CMPE321: Introduction to Database Systems, Spring 2022
Project 4: Project Horadrim

Ali Kaan Biber, 2018400069
Yavuz Samet Topcuoglu, 2019400285

June 1, 2022

Contents

1 Introduction **3**

2 Assumptions & Constraints **4**

2.1 Assumptions 4

2.2 Constraints 4

3 Storage Structures **4**

3.1 System Catalogue 4

3.2 File Design 5

3.3 Page Design 5

3.4 Record Design 5

3.5 B^+-Tree Design 6

4 Operations **7**

4.1 HALO Definition Language Operations 7

4.2 HALO Manipulation Language Operations 8

5 Conclusion & Assessment **9**

1 Introduction

Aside from the story, the project is all about designing a database system -a design that have a catalog and data storage units- and of course the implementation of it. The design supports several operations such that:

- Creating and deleting a type
- Listing all types in the system
- Creating and deleting a record with given type
- Listing all records of a type in the system
- Searching and updating a record with primary key
- Filtering all records of a type with given condition

They are explained in detail in chapter 4 of the report.

The program takes input as a .txt file and gives a couple of files as outputs. The outputs files can be:

- output.txt: The file where the output of listing, searching and filtering results are stored. Overwrites every time when program runs.
- types$x$.txt: The file(s) where all types in the system are stored. x is a natural number which denotes the numbering of types files. It is needed because a new types file is created when all available slots are filled with *type* records in a file. Not overwrites but appends, same files are used for different runs of the program.
- records$x$.txt: The file(s) where all records in the system are stored. x is a natural number which denotes the numbering of records files. It is needed because a new records file is created when all available slots are filled with *record* records in a file. Not overwrites but appends, same files are used for different runs of the program.
- B+*type*.txt: The file(s) where all records of the type *type* are stored. These files used for indexing. Not overwrites but appends, same files are used for different runs of the program.
- horadrimLog.csv: The file where all logs are stored for each input line. Not overwrites but appends, same file is used for different runs of the program.

There are some assumptions made when implementing the project, and they are explained in chapter 2 with the constraints.

Since this project is a database system design project, the information given in chapter 3 of this report is the most important chapter of all in our opinion as it describes the design choices and storage structures.

2 Assumptions & Constraints

2.1 Assumptions

- All fields shall be alphanumeric. Also, type and field names shall be alphanumeric.
- User always enters alphanumeric characters inside the test cases.
- The hardware of Horadrim center and Horadrim instances will be built according to the blueprints, thus you do not need to consider the Horadrim physical storage controller to interact with the storage units.
- Note that type refers to a relation.
- Maximum number of fields is 12.
- Maximum length of entries to fields and type names is 20.

2.2 Constraints

- B+Tree is used for indexing.
  - Primary keys of types is used for the search-keys in trees.
  - There are separate B+-Trees for each type, and these trees must be stored in file(s). So, when a type is created, its B+-Tree will also be created and its structure is stored.
  - With every create record and delete record operations, the corresponding tree is updated.
  - B+-Trees are not generated from scratch on the fly using records for DML operations. The tree structures that are stored in files are utilized.
- The data records are organized in pages and pages contain records.
- All pages are not stored in the same file and a file contains multiple pages. This means that the system creates new files as Horadrim grows. Moreover, when a file becomes free due to deletions, that file is deleted.
- Although a file contains multiple pages, it reads page by page when it is needed. Loading the whole file to RAM is not done, and also not necessary.

3 Storage Structures

3.1 System Catalogue

- We have two different types of files which are *types* and *records*.

- Note that system catalogue is itself a file, and throughout this report we call it as *types* files.
- Further explanation is given below.

