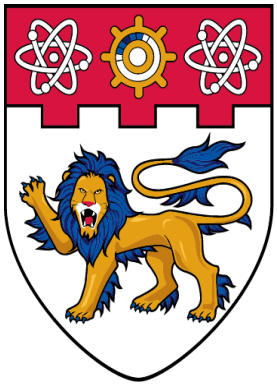

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4031 Database System Principles

Project 1 Report

Group 48

Team members

Lee Jun Wei (U1922896C)

Lim Kai Sheng (U2020321C)

Ma JiaXin (U2022011J)

Yeow Ying Sheng (U1722395B)

Zhu Zeyu (N2202577F)

All members contributed equally to this project

Table of Content

1. Introduction

1.1. Project Overview

This project aims to design and implement a database management system's storage and indexing components. We decided to use C++ as the primary programming language for this project.

Our implementation consists of the following files with corresponding responsibilities:

- disk.cpp, disk.h: contains the implementation of the Disk class for the storage component. It is responsible for reading and writing data to the disk.
- dtypes.h: contains the declaration of custom data structures.
- main.cpp: contains the main function. It is responsible for executing the Experiments.
- tree_display.cpp: contains the implementation of the display functions, such as printing the tree nodes.
- tree_insert.cpp: contains the implementation of the insertion functions. It is responsible for inserting key-pointer pairs into the B+ tree.
- tree_remove.cpp: contains the implementation of the removal functions. It is responsible for removing key-pointer pairs from the B+ tree.
- tree_search.cpp: contains the implementation of the search functions. It is responsible for searching keys in the B+ tree.
- tree.cpp, tree.h: contains the implementation of the B+ tree, including some utility functions.

We employed the object-oriented programming model to structure this assignment into smaller components, each with its attributes and functions. This makes the "code" mode understandable, as each object now serves to carry out its purpose. The objects are then integrated with the "main" algorithm for the respective experiments.

2. Storage Component

2.1. Dataset

A brief analysis of the dataset was conducted to identify its characteristics.

Field	Type	Min length	Max length	Min value	Max value
tconst	String	9	10	-	-
averageRating	Decimal	-	-	1.0	10.0
numVotes	Integer	-	-	5	2279223

2.2. Record Structure

For simplicity, we have decided to adopt a design with fixed-sized records. Although using fixed-sized records may incur some unused space at the end of each block, the implementation is much simpler and easier to understand. Additional memory to indicate the size of each field is not required (required for variable-length fields). Furthermore, it also enables us to retrieve records via its block and record indexes easily.

A Record structure was defined to represent the structure of a record in memory:

```
struct Record {
   char tconst[11];  // 11 bytes
   unsigned char averageRating;  // 1 byte
   unsigned int numVotes;  // 4 bytes
};
```

Figure 1: Structure of a record

Data type of fields:

- tconst: A character array of size 11 was used to represent this field as the maximum length of the tconst attribute in the dataset is 10. For it to be a valid string, it must be suffixed by an additional null byte.
- averageRating: To promote memory efficiency, we multiply the value by 10 and used the unsigned char data type (range of 0-255) for this field, instead of the float or double data types.
- numVotes: As all values of numVotes are positive integers, the unsigned int data type was selected for this field.

2.2. Disk Structure

The storage component is implemented in the Disk class. When it is instantiated, a contiguous block of memory is allocated in the heap to be used as disk storage. The disk instance can then be used to store records from the dataset.

