Spring 2022 CmpE 321 Project 4: Horadrim

Halil Burak Pala - 2019400282
Berat Damar - 2018400039

June 4, 2022

Contents

1 Introduction

In this project, we designed a Database Management System. We designed a storage structure for our database. In this storage structure, every **type** which is consisted of **field**s and refers to *relation*s in actual database systems has some **record**s. These records are stored in **page**s, and pages are stored in **file**s. There is a corresponding **B+ Tree index** for every type. These indexes serve as a pointer for our records and by using these indexes, we can easily access records. Also we store information about fields of types in **system catalog**s so that we can perform operations easily.

The design supports some Data Definition and Data Manipulation operations. This system, as definition operations, supports *creating a new type*, *deleting a type* and *listing all types* operations. As manipulation operations, it supports *creating a record for a type*, *deleting a record of a type*, *searching for a record*, *updating a record*, *listing all records of a type* and *filtering records of a type by a constraint* operations. These operations are inspected in more detail in later sections of this report.

In the implementation, we created the storage structure and stored our data in this structure. We created system catalog for our system and we logged every operation in a log file. We used an external implementation for B+ Tree which can be found here. More detailed information can be found in the later sections of this report.

2 Assumptions & Constraints

2.1 Assumptions

1. A type can have a maximum of 12 fields.
2. A field of a type can be maximum of 20 bytes.
3. A page includes 8 records.
4. A file includes 4 pages.
5. All fields are alphanumeric. Also, type and field names are alphanumeric.
6. User writes input statements in correct forms. (Misspellings are not checked.)
7. User always enters alphanumeric characters inside the test cases.
8. Paths of the input files are given as a parameter to the comand while running the program.
9. Names of the output files should be given as a parameter. Output files are assumed to be in the same directory with the program.
10. The program should be run with the command:

```
python3 horadrimSoftware.py <path-of-the-input-file>
<name-of-the-output-file>
```

11. Names of the input files are in the form : `input_<no>.txt`. e.g.: `input_23.txt`

2.2 Constraints

1. We used B+ Tree for indexing.
   (a) We used primary keys of types for the search-keys in trees.
   (b) We have separate B+ Tree for each type.
   (c) With every create record and delete record operations, the corresponding tree is updated.
2. Our data is organized in files and pages. Pages include records. More information can be found below.
3. When it is needed, only a single page is loaded in the memory. The whole file is not loaded.
4. *File*s, csv files and binary tree related files are created in the same directory of the program.
5. Names of the *file*s are in the form: `file<no>` without any extension (they are binary files).

3 Storage Structures

Here is the structures we used and created to implement this project.

3.1 System Catalog Design

Our system catalog is a csv file consisting of information about the types and their primary keys. Fields of the catalog and an example system catalog is as follows:

Type Name	**Primary Key Order**	**Primary Key Data Type**
angel	1	str
evil	1	str
potion	1	int

Table 1: An example system catalog

3.2 B+ Tree Design

B+ Trees are index structures used to facilitate equality and range searches in a database system. They serve as a map to reach our data more easily.

Each node of a B+ tree contains an ordered list of keys and pointers to lower level nodes in the tree. These pointers can be thought of as being between each of the keys. To search for or insert an element into the tree, one loads up the root node, finds the adjacent keys that the searched-for value is between, and follows the corresponding pointer to the next node in the tree. Recursing eventually leads to the desired value or the conclusion that the value is not present.

B+ trees use some balancing mechanisms to provide that all of the leaves are always on the same level of the tree. Therefore, it increases efficiency of tree functionality.This B+ tree does not contains record directly. It contains pointers to actual record locations. We use B+ tree pointers to store file,page and slot of data. Therefore, when we search a record with key ;we can get location of the record in efficient manner.

We used external *bplustree* package to use B+ Tree indexes in our project. Here is the link to the Phyton Package Index of that package.

By the help of this package, we easily created and modified B+ Trees. In our design, for every type, we created and maintained a separate B+ Tree index.

Essentially, this library creates a B+ Tree index in a given file and maintains it. Some important classes and methods implemented by this library are:

- `BPlusTree(filename,page_size,order,key_size,value_size,serializer)` is the constructor of the BPlusTree index.
  - It creates a B+ Tree index in given file with name `filename`.
  - `pagesize` is the size of a single page in bytes in the index and it is set to 4096 by default.
  - `order` is the number of record entries in a single page. It is 100 by default.
  - `key_size` is the size of a key in a record entry in bytes. It is 8 by default.
  - `value_size` is the size of the value of a record entry in bytes. It is 32 by default.
  - `serializer` is a `Serializer` class of this library indicating the data type of the keys of a record entry. If keys of the entries are string, then it should be `StrSerializer()`, if they are string, then it should be `IntSerializer()`. It is `IntSerializer()` by default.
  - After a `BPlusTree` object is created, a WAL file is created and every inserts to this index is stored in this file. After that object is closed with the function `close()`, these insertions are occured in the original file.
- `insert` function inserts a record entry to a tree index: `tree.insert(<key>,<value>)`. Instead of this function, brackets can be used: `tree[<key>] = <value>`. Brackets allow updating a record entry.

