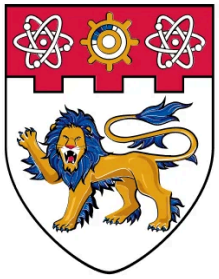


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC3020 Database System Principles

School of Computer Science and Engineering
AY22/23 Semester 2

Project 1

Grp No. : 7

Jeffrey Lim Yi Ren (U2120937B)
Karishein Chandran (U2122012G)
Chen Hongpo (U2022097E)
Don Seo Kang Rui (U2120866H)

Due On: 10 March 2024, 5 pm

Contribution of each group member:

Name	Contribution
Jeffrey Lim Yi Ren	Report, B+ Tree
Karishein Chandran	Report, Storage Implementation
Chen Hongpo	Report, Debugging
Don Seo Kang Rui	Storage and experiments implementation, Report

1. Introduction

This report aims to explain our group's comprehensive design and implementation of a database management system's storage and index components for Project 1 using C++.

The database will be used to contain the IMDb rating and votes information for movies with a title (alphanumeric unique identifier), averageRating (weighted average of all the individual user ratings) and numVotes (number of votes the title has received).

The database functionality will undergo testing through five distinct experiments, each accompanied by its experimental results.

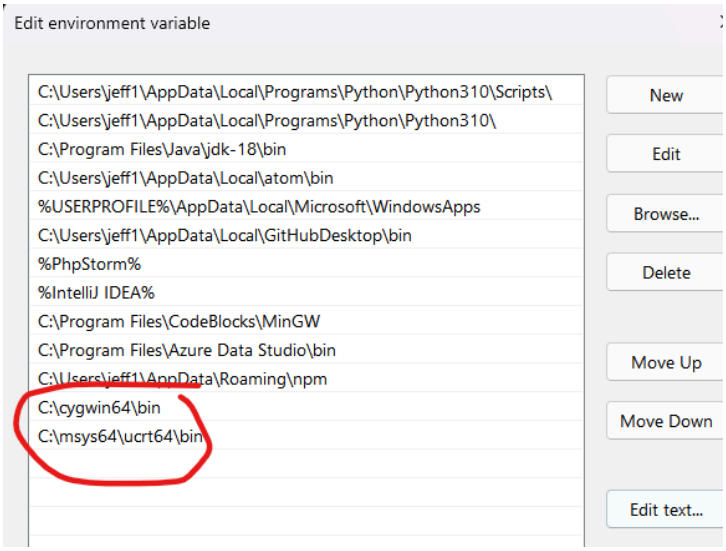
2. Installation Guide

Running our project from Windows VSCode

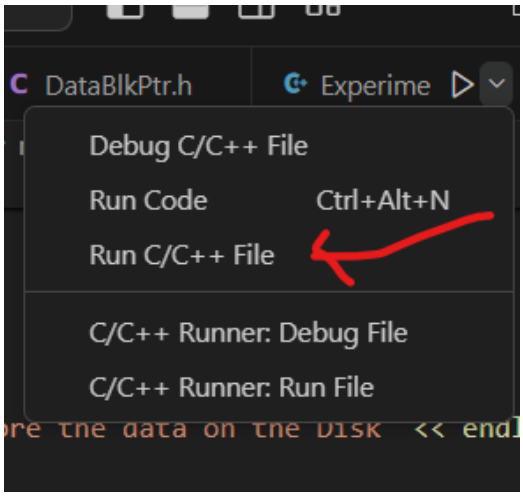
To run directly from VSCode, you may download Mys64's UCRT64 (Universal C runtime) (https://www.msys2.org/) for better compatibility. Follow through the installation on the website. Ensure that you have added MingGW UCRT to the environment path.

Cygwin64 or MingGw Standalone installation from the above installation guide can be added to the environment path if needed to build and compile the C++ program from Terminal.