The internal structure of the Disk class is shown in the following code snippet:

```
class Disk {
private:
   size_t blockSize;
   size_t diskSize;
   size_t blockIdx;
   size_t recordIdx;
   unsigned char *pMemAddress;

   size_t maxRecordsPerBlock;
   size_t maxBlocksInDisk;

public:
   // constructor
   Disk(size_t aDiskSize, size_t aBlockSize);

   // functions
   Record *insertRecord(const std::string &tconst, unsigned char avgRating, int numVotes);
   Record *getRecord(size_t aBlockIdx, size_t aRecordIdx);
   void printRecord(Record *record);
   size_t getBlockId(Record *record);
   void printBlock(size_t aBlockIdx);
   size_t getBlocksUsed();
}
```

Figure 2: Internal structure of the Disk class

2.1.1 Instance attributes

Each Disk instance comprises the following instance attributes:

- diskSize: User specified disk size
- blockSize: User specified block size
- pMemAddress: Pointer to store the allocated memory on the heap
- maxRecordsPerBlock: Maximum number of records that can fit in a block
- maxBlocksInDisk: Maximum number of records that can fit in a block
- blockIdx: Current number of blocks used
- recordIdx: Current number of records used

2.1.2 Functions

The Disk class has the following functions:

- insertRecord: Inserts the new record in the current location pointed by the indexes and increments blockIdx and recordIdx. If the current block is full after the insertion, the indexes are updated to point to the next block.
- getRecord: Calculates and returns the memory address of a record in the disk, given a blockIdx and recordIdx.
- printRecord: Prints the three attributes of a given record.
- getBlockId: Calculates and returns the block ID that a given record resides in.
- printBlock: Prints the tconst attribute of all records in a block.
- getBlocksUsed: returns the number of blocks used so far.

2.3. Organization of Records in Disk

The following is a simplified diagram of how records are packed into a block, and how blocks are packed into the disk:

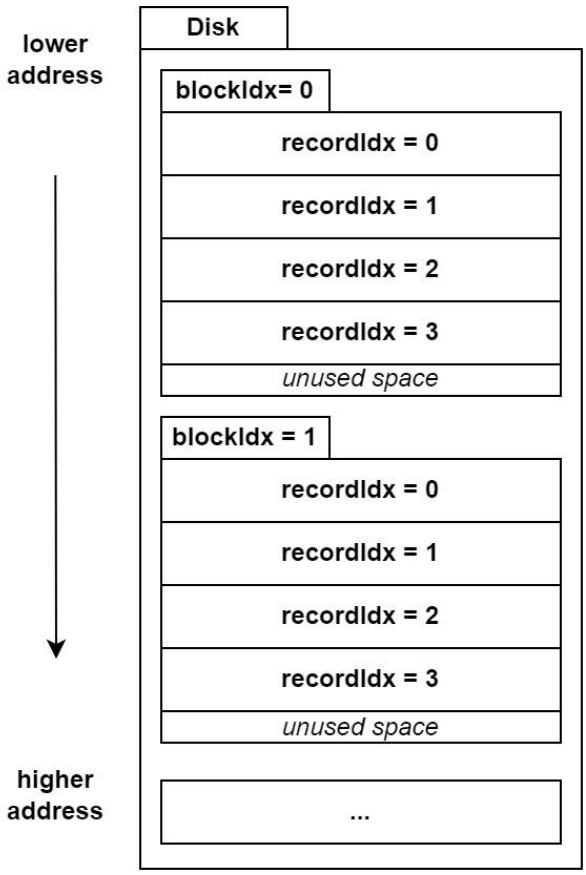

Figure 3: Disk structure

The disk instance maintains a running blockIdx and recordIdx indexes, and increments them accordingly as records are inserted into the disk. When a block is full, the blockIdx is incremented to point to the next block, and recordIdx is reset to 0.

With this design, we are able to easily compute the memory address to store each new record as they are inserted (figure 4). Retrieval is easy as well.

```
Record *Disk::getRecord(size_t aBlockIdx, size_t aRecordIdx) {
    /*
     * Returns a pointer to a record!
     *
     * The offset consists of two parts:
     * -> blockIdx x BLOCK_SIZE: offset to the start of a specified block
     * -> recordIdx x recordSize: offset to the start of a record in a block
```

```
     */
    size_t offset = (aBlockIdx * blockSize) + (aRecordIdx * sizeof(Record));
    return reinterpret_cast<Record *>(pMemAddress + offset);
}
```

Figure 4: Calculation of memory address based on blockIdx and recordIdx

3. Indexing Component

This section introduces the indexing component, implemented using a B+ tree.