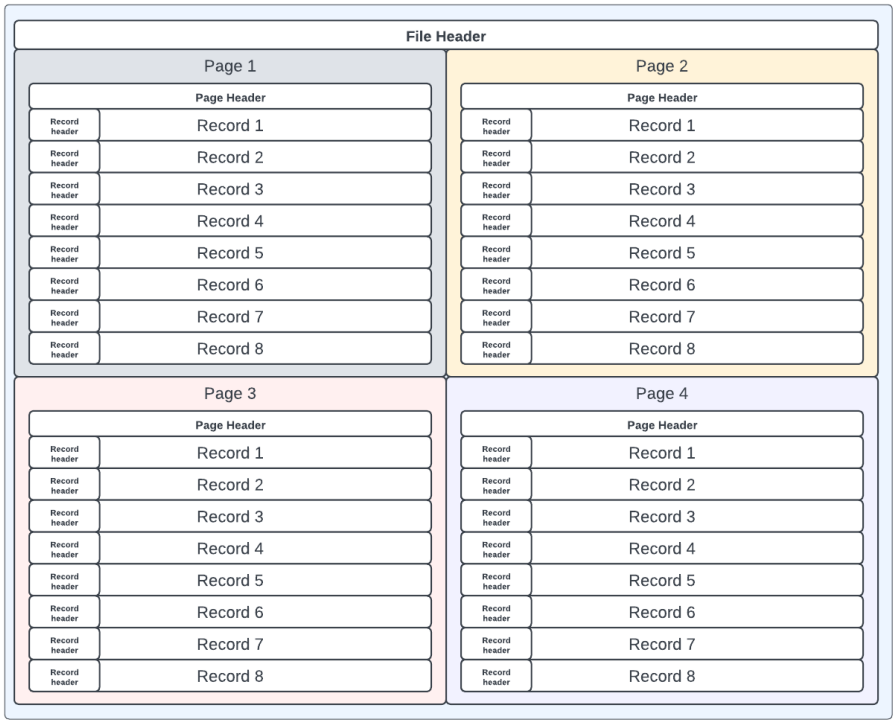
- `<key>` can be a string or an integer based on the serializer specified in the constructor of the index. In our project, these keys are the primary keys of the records of a type.
- `<value>` must be a byte array. In this project, we stored the location of the record specified in the key in our database structure. The value is in the form: `b"<file-id>-<page-id>-<record-slot-in-the-page>"`. For example `b"3-4-2"` indicates the record in the 2nd record slot of the 4th page of the 2nd file.

In this B+ Tree implementation, there is no delete functionality. So, to indicate a deletion in the page, we set the value of a record entry to `b"-1"`.

3.3 Database Design

In our database design, we organized the data in files, pages and records.

Figure 1: Representation of the structure of a single file



3.3.1 Record Structure

We implemented fixed-length record approach in our project. Every record in our structure consists of a record header and record data which is a collection

of field values. As we stated earlier, we assumed that a record can have at most **12** fields and every field value can be at most **20** bytes.

- Every record header is **1** byte. This byte can have three values:
  - If it is **0**, then it means that this record is full and contains an active data.
  - If it is **1**, then it means that this record is deleted from the database.
  - If it is **2**, then it means that this record is empty. All of the record headers are set as **2** when a new file is created.
- Every record data is **240** bytes. Every 20 bytes of this record data includes a field value. This is because a record can have a maximum of 12 fields and every field value is at most 20 bytes.
- So, one single record is **241** bytes.

3.3.2 Page Structure

Every page in our design has a page header and 8 records.

- Every page header is **1** byte. It indicates how many records are in use in that page. So, this byte can have values 1,2,3,4,5,6,7 and 8. If it is 8, then that means this page is full and no new record can be inserted.
- Every page data includes 8 records. So size of the data of a single page is $241 \cdot 8 =$**1928** bytes.
- So, a single page is a total of **1929** bytes

3.3.3 File Structure

We implemented unordered (heap) file structure in our project. Every file in our design has a file header and 4 pages.

