CMPE321 Project 4 Report

Berfin Şimşek 2018400009
Kardelen Demiral 2018400027

June 4, 2022

Contents

1 Introduction

We implemented a basic database management system for our CMPE321-Introduction to Database Systems assignment 4. It includes 3 definition language operations which are "create a type", "delete a type" and "list all types", and 6 manipulation language operations which are "create a record", "delete a record", "search a record", "search for a record", "update a record", "list all records" and "filter records". The records created and stored for the associated types are the members of Horadrim game.

2 Assumptions & Constraints

2.1 Assumptions

- All field and type names are alphanumeric.
- All fields are alphanumeric.
- User always enters alphanumeric characters inside the test cases.
- All fields' length is smaller than or equal to 20 characters.
- A type can have at most 20 fields.
- The length of a type can be at most 20 characters.
- The length of a field name can be at most 20 characters.

2.2 Constraints

1. B^+-Tree should be used for indexing. Every type has its own B^+-Tree. For search and filter operations, these trees should be used. Giving the primary key of a record, one should be able to retrieve the address that corresponding record is stored in from that type's B^+-Tree. These trees must be stored in separate files in the system. In every delete and create record operations, the corresponding trees must be updated.
2. The data must be organized in pages and pages must contain records.
3. The system must be able to create new files as Horadrim grows. So, every record of a type should not be stored in a single massive file.
4. The file should be read page by page when reading records.

3 Storage Structures

3.1 System Catalog

We have a SystemCatalog.csv file storing the information about the types. Every line in this file includes the attribute names and types, number of attributes, primary key order and the latest file name storing the records of the corresponding type. Everytime a type is created, it is added to the System Catalog. You can see an example structure for our System Catalog below:

angel	['3', '1', 'name', 'str', 'alias', 'str', 'affiliation', 'str']	angel_1.txt
evil	['4', '1', 'name', 'str', 'type', 'str', 'alias', 'str', 'spell', 'str']	evil_1.txt

3.2 B^+-Tree

We used the B^+-Tree implementation in the link `https://gist.github.com/benben233/2c8a2a8ab44a7beabad0df1b6658232e/`. It's structure is just as given in the lecture slides. Every type has its B^+-Tree stored in a file named "bTreetypename.txt". The items in the B^+-Tree's are stored in these text files in the json format and they are retrieved from there with "createBTrees()" function every time the program starts. They are saved again with "saveBTrees()" function every time the program terminates. So the most recent information about the records and their addresses are stored in these files.

3.3 File Design

Every type has its own files which include the records of that type. These files are named as "typename_x.txt". x is an integer. When a file is full and a new record of that type is being added, a new file is created and the x value is incremented for the name of that file. Every file includes 10 pages.

3.4 Page Design

Every page has a size of 2KBytes. According to the record size, the number of records in a page varies depending on the type. Every page has a page header which has a size 1 byte. There is a "0" in the page header if the page is not full yet and there is a "1" if it is full.

3.5 Record Design

Every record in a page has a size of 20*(number of fields in the type)+1 bytes. So, we used a fixed sized fields approach. The remaining part of every field is filled with spaces. That "+1" corresponds to the record header. Similar to the page header, there is a "0" in the record header if there is a record in the corresponding record location and there is a "1" if there is a record there.

4 Operations

4.1 Horadrim Definition Language Operations

4.1.1 Create Operation

For the create operation, we have a function called "createType". First, it checks whether this type has created previously with using "getAllTypeNames" function. This function reads the system catalog line by line and returns a list storing the existing type names. If this type has created previously, "createType" function returns false. Otherwise, it creates a B+ tree and adds this tree to a dictionary called bTrees with key as type name, value as a tree object. For B+ tree, we used the implementation of B+ tree from this link `https://gist.github.com/benben233/2c8a2a8ab44a7beabad0df1b6658232e/`. After that, it adds the type to the system catalog and creates 2 new files. First one is for records and the other one is for the B+ tree of that type. The name of the file in the system catalog that stores the records changes when the file is full, its ID is increased by one. For example, if type angel is created, "angel_1.txt" is created for records. Then, if this file is full, a new file called "angel_2.txt" is created. After that, it fills the record file with zeros and spaces according to the calculations. Records are filled with empty spaces of length (20*number of fields of the type) and records have 0 in the beginning as record header. If the record is full, it is 1. Every page of a file filled with records and number of records are calculated with math.floor((page size-1) / (length of a record+1)) and pages have 0 in the beginning as page header which shows whether the page is full or not. If it is full, header is 1. Since page size is not always a multiple of the record size, the remaining part of the page is calculated and that number of spaces is added to the end of every page. So, every page starts at the multiples of the PAGESIZE, which is 2000.

4.1.2 Delete Operation

For the delete operation, we have a function called "deleteType". First, it checks whether this type exists or not with "getAllTypeNames" function. If there is no type with this name, it returns false. Otherwise, it deletes its B+ tree from the bTrees dictionary and all the files belongs to this type such as the files that contain its records and the file that contains its B+ tree. Finally, it deletes this type from system catalog.