3.1. Node Structure

The structure of a B+ tree node is shown in the following code snippet:

```
class Node {
public:
   bool isLeafNode;
   std::vector<int> keys;
   Node *pNextLeaf;

   union ptr {
       std::vector<Node *> pNode;
       std::vector<std::vector<Record *>> pData;

       ptr();
       ~ptr();
   } pointer;

   friend class Tree;

public:
   Node();
};
```

Figure 5: Structure of the Node class

3.1.1 Instance attributes

The explanation of key attributes is as follows:

- isLeafNode: Indicates if the node is a leaf node.
- keys: A vector that contains all the keys in the node.
- pNextLeaf: A pointer points to the next leaf node on the right, only applicable to leaf nodes.
- pointer:
    - pNode: A vector of child Node pointers, only applicable to internal nodes.
    - pData: A vector of vectors that contains record pointers, only applicable to leaf nodes.

3.2. Tree Structure

The structure of the Tree class is shown in the following code snippet:

```
class Tree {
private:
   int maxInternalChild;
   int n;
   int nodesAccessedNum;
   Node *rootNode;

   void insertInternal(int x, Node **currentNode, Node **child);

   Node **findParentNode(Node *currentNode, Node *child);

public:
   explicit Tree(int blockSize);

   Node *getRoot();

   int countNodes();

   int countHeight();

   int getMaxInternalChild();

   int getN();

   int getNodesAccessedNum();

   void setNodesAccessedNum(int setNumber);

   void setRoot(Node *);

   void displayCurrentNode(Node *currentNode);

   std::vector<Record *> *search(int key, bool printNode);

   Node *searchNode(int key, bool printNode);

   void insert(int key, Record *pRecord);

   void removeKey(int key);

   void removeInternal(int x, Node *currentNode, Node *child);

};
```

Figure 6: Structure of the Tree class

3.2.1 Computing the n attribute

When a tree is instantiated, the n attribute (maximum number of keys in a node) of the tree is computed. As each node needs to be bounded by the block size, we need to ensure the n attribute is computed correctly.

The following assumptions were made:

- A pointer is 8 bytes (64-bit address)
- Each key is 4 bytes (unsigned int)

As each node has n keys and $n + 1$ pointers, the size of one node can be represented as:

$$Size\ of\ a\ Node\ =\ p\ +\ n\,(\,p\ +\ k\,)$$

Thus, the n attribute can be calculated with:

$$n\ =\ (block\ size\ -\ p)\ /\ (\,p\ +\ k\,)$$

With a block size of 200 bytes:

$$n\ =\ (200\ -\ 8)\ /\ (8\ +\ 4)\ =\ 16$$

With a block size of 500 bytes:

$$n\ =\ (500\ -\ 8)\ /\ (8\ +\ 4)\ =\ 41$$

3.2.2 Instance attributes

The explanation of key attributes is as follows:

- maxInternalChild: The maximum number of children a node can have.
- maxN: The maximum number of keys a node can have.
- nodesAccessedNum: The number of index nodes accessed during a search.
- rootNode: The root node of the tree.

3.2.3 Functions

The explanation of key functions is as follows:

- insertInternal: Inserts a key into an internal node.
- findParentNode: Finds the parent of a node.
- tree: The constructor of the tree.
- getRoot: Returns the root node of the tree.
- countNodes: Counts the number of nodes in the tree.
- countHeight: Counts the height of the tree.
- getMaxInternalChild: Returns the maximum number of children a node can have.