- Every file header is **1** byte. It indicates which page is available for an insert at that moment. This byte can have 5 values: 1,2,3,4 and 0. If it is 0, then that means this file is full and no new record can be inserted to any page in that file. In such a case, this file should be deleted.
- Every file data includes 4 records. So, size of the data of a single file is $1929 \cdot 4 =$**7716** bytes.
- So, a single file is a total of **7717** bytes.
- In our program, in case of a deletion, we check the file which that deletion occurred. If there is no active record in that file, we deleted the file.

4 Operations

4.1 Horadrim Definition Language Operations

We have three types of Horadrim Definition Language Operations: Create a type, delete a type and list all types.

- Create a type: If the operation is "create type", we divided given line as type name, number of fields and primary key order.If there is already a type with the given name, we create a Fail log. If operation is not fail,information about this type is inserted into the system catalog. Finally, A new B+ tree is created for this type. Since operation is completed successfully, success(true) log is created on log file.(Related code **horadrimSoftware.py** line[197:234])

- Delete a type: Firstly, we took type name from line. If there is no a type with given name, Fail log is created. If there is a type with given name, we traverse B+ tree of this type and find location of all records using B+ tree.Then, we set the record header of these records as 1 which indicates the record is deleted. After we set header of these records, we check if all records in the file of deleted record is deleted. If so, we deleted the file. After that step, we deleted the B+ tree of the type.Finally, we deleted the type information in the system catalog. (Related code **horadrimSoftware.py** line[234:308])

- List all types: Firstly, we checks number of types. If there is no type, we created a Fail log. If it is not fail, we simply write all types to output file.(Related code **horadrimSoftware.py** line[308:323])

4.2 Horadrim Manipulation Language Operations

We have six types of Horadrim Manipulation Language Operations: Create a record, Delete a record, Search for a record (by primary key), Update a record (by primary key), List all records of a type and Filter records (by primary key).

- Create a record: Firstly, we took type name and fields from given line and then got primary key.Secondly, we checks that if given type is already exist. If it tries to create a record for a non-existing type, we created a Fail log. Another control mechanism is creating a record with a primary key of an existing record. We create a Fail log in that case.For this purpose, we traverse corresponding B+ tree for a record. If there is no problem with these cases, we get available file number and we check file header: If file is full, we created new file.After that, we get the available page in the file to the memory and then we get number of record slots in use from the page header(Page header includes the number of occupied record slots). Then we calculate the available slot number We create the record. Then we write it to the page in memory and update page header. If page is full after this insertion,we update the file header.After page related tasks

are completed, we write page in memory back to its file.Finally, we ınsert record id to the B+ Tree. Record id includes the location of the record in our file structure.(Related code **horadrimSoftware.py** line[323:430])

- Delete a record: Firstly, we take type name and primary key from given line.Secondly, we get file and page location of the record from B+ Tree. If record is non-existing(deleted or not created before), we create a Fail log.If operation is not fail, we get corresponding page and then we mark record header as 1 which indicates the record is deleted.After that, we deleted related index in B+ tree.Finally, we check file if all records are deleted by inspecting record headers. If all of the records are deleted, we delete the file.(Related code **horadrimSoftware.py** line[430:504])

- Search for a record: Firstly, we take type name and primary key of the record.Secondly, we traverse B+ tree and we get file no, page id, slot from B+ tree.If there is no such record in the tree(deleted or not created before), we create a Fail log.If there is no failure, program bring the page to the memory,go the corresponding record location and get the record fields.Finally, we write the field values to the output file and then write page back to the file.(Related code **horadrimSoftware.py** line[554:594])

- Update a record: Firstly, we take type name, primary key and fields from given file.Secondly, we traverse B+ tree and we get file no, page id, slot from B+ tree. Program bring the page to the memory and overwrite the corresponding record in page and write page back to the file.(Related code **horadrimSoftware.py** line[504:554])

- List all records of a type: Firstly, we take type name of records that will be listed from given file.If there is no type with the given name, we created a Fail log for that process.If there is no fail, we perform search operation on files using every node in the tree of the given type.(Related code **horadrimSoftware.py** line[594:625])

- Filter records: We have a assumption: Condition statements are given in the correct form always: <**primary-key-name**> <**condition-operator**> <**query-value**>. Firstly, we parse given line as type name and condition.If there is no type with the given name, we create a Fail log for that query. If not fail,we create boolean condition variable for the key of the tree (which is the primary key) and we get the location information of the record for given condition statement.If the condition is satisfied and the record is not deleted perform search operation described above.(Related code **horadrimSoftware.py** line[625:685])

Note: For all successful operations, we create a success log.

5 Conclusion & Assessment

We think that we design our database properly as explained in description. We used files, pages and record slots to implement search, create, delete,list functionalities etc. Our headers of pages and files are good to get necessary information about structure. We did not arrange file after deleting a record in middle of file.We used another bit to show that this record is deleted until all records in the file will be deleted. This may be down of our design.