3.2 File Design

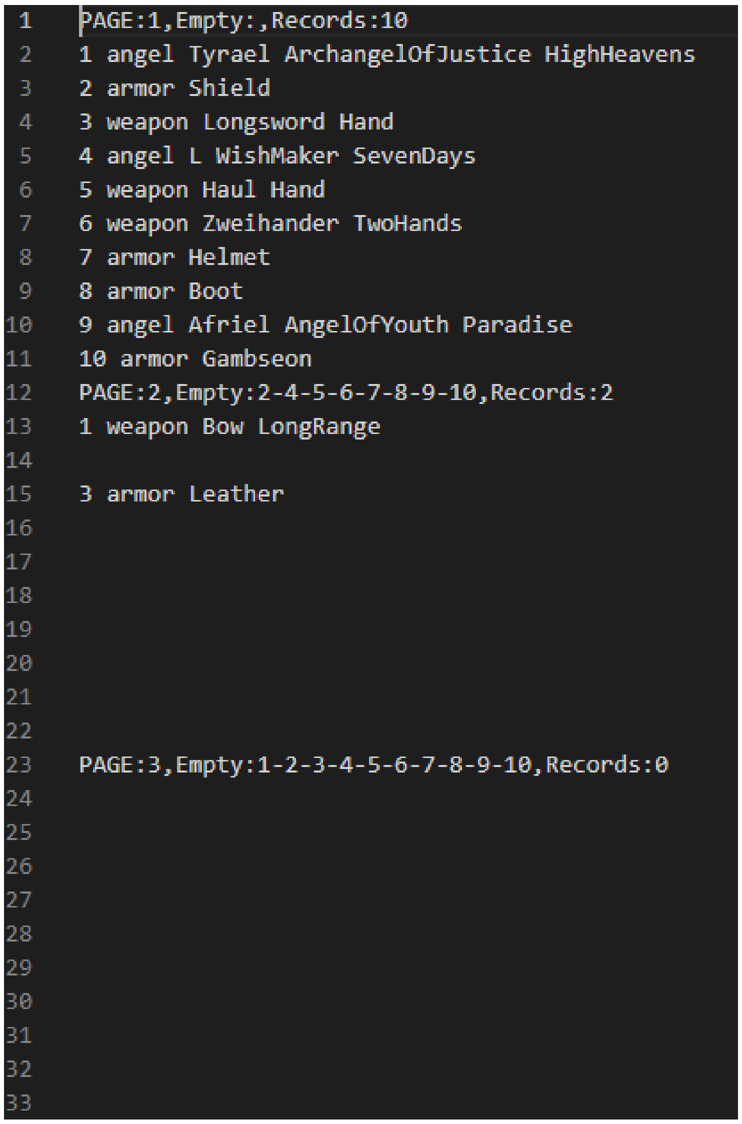
- *Types* and *Records* files consist of 3 pages.
- Files do not have any header. Any information needed for a file can be fetched from its 3 pages.
- A new record is inserted to the first available file. For example, insertion into *records1.txt* over *records2.txt* given that both files exist at the time of insertion.

3.3 Page Design

- All pages consist of 11 lines, which 10 of them store records and the other is the header.
- Page headers contain three information:
  1. Page Number in that file.
  2. Empty slots from 1 to 10 in which records can be placed.
  3. Number of Records in the page.
- A new record is inserted to the first available spot in a page. For example, insertion into slot 3 over slot 6
- Page headers consist of 89+1 bytes of fixed-length where +1 is the newline.

3.4 Record Design

- Records consist of Record ID, *Type*, and fields.
- Fields of the records always start with the primary key implementation purposes.
- Data records consist of 12*20+1 bytes of fixed-length where +1 is the newline, 12 is the maximum number of fields, and 20 is the maximum length of field names.



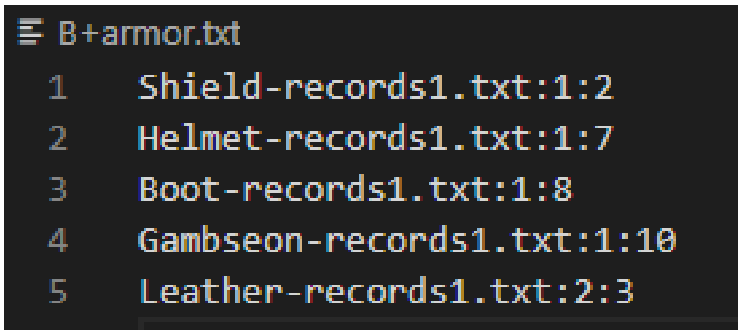
```
PAGE:1,Empty:,Records:10
1 angel Tyrael ArchangelOfJustice HighHeavens
2 armor Shield
3 weapon Longsword Hand
4 angel L WishMaker SevenDays
5 weapon Haul Hand
6 weapon Zweihander TwoHands
7 armor Helmet
8 armor Boot
9 angel Afriel AngelOfYouth Paradise
10 armor Gambseon
PAGE:2,Empty:2-4-5-6-7-8-9-10,Records:2
1 weapon Bow LongRange

3 armor Leather







PAGE:3,Empty:1-2-3-4-5-6-7-8-9-10,Records:0










```

Figure 1: A File with 3 Pages and 12 Records in total. First Page is full, Second Page has two records, and Third Page is empty.

3.5 B^+-Tree Design

- The search keys and the data of a record in the leaf nodes are stored in B^+*type* files, and separated by - respectively.
- The data in the leaves contain three information separated by a single colon:

1. The name of the file where the record is stored.
2. Page Number in that file.
3. The Record ID in that page.



```
B+armor.txt
Shield-records1.txt:1:2
Helmet-records1.txt:1:7
Boot-records1.txt:1:8
Gambseon-records1.txt:1:10
Leather-records1.txt:2:3
```

Figure 2: B^+armor file is an example B^+*type* file where the information about how to find a record is stored.

4 Operations

After reading the line from the input file, we tokenize the line and make the necessary operations accordingly.