- getN: Returns the n attribute.
- getNodesAccessedNum: Returns the number of index nodes accessed during a search.
- setNodesAccessedNum: Sets the number of index nodes accessed during a search.
- setRoot: Sets the root node of the tree.
- displayCurrentNode: Displays a single node.
- search: Searches the tree for a key and returns a vector of records.
- searchNode: Searches the tree for a key and returns the leaf node.
- insert: Inserts a key-pointer pair into the tree.
- removeKey: Removes a key from the tree.
- removeInternal: Removes a key from an internal node.

3.3. Data Pointers at the Leaf Node

To deal with the existence of identical “numVotes" values in records of the dataset, we have decided to use a vector to store a list of record pointers at the leaf node, like a chain. This allows us to avoid having duplicate keys in the B+ tree.

This means that data pointers at the leaf nodes will point to a vector of record pointers, instead of a single record pointer. A simplified diagram of this approach is shown in Figure 7 below.

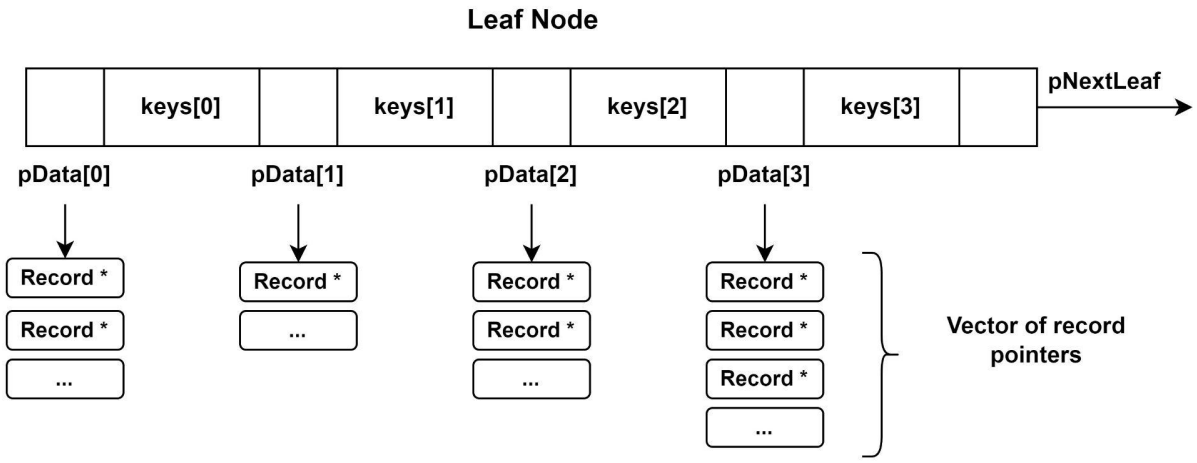

Figure 7: Leaf Node

This approach has two main advantages. First, when we search the tree for records with a particular “numVotes” value, we only need to traverse to the leaf node with the desired key and access all the records in the corresponding vector. There is no need to traverse across multiple successive leaf nodes. Another advantage is that when we insert key-pointer pairs or remove a key from the tree, lesser key-pointer pairs needs to be manipulated, and significantly reducing the operations needed to update the tree.

4. Experiments

4.1. Experiment 1

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

1. The number of blocks used
2. The size of the database (in terms of MB)

Block Size	No. of Blocks Used	Size of Database (MB)
200 bytes	89,194	17.8387 MB
500 bytes	34,527	17.2635 MB