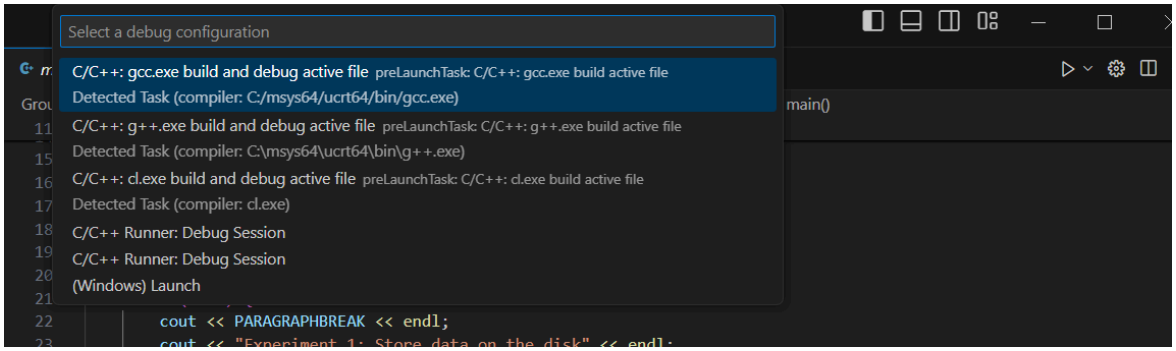
> System Properties > Advanced > Environment Variables > User Variables for UserA > Path > Edit > New > Locate the installation directory containing the C++ files from above



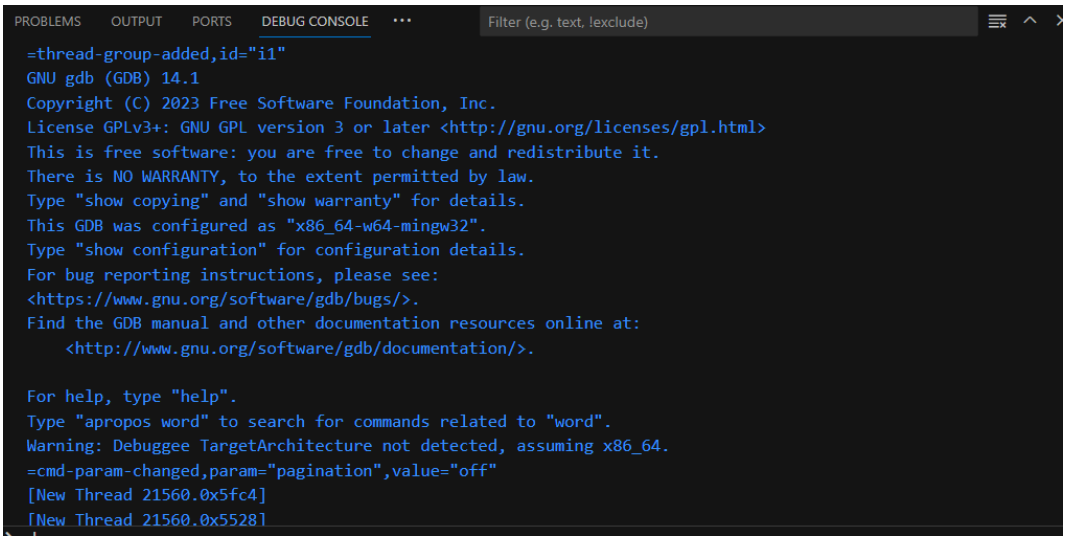
Fire up VSCode, open a folder, and make sure to open it from the **1st folder (Group 7 - Project 1)** you see after unzipping the project. Locate your main.cpp, click on Run C/C++ File for the main.cpp



