Ονοματεπώνυμα και ΑΜ

Αντώνιος Καρβελάς 1115201600060

Παντελεήμων Μαλέκας 1115201600268

Θεοφάνης Μπιρμπίλης 1115201600110

Here's all the information regarding what was implemented in the AM.c:

Regarding the array of the open files we used a global array, named tableOfIndexes. Each item in this array consists of a struct named openAM, which contains:

- char attrType1 -> The type of the key
- int attrLength1 -> The size of the key type
- char attrType2 -> The type of the value
- int attrLength2 -> The size of the value type
- int root -> The root of the B+ tree(updates every time it is needed)
- int fd -> file descriptor
- int dataSize -> how many elements can be stored in data block
- int indexSize -> how many elements can be stored in index block

For the open scans we used a global array, named tableOfScans. Each item in this array consists of a struct named scan. Scan struct is used to keep track of important information regarding each open scan such as:

- openAM -> openAM struct as explained above
- operation -> the operation partaking in the scan
- blockNumber -> the block of the last value we found
- positionInBlock -> position of the last value we found
- value -> here we"ll store the desired value if we find it
- key -> the value we will be searching with

Here's how our B+ tree is defined: The are two types of blocks: index and data. The first block contains our metadata. Each index block consists of: a character "I" which states that it's an index block, a counter of how many elements it has stored and the index elements. Each index element is the key value and the block numbers of the previous and next block for each key. Left block numbers contain values less than the key and right block numbers contains values equal or greater than the key.

Each data block consists of: a character “D” which states that it’s a data block, a counter of how many elements it has stored, the data elements and the block number of the next data block (-1 if there isn’t any). Each data element is the key value and the corresponding data value.

Extra functions used throughout the program:

howManyInDataBlock(openAM* currentAM)

Calculate how many elements fit in a data(leaf) block.

howManyInIndexBlock(openAM* currentAM)

Calculate how many elements fit in an index block.

keyCompare(openAM* bplus, void* value1, void* keyValue)

This function compares value1 with keyValue. It returns a negative number if value1 is less than the key, 0 on equality or a positive integer otherwise. If an error occurs it will return the appropriate error message.

recursiveInsert(openAM* bplus, int currentBlockNum, void *value1, void *value2, void* returnKey)

RecursiveInsert adds the new item and returns 0 if no split occurred or the block number of the new created block. If we’re on an index block: we will go to each element and use keyCompare to find out where we will go next. If keyCompare returns negative, go to left pointer. If no pointer visited, go to last pointer. So after we get the pointer we need we will call recursiveInsert and get its result. If it’s not 0, which means no split happened, we will check where our new item has to be placed. If index split is necessary, we use a buffer which temporarily stores all the items of the block about to be splitted (every item it already had and the new one that caused the split). So we get the half point of the buffer and we copy the first half on the already existing block and we create a new index block to place the second half. If no split is necessary, we place the new item at the already existing index block. If no split happened return 0. Otherwise, if we’re on a data block: we get the position where the new item will be inserted using keyCompare. If split is necessary we use a buffer, same as in the index case, to temporarily store all the items of the block about to be splitted. After getting the half point of the buffer and we copy the first half on the already existing block and we create a new data block to place the second half. If no split is necessary, we place the new item at the already existing data block.

recurseSearchFirst(openAM* bplus, void* key, int block)

This function finds the first block that is needed for the scans in FindNextEntry. If we're on an index block, we find the next block to go to with keyCompare and call the function again. If we reach a data block, we return it.

findNext(int position, int block, openAM* bplus, scan* currentScan)

This function is used in findNextEntry. If something is found it returns 0, else 1. It keeps moving in every item in the data block until we find the desired value.

AM_printSudo(int fileDesc, int blockNum)

Function that prints every item in the B+ tree. This function isn't called anywhere but it can be used for debugging purposes. (Intended use: AM_printSudo(filename, -1))

AM functions that were implemented:

1)AM_Init()

AM_Init() initialises every entry of our global tables(to NULL) as well as the BF block level.

2)AM_CreateIndex(char *fileName, char attrType1, int attrLength1, char attrType2, int attrLength2)

AM_CreateIndex() checks if the attribute types are valid. If there's a problem it will return the appropriate error message. If they're valid, we check if the file already exists using the access function (from unistd.h) and print the error message. If everything's fine, the B+ file is created, and the metadata is placed inside the file. First, a "BP" string is placed to signify that it's a B+ tree file. First block also includes the attributes with their lengths and the root.

3)AM_DestroyIndex(char *fileName)

Remove function (from unistd.h) is used to remove the file from the disk.

4)AM_OpenIndex(char *fileName)

This function opens the B+ file and checks if we have a B+ file and whether it can be opened or not. If so, an openedIndex struct is allocated and it's placed in the array of open files. The function also returns the index of the file inside the array.

5)AM_CloseIndex(int fileDesc)

This function finds the file in the array of open files and closes the B+ file using its file descriptor.

6)AM_InsertEntry(int fileDesc, void *value1, void *value2)

This function adds a new entry in our B+ tree. If the tree is empty, we create the root (one leaf block). Otherwise, we go through B+ tree using the recursiveInsert function. If split occurred we also create a new root.

7)AM_OpenIndexScan(int fileDesc, int op, void *value)

This function creates a new scan. If we have reached max open scans, it returns an error. Otherwise, it finds the first empty position in table of open scans and places the new scan struct. Also, we find the first block where the search in AM_FindNextEntry will begin and we return the index of the scan inside the array.

8)AM_FindNextEntry(int ScanDesc)

Returns the value we need using the findNext function

9)AM_CloseIndexScan(int ScanDesc)

This function frees the opened scan and any other allocated memory.

10)AM_PrintError(char *errString)

Here we print the string in the errString argument as well as the error corresponding to the AM_errno variable.

11)AM_Close()

Closing the BF block level

Changes done in AM.h:

In AM.h we added the new error macros for the errors that may occur throughout the program. AM_EOF is already defined as -1. We added:

AME_PARAMETER -2 //Error in parameters.
AME_EXISTS -3 //B+ File exists.
AME_MAX_FILES -4 //Max files opened reached.
AME_NOT_BP -5 //Opened file not B+ File.
AME_MAX_SCANS -6 //Max opened scans reached.
AME_INVALID_TYPE -7 //Invalid type in attrType1.
AME_DELETE_ERROR -8 //Error occurred while deleting file.
AME_INSERT_ERROR -9 //Error in insertion.
BF_BLOCK_LEVEL_ERROR -10 //Error occurred in a BF function.