4.2. Experiment 2

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

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

Block Size	Parameter n	Number of Nodes	Height	Contents of root node and its first child node
200 bytes	16	1,747	4	*Refer to Figure 8*
500 bytes	41	658	3	*Refer to Figure 8*

```
EXPERIMENT 1 & 2 (Block Size: 200B)
Inserting records from the data file into disk and building index...
 -> No of records processed: 1070318
 -> No of blocks used: 89194 blocks
 -> Size of the database (blocks used x blockSize): 17838800 bytes
 -> Parameter N of the B+ Tree: 16
 -> No of nodes in the B+ Tree: 1747
 -> Height of the B+ Tree: 4
 -> Content of rootNode: {1342, 2631, 3897, 5970, 8878, 12134, 20151, 30034, 49802, 125979}
 -> Content of rootNode's first child node: {106, 267, 448, 573, 726, 837, 960, 1122, 1224}
```

```
EXPERIMENT 1 & 2 (Block Size: 500B)
Inserting records from the data file into disk and building index...
 -> No of records processed: 1070318
 -> No of blocks used: 34527 blocks
 -> Size of the database (blocks used x blockSize): 17263500 bytes
 -> Parameter N of the B+ Tree: 41
 -> No of nodes in the B+ Tree: 658
 -> Height of the B+ Tree: 3
 -> Content of rootNode: {810, 1970, 2998, 3824, 4754, 5795, 6963, 8596, 10593, 12944, 15653,
19079, 23444, 28454, 33701, 41156, 50539, 72884, 106475, 209225}
 -> Content of rootNode's first child node: {26, 59, 88, 120, 160, 188, 222, 253, 291, 316,
338, 360, 395, 418, 444, 484, 506, 528, 553, 575, 611, 652, 676, 699, 727, 754, 780}
```

Figure 8: Program output of Experiments 1 and 2

4.3. Experiment 3

We retrieve the records with “numVotes” equal to 500 and report the following statistics:

1. The number and the content of index nodes the process accesses.
2. The number and the content of data blocks the process accesses.
3. The average of “averageRating” of the records returned.