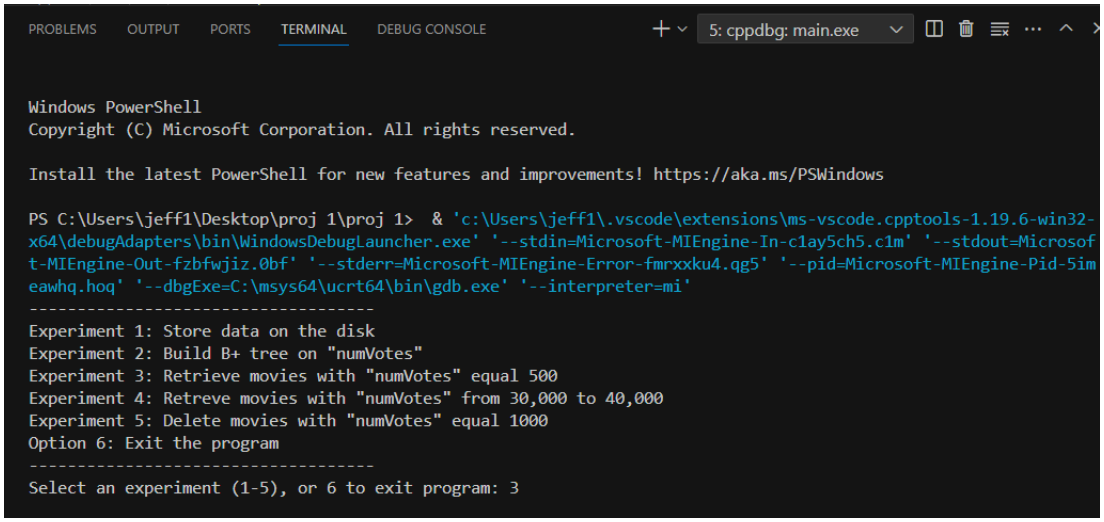
Select **G++.exe build active file**



Your debug console should look something like this



Head back to the Terminal, and you will be greeted with the startup of the programme.



If it fails:

1. Locate .vscode folder in the (Group 7 - Project 1 > .vscode)
2. Create/edit a task.json file
3. Include this line to ensure all C++ files are added to the build

```
"tasks": [
  {
    "args": ["-g", "${workspaceFolder}\\*.cpp", "-o",
"${fileDirname}\\${fileBasenameNoExtension}.exe"],
  }
]
```

3. Design of Storage Component

Data Item to Fields

The IMDB dataset (data.tsv) file contains the data required for the experiments. Each row represents a movie record, and there are approximately 1070318 records in the dataset. Each record (or tuple) consists of three attributes: *tconst, averageRating and numVotes*.

Data Item 1: tconst (string)

This data item is a unique identifier for that particular record. It is an alphanumeric variable with a maximum of 10 characters (e.g. tt0034375). Since it is a fixed length, we stored it in a char array with a length of 11, including the null character.

Data Item 2: averageRating (float)

This data item represents the weighted average of all the individual user ratings for that particular movie. From the dataset, it can be seen that the rating ranges from 0.0 to 10.0, rounded off to one decimal point. The appropriate method to store this variable would be float type.

Data Item 3: numVotes (int)

This data item represents the number of votes the particular movie record has received. From the IMDB dataset, it can be seen that the numVotes are positive numerical values, and the maximum number of digits is 7 (e.g. 2,279,223). The appropriate method to store this variable would be int type.

tconst	averageRating	numVotes
tt0000001	5.6	1645
tt0000002	6.1	198
tt0000003	6.5	1342
tt0000004	6.2	120
tt0000005	6.2	2127

Figure 1: An excerpt of the IMDB dataset movie records

Each data item in the IMDB dataset (*data.tsv*) file is converted to a field using a data structure in the Record class as shown below. There is an additional attribute *isDeleted* included in the record structure as well. This attribute is used specifically to aid in the record insertion and deletion in the B+ tree, by marking the record as deleted, but not actually deleting the record. This is further explained in the *packing records into blocks* section. It is stored as a char variable, where 'y' represents deleted and 'n' represents not deleted.

```
class Record {
    private:
        // LEN = "tt10630794" = 10 + 1
        char tConst[LEN];
        float averageRating;
        int numVotes;
        char isDeleted;
```

Figure 2: The record structure in the Record class

Packing Fields into Records

A record in the IMDB dataset will consist of 3 fields stored in a fixed format with a fixed length after being stored using the record structure. A fixed format with a fixed length field means that all the attributes are present for each tuple, and the attributes (data item) have a fixed length. So, even if a particular data item is smaller than the allocated space (bytes), it will still occupy the same amount of space it was declared to be.

The record (or tuple) can be visualised as follows:

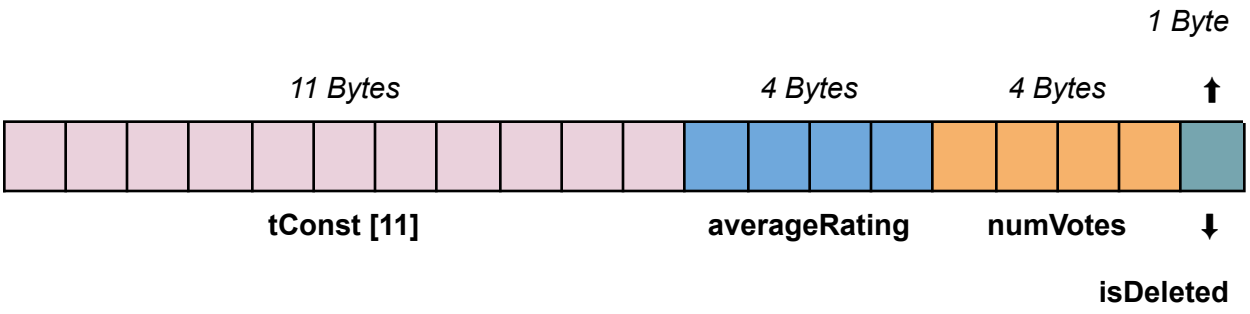


Figure 3: The visualisation of the fields in a record and the space (bytes) allocated

tConst[11] - A char array of fixed length 11 represents this data item. Each character takes up 1 byte in C++, so the total size of tConst is 11 bytes.

