: 1618 3024 4596 7000 11555 18122 31173 78494 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 34541 35961 43003 36661 40242 37618 44734 38444 40766 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -- Block id is: 44734 Parent id is: 35962 Type is: 1 Keys: 18365 18503 18624 18888 19127 19330 19557 19793 19994 20184 20336 20511 20762 20945 21284 21570 21913 22214 22531 22799 23024 23201 23409 23603 23832 24285 24580 24769 25096 25465 25672 25900 26238 26566 26906 27153 27413 27668 28007 28313 28759 29189 29585 29880 30240 30552 30816 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 40545 37185 48812 42427 38039 43601 36353 45394 38846 44733 37828 48024 39499 45801 37051 40572 38011 40814 35297 40396 46359 38038 48591 40617 46759 37087 39903 47102 37801 46726 39408 47820 36400 45472 39727 47483 37932 47985 39895 48348 37091 43034 38001 46773 40367 46738 35960 46170 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -- Block id is: 40367 Parent id is: 44734 Type is: 2 Keys: 29880 29882 29900 29910 29919 29949 29954 29956 29959 29962 29974 29975 29978 29982 29988 29996 30022 30034 30037 30041 30049 30053 30056 30078 30081 30085 30090 30136 30144 30149 30158 30168 30175 30177 30195 30206 30221 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 6841 21758 33097 704 2730 26543 22691 49392 23012 7566 3065 46953 5406 27815 26104 22691 1064 344 2104 24961 18484 2608 2480 40901 22596 21426 1744 12833 18366 23692 39440 33768 33453 1850 20125 24499 1837 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 46738 -- Block id is: 40901 Type is: 3 Record addresses: 4963 8172 0 Next linkedList Block Pointer: 0 -- Block id is: 39440 Type is: 3 Record addresses: 2548 5126 0 Next linkedList Block Pointer: 0 ```

<u>Content of Data blocks accessed</u>

200 B	`Total Number of Data Blocks accessed (Unique): 932` `First 5 Unique Data Blocks accessed:` `2748||888||5434||64483||47749||` `------------------------------------------` For full content of data blocks accessed refer to our Logs
500 B	`Total Number of Data Blocks accessed (Unique): 911` `First 5 Unique Data Blocks accessed:` `1064||344||2104||24961||18484||` `------------------------------------------` For full content of data blocks accessed refer to our Logs

*As per requirements we are only reporting the first 5 index nodes and data blocks.

Verification:

For **200B (Divide by 12 since 12 records in each block) For first 5 datablock accessed):**

Block id: **32968 / 12 = 2748**

Block id: **10652/ 12 = 888**

Block id: **65199/ 12 = 5434**

Block id: **773789/ 12 = 64483**

Block id: **572980/ 12 = 47749**

For **500B (Divide by 31 since 31 records in each block) For first 5 datablock accessed):**

Block id: **32968 / 31 = 1064**

Block id: **10652/ 31 = 344**

Block id: **65199/ 31 = 2104**

Block id: **773789/ 31 = 24961**

Block id: **572980/ 31 = 18484**

In [61]:
```
1 new.sort_values(by="numVotes")
```

Out[61]:

	tconst	averageRating	numVotes
32968	tt0054167	7.7	30022
10652	tt0026778	7.9	30034
65199	tt0091828	5.6	30037
773789	tt3361792	6.8	30041
572980	tt1456941	6.2	30049
...	...	...	...
616056	tt1698641	6.2	39951
83906	tt0114287	6.9	39963
226387	tt0421729	4.7	39979
567389	tt1423995	5.4	39988
958452	tt6742252	7.5	39996

953 rows × 3 columns

Experiment 5:

In the last experiment we are tasked to delete the movies with the attributes "numVotes" equal to 1000 and report on the statistics mentioned below on the updated B+ tree

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 First child nodes of the updated B+ tree

Block Size	No. of node is Deleted	No. of nodes of the updated B+ Tree	Height of B+ Tree
200 B	1	29065	4
500 B	1	15486	3

We verified that since the deletion of the key from the node left the node with greater or equal to the number of minimum keys, there was no borrowing/merging to be done. Furthermore, the deletion of the key was not the start of the node, hence no updating of keys in the parent nodes were needed. Since there was no merging, **the height of the B+ Tree** remained the same.

We also checked that only one node needs to be deleted as for num Votes = 1000 there are only 42 records, which fits in one linkedListBlock. Hence only one LinkedList block was deleted and the total no. of nodes of the updated B+ Tree decreased by 1.

Bplus delete

```
len(data[data["numVotes"] == 1000])
```

```
42
```

<u>The content of root node and first child node</u>

Block Size	Content of root node and first node
200 B	Content of root node: -- Block id is: 93976 Parent id is: 0 Type is: 1 Keys: 5254 18365 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 89387 93975 102502 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Content of 1st child node: -- Block id is: 89387 Parent id is: 93976 Type is: 1 Keys: 300 600 951 1347 1699 1926 2151 2510 2922 3250 3579 3836 4081 4340 4599 4922 0 0 0 0 0 0 0 0 Ptrs: 89198 89386 89944 89523 90264 90980 95858 89807 91866 90719 93421 91566 106860 92810 104983 90281 97671 0 0 0 0 0 0 0 0
500 B	Content of root node: -- Block id is: 35962 Parent id is: 0 Type is: 1 Keys: 1618 3024 4596 7000 11555 18122 31173 78494 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 34541 35961 43003 36661 40242 37618 44734 38444 40766 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Content of 1st child node: -- Block id is: 34541 Parent id is: 35962 Type is: 1 Keys: 59 91 135 188 225 267 307 340 395 451 501 557 602 652 685 728 782 837 879 917 977 1008 1040 1089 1148 1194 1240 1293 1350 1386 1421 1457 1496 1546 1584 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 Ptrs: 34528 34554 34652 34575 34664 34822 34540 34976 34707 34902 34599 35004 34791 35128 34674 35294 34854 35235 34564 35595 34875 35254 36201 34692 35466 34891 35707 35150 35575 34658 36646 35545 36750 35112 35761 37150 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0