Block Size	No. of Index Nodes accessed	No. of Data blocks accessed	No. and content of Index/ Data blocks accessed	Average of “averageRating”
200 bytes	4	110	*Refer to Figure 9*	6.73182
500 bytes	3	110	*Refer to Figure 9*	6.73182

```
EXPERIMENT 3 (Block Size: 200B)
 -> Index Nodes accessed:
{1342, 2631, 3897, 5970, 8878, 12134, 20151, 30034, 49802, 125979}
{106, 267, 448, 573, 726, 837, 960, 1122, 1224}
{458, 468, 478, 487, 501, 514, 525, 538, 552, 561}
{487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500}
 -> No of Index nodes accessed: 4
 -> Data Blocks accessed:
Contents of Data block (blockIdx=299):
tt0013627 tt0013629 tt0013631 tt0013658 tt0013662 tt0013668 tt0013672 tt0013674 tt0013679
tt0013681 tt0013682 tt0013687
Contents of Data block (blockIdx=751):
tt0024553 tt0024554 tt0024555 tt0024558 tt0024559 tt0024560 tt0024561 tt0024562 tt0024563
tt0024564 tt0024567 tt0024568
Contents of Data block (blockIdx=985):
tt0028271 tt0028272 tt0028273 tt0028274 tt0028275 tt0028276 tt0028277 tt0028278 tt0028279
tt0028280 tt0028281 tt0028282
Contents of Data block (blockIdx=1898):
tt0041953 tt0041954 tt0041955 tt0041956 tt0041957 tt0041958 tt0041959 tt0041961 tt0041962
tt0041963 tt0041966 tt0041967
Contents of Data block (blockIdx=2265):
tt0047355 tt0047356 tt0047357 tt0047358 tt0047359 tt0047360 tt0047361 tt0047362 tt0047363
tt0047364 tt0047365 tt0047366
 -> No of Data blocks accessed: 110
 -> No of unique Data blocks accessed: 110
 -> Average of averageRating: 6.73182
```

```
EXPERIMENT 3 (Block Size: 500B)
 -> Index Nodes accessed:
{810, 1970, 2998, 3824, 4754, 5795, 6963, 8596, 10593, 12944, 15653, 19079, 23444, 28454,
33701, 41156, 50539, 72884, 106475, 209225}
{26, 59, 88, 120, 160, 188, 222, 253, 291, 316, 338, 360, 395, 418, 444, 484, 506, 528, 553,
575, 611, 652, 676, 699, 727, 754, 780}
{484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501,
```

```
502, 503, 504, 505}
 -> No of Index nodes accessed: 3
 -> Data Blocks accessed:
Contents of Data block (blockIdx=115):
tt0013538 tt0013540 tt0013555 tt0013556 tt0013570 tt0013571 tt0013572 tt0013573 tt0013574
tt0013579 tt0013590 tt0013592 tt0013596 tt0013597 tt0013603 tt0013607 tt0013611 tt0013615
tt0013617 tt0013619 tt0013620 tt0013624 tt0013626 tt0013627 tt0013629 tt0013631 tt0013658
tt0013662 tt0013668 tt0013672 tt0013674
Contents of Data block (blockIdx=290):
tt0024517 tt0024518 tt0024519 tt0024523 tt0024524 tt0024527 tt0024531 tt0024532 tt0024534
tt0024535 tt0024536 tt0024537 tt0024538 tt0024539 tt0024542 tt0024545 tt0024546 tt0024547
tt0024548 tt0024549 tt0024550 tt0024551 tt0024553 tt0024554 tt0024555 tt0024558 tt0024559
tt0024560 tt0024561 tt0024562 tt0024563
Contents of Data block (blockIdx=381):
tt0028257 tt0028258 tt0028259 tt0028260 tt0028264 tt0028267 tt0028268 tt0028269 tt0028270
tt0028271 tt0028272 tt0028273 tt0028274 tt0028275 tt0028276 tt0028277 tt0028278 tt0028279
tt0028280 tt0028281 tt0028282 tt0028283 tt0028284 tt0028285 tt0028286 tt0028287 tt0028288
tt0028289 tt0028290 tt0028291 tt0028292
Contents of Data block (blockIdx=734):
tt0041925 tt0041926 tt0041928 tt0041929 tt0041930 tt0041931 tt0041932 tt0041933 tt0041934
tt0041935 tt0041938 tt0041939 tt0041940 tt0041943 tt0041944 tt0041945 tt0041946 tt0041947
tt0041948 tt0041949 tt0041951 tt0041952 tt0041953 tt0041954 tt0041955 tt0041956 tt0041957
tt0041958 tt0041959 tt0041961 tt0041962
Contents of Data block (blockIdx=876):
tt0047324 tt0047325 tt0047326 tt0047327 tt0047328 tt0047329 tt0047330 tt0047331 tt0047333
tt0047334 tt0047335 tt0047336 tt0047337 tt0047338 tt0047339 tt0047340 tt0047341 tt0047342
tt0047343 tt0047345 tt0047348 tt0047349 tt0047351 tt0047353 tt0047355 tt0047356 tt0047357
tt0047358 tt0047359 tt0047360 tt0047361
 -> No of Data blocks accessed: 110
 -> No of unique Data blocks accessed: 110
 -> Average of averageRating: 6.73182
```

Figure 9: Program output of Experiment 3

4.4. Experiment 4

We retrieve the records with “numVotes” from 30,000 to 40,000 (both inclusive) and report the following statistics:

1. The number and the content of index nodes the process accesses.
2. The number and the content of data blocks the process accesses.
3. The average of “averageRating” of the records returned.