averageRating - This data item is represented as a float type (4 bytes in C++).

numVotes - This data item is represented as an int type (4 bytes in C++).

isDeleted - This variable in the record structure is represented as a char type (1 byte in C++).

Each movie record can be initialised as an object using the record structure. The record attributes will contain the four items mentioned above. A movie record object will have a fixed size of 11 + 4 + 4 + 1 = 20 Bytes in the memory.

Packing Records into Blocks

The Block class is used to pack the records into a block. Since we have a fixed block size of 200 bytes, a block can essentially store about ten records. Since all the records have fixed lengths of 20 bytes, they don't need to be separated and are thus stored consecutively.

Block Size = 200 Bytes

20 Bytes	*20 Bytes*	*20 Bytes*	*20 Bytes*	*20 Bytes*	*20 Bytes*	*20 Bytes*	*20 Bytes*	*20 Bytes*	*20 Bytes*

Figure 4: Visualisation of the records in a single block

The Block controller class consists of 4 variables as shown below, and it is used to create a block object.

```
class Block {
    private:
        vector <Record> listOfRecords;
        int blockSize;
        int blockID;
        queue<int> deletedRecords;
```

Figure 5: The block structure in the Block class

1. **vector<Records> listOfRecords:** It is dynamic array that is used to store the record objects.
2. **int blockSize:** Represents the size of the block, which is 200 bytes.
3. **int blockID:** This is used to identify the position of the block in memory.
4. **queue <int> deletedRecords:** This queue stores the record IDs of deleted records to locate available spaces within a block to allocate another record.

The int Block::insertRecord(Record record) method in the Block class is used to pack the records into the created block object. This method will be overseen by the BlockController class as explained below. When the method is invoked, it will determine the precise location within the block for storing the record. The BlockController class also handles deletion through the void Block::deleteRecord(int recordID) method. This method will determine the exact location within a block that needs to be marked as deleted when we want to delete a record.

The BlockController class is used to insert blocks into the memory and delete them from the memory. It also deals with record insertion and deletion. The structure of the BlockController is shown below.

```
class BlockController {
    private:
        int blockSize;
        vector<Block> listOfBlocks;
        list<pair<int, int> > mappingTable;
        queue<int> isEmptyBlocks;
```

Figure 6: The block controller structure in the BlockController class

1. **int blockSize:** Represents the size of the block, which is 200 bytes.
2. **vector<Block> listOfBlocks:** It is dynamic array that is used to store the block objects.
3. **list <pair <int, int> mappingTable>:** This represents a list which stores a pair of block ID and record ID, used to indicate where the record is stored.
4. **queue <int> isEmptyBlocks:** This represents a queue containing the block IDs of those blocks that have available space to allocate more records.

In the insert method *(void * BlockController::insert(string recordString)),* a new record object is created, and the correct block for insertion is determined. If the *isEmptyBlocks* queue is empty, the record will be inserted at the end of the block list if there is space, or a new block will be

initialised if necessary. If the queue is not empty, the record is inserted into the first block of the queue. The method will return the precise location where the record was inserted using the mapping table. This location will consist of the block ID, which represents the block's position, and the record ID, which represents the record's position.

In the deleteRecord method *(void BlockController::deleteRecord(int blockID, int recordID))*, the target record's block ID and record ID are provided as the inputs. The target record is not deleted from memory. Instead, it is marked as deleted, and the block ID this record is currently stored in will be added to the isEmptyBlocks queue. This will signify that space is available in that particular block for additional records. This implementation helps avoid the need to shift existing records or blocks. The mappingTable can also be easily updated to remove the corresponding block ID and record ID pair without adjusting all other existing block IDs and record IDs in the mappingTable.