4.1.3 List Operation

For the list operation, we have a function called "listType". We used "getAllTypeNames" function for getting the names of types as a list. After calling this function, it writes each element to output file. Also, it checks whether the list which "getAllTypeNames" function returns is empty or not. If it is empty, it means there is no type with this name and it returns false.

4.2 Horadrim Manipulation Language Operations

4.2.1 Create Operation

For the create operation, we have a function called "createRecord". Firstly, it checks whether the type of the record exists or not with "getAllTypeNames" function. If it doesn't exist, it returns false. Otherwise, it retrieves the name of the file where the new record should be stored from the system catalog. Then, it checks whether this record was created previously or not looking at the corresponding type's tree. If it is, it returns false. Also, it checks whether the number of the fields given in the input is the same as it is defined as in createType operation by retrieving this information from the system catalog. After error checking, it formats the length of the fields to 20 since we used fixed-length field approach. Then, it reads the file byte by byte until it finds an empty page by checking page headers and reads the page until it finds an empty record by checking record headers. If it finds an empty record, it writes the fields into the file and stores its address in the B+ tree with its primary key as key, address as value. Address is a string which consists of the file name where record is stored and the byte where the record is located in the file. If it cannot find an empty page, it means the file is full and a new file should be created. After a new file is created, it is filled with zeros and empty spaces just as in the create type operation and it writes the record to the first record place because the file is empty. Then again it stores its address and changes the system catalog so that the new file name is stored as the file where new records should be stored in.

4.2.2 Delete Operation

For the delete operation, we have a function called "deleteRecord". Firstly, it checks whether the type of the record exists or not with "getAllTypeNames" function. If it doesn't exist, it returns false. Then it checks every leaf of the B+ tree of the type to find the record. If it cannot find the primary key of the record in the leaves, it returns false. Otherwise, it gets the address of the record with "getAddress" function. Since we stored the exact byte of the record in the address it was very easy to delete the record. It gets the file name and the byte from the address and replace the record header of the record and page header of the page that the record is stored with 0. Then deletes the record's primary key from the B+ tree of its type.

4.2.3 Update Operation

For the update operation, we have a function called "updateRecord". Firstly, it checks whether the type of the record exists or not with "getAllTypeNames" function. If it doesn't exist, it returns false. Then it checks every leaf of the B+ tree of the type to find the record. If it cannot find the primary key of the record in the leaves, it returns false. Otherwise, it gets the address of the record with "getAddress" function. After error checking, it formats the length of the

fields to 20 and writes the new fields with retrieving its location from address variable.

4.2.4 Search Operation

For the search operation, we have a function called "searchRecord". Firstly, it checks whether the type of the record exists or not with "getAllTypeNames" function. If it doesn't exist, it returns false. Then it checks every leaf of the B+ tree of the type to find the record. If it cannot find the primary key of the record in the leaves, it returns false. Otherwise, it gets the address of the record with "getAddress" function and number of fields of its type by reading system catalog. We need number of fields because we should know how many bytes we should read from the file to retrieve the data. Then, it finds the file name and the starting byte of the record and reads the file from this byte until record length which is computed as (number of fields* 20). It returns the result as string to main and it is written the output file in the main. Since we used this function a lot, we preferred to return the result as a string.

4.2.5 List Operation

For the list operation, we have a function called "listRecord". Firstly, it checks whether the type of the record exists or not with "getAllTypeNames" function. If it doesn't exist, it returns false. Then it iterates every leaf of the B+ tree of the type and stores its keys in a list. If the list is empty, it means this type has no records and it returns false. Otherwise, it calls "searchRecord" function for every key and writes the result of the function to output file.

4.2.6 Filter Operation

For the filter operation, we have a function called "filterRecord". Firstly, it checks whether the type of the record exists or not with "getAllTypeNames" function. If it doesn't exist, it returns false. Then it iterates every leaf of the B+ tree of the type and stores its keys in a list and if the key is an integer it changes its type to integer for comparing reasons then sort them in ascending order. After that, it calls "searchRecord" function for every key according to the type of the comparing operator. Then writes the result to the output file.

5 Conclusion & Assessment

First of all, it was a very beneficial and fun project although we've gone thorough many challenges. Thanks to this project, we are now able to understand better how database management systems work internally.

The most difficult parts for us to understand were the concept of files, pages and records and also writing/reading data page by page/in bytes. We were confused about what to store in the system catalog or in the page/record headers.

After figuring all that out and making a logical design, it was not an overhead to write the code.

I think our design made the coding part easier since we used fixed-length fields approach and stored the exact byte addresses in the trees so that reading a record was just a few lines of code.

However, there are also things that can be improved about our project but due to the time restriction, we didn't have the time to fix them. For example, our code is very redundant. We could've written many functions for basic operations like retrieving the address of a record. We repeated so many parts so our code became very long.