Block Size	No. of Index Nodes accessed	No. of Data blocks accessed	No. and content of Index/ Data blocks accessed	Average of “averageRating”
200 bytes	86	953	*Refer to Figure 10*	6.72791
500 bytes	38	953	*Refer to Figure 10*	6.72791

```
EXPERIMENT 4 (Block Size: 200B)
 -> Index Nodes accessed (first 5):
{1342, 2631, 3897, 5970, 8878, 12134, 20151, 30034, 49802, 125979}
{20937, 21812, 22631, 23444, 24703, 26119, 27317, 28769}
{28916, 29004, 29116, 29268, 29387, 29594, 29633, 29743, 29848, 29959}
{29959, 29962, 29974, 29975, 29978, 29982, 29988, 29996, 30022}
{30034, 30037, 30041, 30049, 30053, 30056, 30078, 30081, 30085, 30090}
 -> No of Index Nodes Accessed: 86
 -> Data Blocks accessed:
Contents of Data block (blockIdx=2747):
tt0054162 tt0054164 tt0054165 tt0054166 tt0054167 tt0054168 tt0054169 tt0054170 tt0054171
tt0054172 tt0054173 tt0054174
Contents of Data block (blockIdx=887):
tt0026769 tt0026771 tt0026772 tt0026773 tt0026774 tt0026775 tt0026776 tt0026777 tt0026778
tt0026779 tt0026781 tt0026783
Contents of Data block (blockIdx=5433):
tt0091825 tt0091826 tt0091827 tt0091828 tt0091829 tt0091830 tt0091831 tt0091832 tt0091833
tt0091834 tt0091835 tt0091836
Contents of Data block (blockIdx=64482):
tt3361702 tt3361726 tt3361740 tt3361784 tt3361786 tt3361792 tt3361794 tt3361812 tt3361814
tt3361834 tt3361856 tt3361874
Contents of Data block (blockIdx=47748):
tt1456915 tt1456931 tt1456937 tt1456939 tt1456941 tt1456944 tt1456946 tt1456947 tt1456948
tt1456949 tt1456950 tt1456953
 -> No of Data blocks accessed: 953
 -> No of unique Data blocks accessed: 932
 -> Average of averageRating: 6.72791
```

```
EXPERIMENT 4 (Block Size: 500B)
 -> Index Nodes accessed (first 5):
{810, 1970, 2998, 3824, 4754, 5795, 6963, 8596, 10593, 12944, 15653, 19079, 23444, 28454,
33701, 41156, 50539, 72884, 106475, 209225}
{28603, 28890, 29099, 29297, 29730, 29956, 30158, 30457, 30658, 30865, 31144, 31440, 31607,
```

```
31768, 32028, 32195, 32470, 32644, 33044, 33357}
{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, 30240, 30246, 30247, 30248, 30254, 30259,
30262, 30275, 30319, 30326, 30333, 30341, 30354, 30361, 30370, 30376, 30391, 30395, 30402,
30418, 30423, 30431, 30446, 30453, 30456}
{30457, 30458, 30462, 30468, 30492, 30516, 30522, 30530, 30540, 30547, 30548, 30550, 30552,
30554, 30569, 30571, 30576, 30578, 30585, 30605, 30608, 30611, 30619, 30620, 30621, 30639}
 -> No of Index Nodes Accessed: 38
 -> Data Blocks accessed:
Contents of Data block (blockIdx=1063):
tt0054151 tt0054152 tt0054153 tt0054154 tt0054155 tt0054156 tt0054157 tt0054158 tt0054159
tt0054160 tt0054161 tt0054162 tt0054164 tt0054165 tt0054166 tt0054167 tt0054168 tt0054169
tt0054170 tt0054171 tt0054172 tt0054173 tt0054174 tt0054175 tt0054176 tt0054177 tt0054178
tt0054179 tt0054180 tt0054181 tt0054182
Contents of Data block (blockIdx=343):
tt0026754 tt0026755 tt0026756 tt0026757 tt0026758 tt0026759 tt0026760 tt0026761 tt0026762
tt0026766 tt0026768 tt0026769 tt0026771 tt0026772 tt0026773 tt0026774 tt0026775 tt0026776
tt0026777 tt0026778 tt0026779 tt0026781 tt0026783 tt0026784 tt0026785 tt0026786 tt0026787
tt0026788 tt0026789 tt0026790 tt0026791
Contents of Data block (blockIdx=2103):
tt0091821 tt0091823 tt0091824 tt0091825 tt0091826 tt0091827 tt0091828 tt0091829 tt0091830
tt0091831 tt0091832 tt0091833 tt0091834 tt0091835 tt0091836 tt0091837 tt0091839 tt0091841
tt0091843 tt0091844 tt0091845 tt0091846 tt0091847 tt0091848 tt0091850 tt0091851 tt0091852
tt0091853 tt0091854 tt0091857 tt0091858
Contents of Data block (blockIdx=24960):
tt3361332 tt3361338 tt3361356 tt3361358 tt3361380 tt3361400 tt3361428 tt3361436 tt3361490
tt3361532 tt3361556 tt3361572 tt3361576 tt3361578 tt3361580 tt3361584 tt3361586 tt3361588
tt3361590 tt3361614 tt3361618 tt3361630 tt3361638 tt3361644 tt3361702 tt3361726 tt3361740
tt3361784 tt3361786 tt3361792 tt3361794
Contents of Data block (blockIdx=18483):
tt1456903 tt1456912 tt1456913 tt1456915 tt1456931 tt1456937 tt1456939 tt1456941 tt1456944
tt1456946 tt1456947 tt1456948 tt1456949 tt1456950 tt1456953 tt1456957 tt1456958 tt1456961
tt1456963 tt1456964 tt1456966 tt1456967 tt1456970 tt1456971 tt1456974 tt1456975 tt1457043
tt1457045 tt1457062 tt1457065 tt1457089
 -> No of Data blocks accessed: 953
 -> No of unique Data blocks accessed: 911
 -> Average of averageRating: 6.72791
```