4. Design of B+ Tree

B+ Tree node data structure

A B+ tree typically consists of the root, internal and leaf nodes. As compared to a traditional B Tree, the B+ tree eliminates the drawback of storing search keys and data in the same nodes by having internal nodes act as the index. This enhances its performance by making insertion and deletion times consistent.

Our implementation of the B+ tree node uses one c++ class, *BPlusNode* to represent all of the root, internal and leaf nodes, with a boolean attribute *isLeaf* distinguishing between leaf and non-leaf nodes. Root and internal nodes are defined the same way as their behaviour do not differ in the making of the B+ tree.

The structure of a B+ tree node consists of these variables:

- **int maxSize -** The maximum number of keys within the node.
- **BPlusNode** ptrs -** Pointers to child nodes.
- **int* keys -** The keys within the node
- **bool isLeaf -** Whether the node is a leaf node (true) or not (false).
- **int numKeys -** Current count of stored keys.
- **vector<DataBlkPtr*> dataPtrs -** Vector of DataBlkPtr utilised by leaf nodes to store pointers to data blocks.

Maximum number of keys

A pointer in C++ takes up to 8 bytes while integers take up to 4 bytes.

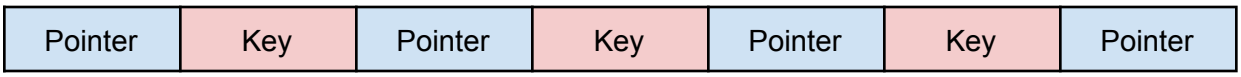


Figure 7: Structure of each node

Since each stored key value is an integer, and there is 1 more pointer than key, the formula for the calculation of number of keys a block is able to store given the block size is as follows:
(*block size* - 8)/(4+8).

Following this formula,

- A block of size 200B can store 16 keys

B+ Tree implementation

The *BPlusTree* class is able to instantiate new instances of the *BPlusNode* class to perform various operations.

The structure of a B+ tree consists of these variables:

- **BPlusNode* root -** Current root node.
- **int maxNodeSize -** Maximum number of keys a node can hold.
- **int blockSize -** Block size(maximum node size), in bytes
- **int numNodes -** Current count of nodes in the B+ tree.
- **int height -** Current height of the B+ tree.
- **int maxDataBlkPtrSize -** Maximum pointers to data blocks that a data block can hold.

Data Block Pointer

Nodes and Data Blocks are not the same, so we decided to create a *DataBlkPtr* class to represent a data block. The *DataBlkPtr* class stores records and allocates a virtual block of main memory for the purpose of storing pointers to each newly inserted record in the database.

Each B+ tree node can hence manage multiple records with identical keys, while ensuring that each vector of data block record pointers adheres to the block size limitation. Like the structure of a linked list, each *DataBlkPtr* object contains a vector of pointers to reference the subsequent *DataBlkPtr* object.

The structure of a Data Block Pointer consists of these variables:

- **maxSize -** Maximum capacity of pointers to records that a data block stores
- **numPtrs -** Current count of pointers within the data block.
- **vector<void*> ptrs -** Vector for storing pointers to the records.
- **DataBlkPtr* next -** Pointer to the next data block in sequence.

Insertion of node

The insertion of a new record is done by the *insert* function within the *BPlusTree* class. Certain conditions are checked for, whereby different insertion processes would be performed.

They are as follows:

1. Leaf node has available space: The new key and record pointer are directly inserted. If the key already exists, no additional key insertion is necessary and the new record pointer is inserted to the corresponding node.
2. Leaf node is full and needs to be split: A new leaf node is generated, and keys are evenly distributed. The *insertChildNode* function is then invoked recursively to insert the new child node into the internal parent node.

Deletion of node

The deletion of an existing record is done by the *delete* function within the *BPlusTree* class. Likewise with insertion, different scenarios require different deletion processes.

They are as follows:

1. Key is not found: The function exits directly.
2. Leaf node retains enough keys after deletion: The function terminates immediately post-deletion.
3. Leaf node lacks enough keys post-deletion and key can be borrowed: A key is borrowed from the left or right sibling leaf node, and the parent node's keys are updated accordingly.
4. Leaf node lacks enough keys post-deletion and key cannot be borrowed: Two leaf nodes are merged, resulting in the removal of one from the tree. Subsequently, the *removeInNode* function is utilised recursively to remove the child node from the parent node.

The number of nodes deleted during this process is returned.