For each line of input, after the operations are performed, we determine if the operation was successful or not. According to that decision we log them into the .csv file with UNIX time.

4.1 HALO Definition Language Operations

- Create a Type: In our design we are storing the information about types in a file called types$x$.txt. If there is no types file we create one. If there is at least one types file already we check if the type name exists in one of them. If so this is considered as a failure. If type is new to the system we create its B+tree file as well. Next, we get the fields and field types from the input then put them in the first available types file.
- Delete a Type: To delete a type, type must be created before. First we check the types files for existence of that particular type. If it exists, we find its file,page and record information. Then using these info we perform the deletion operation. Also all the instances of that type is deleted from records$x$.txt files using the information stored in B+tree. Note that when a type is deleted, its B+tree file is also deleted.

- List All Types: Listing the types in the system is rather easy. All we need to do is to find all the entries in the types files and put them in a list. Since the output of this operation is asked to be in ascending order, before writing to the output file we sort the list containing type names and write the results.

4.2 HALO Manipulation Language Operations

- Create a Record: Creating a record is almost the same as creating a type operation with minor differences. Again we first create a records file if there is none in the system at that moment. For a record of a particular type to be created, its type should be in system. Remember we create a B+tree files for a type during its creation. So trying to create a record with a type that does not have its B+tree is considered failure. Also creating a type with the same primary key of another record of the same type is also considered failure. We use the *retrieve* method of our B+tree for that purpose. After this checks are done, we find a spot in a records file and then put the record in there. Note that, if we cannot find an available file in the system, we create a new records file.

- Delete a Record: Deleting a record is similar to the deletion of a type. First we make sure that the type of record we wish to delete exists in the system. To that end, we check if the corresponding B+tree file exists for that type. For a record to be deleted, it must be created with the same primary key before. We check that by using the *retrieve* method of B+tree again. If the record exists, we find its filename, page number and record number from the B+tree and use this information to delete the record. One last note is if the file contains no records after a deletion operation that file is deleted from the system.

- Update a Record: We start with the same checks such as type existence and record existence with the primary key provided. We find its filename, page number and record number from the B+tree and use this information to update the record with the fields given in the input file. We write the result back in the same place.

- Search for a Record: Again firstly we check if the type exists. If so we use its B+tree file to check if the record with the provided primary key exists and return the necessary information by using the location data in B+tree. If primary key does not exist it is considered as a failure.

- List all records of a Type: Check if the type exists. We first find the type of the primary key field. This will be used later on for sorting the records. To find all the records of the same type, we use B+tree. For each record, its location information is found and then necessary fields are read from that location and put in a list. Now we sort the list with respect to primary key type. (In string comparison ”11” $<$ ”3”, observe that this is not true when the type is *int*)

- Filter Records: Check if the type exists. We first find the type of the primary key field. This will be used later on for sorting the records. Then we initialize B+tree using the data in the B+tree. We found a B+tree implementation on the web. In that implementation there was no linked list structure between the leaf nodes. Our solution was to add a attribute to the b+tree class that contains the keys of that tree in ascending order. In that way we accomplished to traverse the leaves with respect to the condition more efficiently. Also we implemented an additional method in the b+tree class. The aim of the method is to return the matching keys that satisfy the given conditions. Note that this matching keys are already sorted due to our approach. Then in the main code we use the data in b+tree to find the necessary fields and write the results to the output file.

5 Conclusion & Assessment

As previously mentioned, the project was to design a database system. To that end, we needed B+Tree implementation. Instead of implementing the B+Tree on our own, we used the code in which you can find from the following link:
`https://gist.github.com/savarin/69acd246302567395f65ad6b97ee503d`
This implementation really helped us to moving on to the project.

When designing the system, we had to make a decision on how to store data records in files. As a result, we have made our decision on behalf of fixed-length records rather than variable-length records, because the implementation of the former seemed rather easy. The fixed lengths are as follows:
Every file has a fixed length of 3 pages, every page has a fixed length of 1 header and 10 records, every header has a fixed length of 89+1 bytes, every record has a fixed length of 12 fields, and every field has a fixed length of 20+1 bytes where +1 byte is reserved for newline.
We believe, we defined page size between 2KB and 3KB as specified in project description by that fixed lengths.

The page headers contain some useful information about the data in that page such as number of records in that page, empty slots (from 1 to 10) in which a record can be placed, and the page number. Storing these information helped us to check the page when deleting or inserting a record without traversing all the records in that page.

All in all, we believe that the design might be more optimal, but yet again, it is very well performing at the moment. Our B+trees currently do not have pointers between sibling leaf nodes. Adding that particular linked-list functionality to trees could be helpful when there is a comparison operation. Apart from B+tree implementation, we could have implemented variable-length records for storage to get rid of the unnecessary blank spaces in the file. In general, the project was helpful to understand the storage and indexing.