Figure 10: Program output of Experiment 4

4.5. Experiment 5

We delete records that has “numvotes” equal to 1,000, update the B+ tree accordingly, and report the following statistics:

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

Block Size	No. of times node is deleted	No. of Nodes	Height of B+ tree	Contents of root node and its first child node
200 bytes	0	1,747	4	*Refer to Figure 10*
500 bytes	0	658	3	*Refer to Figure 10*

```
EXPERIMENT 5 (Block Size: 200B)
Removed '1000' from the B+ Tree successfully!
 -> No of times that a node is deleted (or two nodes are merged): 0
 -> No of nodes in the updated B+ tree: 1747
 -> Height of the updated B+ tree: 4
 -> Content of rootNode: {1342, 2631, 3897, 5970, 8878, 12134, 20151, 30034, 49802, 125979}
 -> Content of rootNode's first child node: {106, 267, 448, 573, 726, 837, 960, 1122, 1224}
```

```
EXPERIMENT 5 (Block Size: 500B)
Removed '1000' from the B+ Tree successfully!
 -> No of times that a node is deleted (or two nodes are merged): 0
 -> No of nodes in the updated B+ tree: 658
 -> Height of the updated B+ tree: 3
 -> Content of rootNode: {810, 1970, 2998, 3824, 4754, 5795, 6963, 8596, 10593, 12944, 15653,
19079, 23444, 28454, 33701, 41156, 50539, 72884, 106475, 209225}
 -> Content of rootNode's first child node: {26, 59, 88, 120, 160, 188, 222, 253, 291, 316,
338, 360, 395, 418, 444, 484, 506, 528, 553, 575, 611, 652, 676, 699, 727, 754, 780}
```

Figure 10: Program output of Experiment 5

5. Installation guide

5.1 CLion

Requirements: You need to have CLion installed.

1. Open the root folder in CLion.
2. During the first load, CLion should detect the CMake file and display a prompt. Simply accept the defaults.
3. Build and run the project.

5.2 g++

Requirements: You need to have g++ installed.

1. Open a terminal in the same directory as the source files.

2. Compile the program with the g++ command.

   ```
   g++ main.cpp disk.cpp tree.cpp tree_display.cpp tree_insert.cpp
   tree_remove.cpp tree_search.cpp -o main
   ```

3. Run the program.

   Windows:
   ```
   $ main.exe
   ```

   Unix/Linux:
   ```
   $ ./main
   ```