Querying the B+ tree

- Single value query uses the *query* function in the *BPlusTree* class. on the tree will return all corresponding pointers to the records with the queried value.
- Range queries use the *queryRange* function in the *BPlusTree* class. The lower bound of the query range is first identified, following which all pointers are retrieved until the upper bound is reached.

Additional Function to aid Experiment Results

The first 3 functions are used to perform a brute force method , linear search of the blocks of data. The amount of block access will be the total amount of blocks provisioned for the records storage. The 4th is a method we use to calculate the running time for a process.

```
void Experiments::linearScan(int desiredVotes)
```

- Used in Experiment 3, the above function takes in the desiredVotes, iterating from the 1st block through the last block allocated. If the record's numVotes value matches the desiredVotes, get the record's rating to calculate the average rating after.

```
void Experiments::linearScanRange(int minVotes, int maxVotes)
```

- Used in Experiment 4, the above function takes in the range of votes to be included for the linear search function. Iterating from the 1st block through the last block allocated. If the record's numVotes is in the range, get the record's rating to calculate the average rating after.

```
void Experiments::removeByLinearScan(int votesToRemove)
```

- Used in Experiment 5, the above function takes in the votesToRemove, iterating from the 1st block through the last block allocated. If the record's numVotes matches the votesToRemove value, we will increase the removal count.

```
clock_t start = clock();
// Running "Function A" //
clock_t end = clock();
double duration = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
```

- By using the clock() function from <ctime>, we can measure the Processor's running time before we run the function needed to be timed.

5. Experiment Results

NOTE: Please ensure to do the experiment procedurally. You need to build the B+ Tree in experiment 2 before proceeding to do experiment 3, 4, 5

5.1 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 statistic

The number of records;	Number of records: 1070318
The size of a record;	Size of a record (in bytes): 20
The number of records stored in a block;	Number of records stored in a block: 10
The number of blocks for storing the data;	Number of blocks for storing the data: 107032

5.2 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 parameter n of B+ tree: 16
The number of nodes of the B+ tree;	7023
The number of levels of the B+ tree;	The height (level) of B+ tree: 4
The content of the root node (only the keys);	The content of root node (only keys are shown) : || 1342|| 2631|| 3897|| 5970|| 8878||12134|| 20151||30034||49802||125979||

5.3 Experiment 3

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

The number of index nodes the process accesses;	`Total number of nodes accessed: 4`
The number of data blocks the process accesses;	`110`
The average of “averageRating’s” of the records that are returned;	`6.73182`
The running time of the retrieval process (please specify the method you use for measuring the running time of a piece of code);	`25 milliseconds`
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).	No. of data blocks accessed: `107032` Running Time: `441 milliseconds`

Results of the blocks containing Movies with “numVotes” equal to 500

```
 >> Found in blockID: 360 and recordID: 6
tconst    |  averageRating  |  numVotes
tt0013631       6.6     12
tt0013658       6.9     31
tt0013662       6.9     418
tt0013668       6.7     22
tt0013672       6.7     25
tt0013674       7       500
tt0013679       6.9     7
tt0013681       5.6     14
tt0013682       7.5     64
tt0013687       7.1     7
```

```
 >> Found in blockID: 902 and recordID: 9
tconst    |  averageRating  |  numVotes
tt0024550       6.4     24
tt0024551       2.9     9
tt0024553       5.9     137
tt0024554       5.3     822
tt0024555       5.5     195
tt0024558       6.4     11
tt0024559       6.1     140
tt0024560       6.9     397
tt0024561       6.8     500
tt0024562       5.4     10
```

```
 >> Found in blockID: 2278 and recordID: 10
tconst    |  averageRating  |  numVotes
tt0041946       5.5     39
tt0041947       6.1     517
tt0041948       6.6     902
tt0041949       6.3     355
tt0041951       7.4     53
tt0041952       7.6     690
tt0041953       6.9     469
tt0041954       7.3     2435
tt0041955       6.7     1119
tt0041956       6.5     500
```

```
 >> Found in blockID: 2719 and recordID: 7
tconst    |  averageRating  |  numVotes
tt0047355       5.9     99
tt0047356       7       10
tt0047357       6.4     27
tt0047358       5.9     224
tt0047359       6.7     20
tt0047360       4.6     12
tt0047361       7.3     500
tt0047362       5.8     5
tt0047363       6.7     30
tt0047364       4.3     11
```

Method used for measuring running time for a piece of code

```
// Start measuring time
clock_t start = clock();

cout << "- - - - - Content of top 5 index nodes accessed - - - - - " <<  endl;
vector <pair<int, int> > queryResult;
queryResult = this->bTree->query(500);

// Stop measuring time after query.
clock_t end = clock();
double duration = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
```

Brute-force Linear Scan Results

```
-- Brute Force Linear Scan Results --
Average of average ratings of matching records: 6.73182
Number of data blocks accessed: 107032
Running time of the retrieval process: 441 milliseconds
```

5.4 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;	`Total number of nodes accessed: 85`
The number of data blocks the process accesses;	`935`
The average of “averageRating’s” of the records that are returned;	`6.72791`
The running time of the retrieval process	`19 milliseconds`
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).	No. of data blocks accessed: `107032` Running Time: `621 milliseconds`

Top 5 Data blocks access that contains numVotes from 30,000 to 40,0000

```
 >> Found in blockID: 3297 and recordID: 9
tconst    |  averageRating  |  numVotes
tt0054158       5.2     9
tt0054159       7.5     1464
tt0054160       4.7     20
tt0054161       6.6     193
tt0054162       6.8     80
tt0054164       7       470
tt0054165       4.8     36
tt0054166       6.5     15
tt0054167       7.7     30022
tt0054168       5.3     266
```

```
 >> Found in blockID: 1066 and recordID: 3
tconst    |  averageRating  |  numVotes
tt0026776       6.8     73
tt0026777       6.9     15
tt0026778       7.9     30034
tt0026779       5.6     65
tt0026781       6.1     327
tt0026783       6.1     45
tt0026784       6.5     260
tt0026785       5.8     33
tt0026786       5.9     7
tt0026787       6       676
```

```
 >> Found in blockID: 6520 and recordID: 10
tconst    |  averageRating  |  numVotes
tt0091818       5.7     1499
tt0091819       5.4     214
tt0091820       6.5     99
tt0091821       8       280
tt0091823       7.8     974
tt0091824       5.8     69
tt0091825       5.9     89
tt0091826       6.6     25
tt0091827       3.7     589
tt0091828       5.6     30037
```

```
 >> Found in blockID: 77379 and recordID: 10
tconst    |  averageRating  |  numVotes
tt3361618       6.8     5
tt3361630       6.3     7
tt3361638       7.8     91
tt3361644       6.2     41
tt3361702       5.6     82
tt3361726       6.8     11
tt3361740       8       5
tt3361784       7.9     12
tt3361786       8.6     42
tt3361792       6.8     30041
```

```
  >> Found in blockID: 57299 and recordID: 1
 tconst    |  averageRating  |  numVotes
 tt1456941       6.2     30049
 tt1456944       4.2     62
 tt1456946       5.8     8
 tt1456947       7.4     12
 tt1456948       6.2     11
 tt1456949       7.2     1706
 tt1456950       8.6     39
 tt1456953       7.1     17
 tt1456957       3       56
 tt1456958       7.7     10
```

Method used for measuring running time for a piece of code

```
// Start measuring time
clock_t start = clock();

cout << "- - - - - Content of top 5 index nodes accessed - - - - - " <<  endl;
vector <pair<int, int> > queryResult;
queryResult = this->bTree->queryRange(30000, 40000);

// Stop measuring time after query.
clock_t end = clock();
double duration = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
```

Brute-force Linear Search Results:

```
-- Brute Force Linear Scan Results --
Average of average ratings of matching records: 6.72791
Number of data blocks accessed: 107032
Running time of the retrieval process: 621 milliseconds
/// Exited experiment 4
```

5.5 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 of nodes of the updated B+ tree;	7023
The number of levels of the updated B+ tree;	4
The content of the root node of the updated B+ tree(only the keys);	The content of root node of updated B+ tree: || 1342|| 2631|| 3897|| 5970|| 8878||12134||20151||30034||49802||125979||
The running time of the process;	0 milliseconds
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).	No of Data Blocks accessed: 107032 Running time: 523 milliseconds

Method used for measuring running time for a piece of code

```
// Start measuring time
clock_t start = clock();

countDelete = this->bTree->remove(1000);
// Stop measuring time after query.
clock_t end = clock();
double duration = ((double)(end - start) / CLOCKS_PER_SEC) * 1000;
```

Brute-force Linear Search Results:

```
-- Brute Force Linear Scan Results --
Average of average ratings of matching records: 7.22619
Number of data blocks accessed: 107032
Running time of the retrieval process: 523 milliseconds
```