Project Horadrim: CMPE321 2022 Spring

Demet Yayla
2019400105

Ecenur Sezer
2018400183

June 3, 2022

Contents

1 Introduction

The project is mainly given to us to experience a throughout database design procedure. We are given some requirements about the system and are asked to equip the system with some definition language and manipulation language operations while deciding on design structures that will yield a fruitfull and easy-to-implement database design. We are to provide a log file to the system, index files the structure is predefined mostly, data files and two system catalogs. I created two different system catalogs, one for searching the existing types and their primary key indexes and one for the attributes. Also a database manager code to be able to create all these and orchestrate a database management system.

2 Assumptions & Constraints

2.1 Assumptions

- All fields entries shall be alphanumeric.
- All type names shall be alphanumeric.
- All field names shall be alphanumeric.
- User always enters alphanumeric characters inside test cases.
- Each type refers to a relation.
- Each field of a type can have at most 20 character length value, including integer inputs.

2.2 Constraints

- Page size is 2 KB.
- File size is 16000 KB.
- A type shall have at most 12 and at least 6 fields.
- Length of a type name shall be at most 20 and at least 12 characters.
- Length of a field name shall be at most 20 and at least 12 characters.

2.3 Error Cases

- Creating an existing type.
- Type of a field entry being other than integer or string.
- A type shall have at most 12 and at least 6 fields.

- Creating a record with an already existing primary key.
- Adding a record of a non-existing type.
- Deleting a non-existing type.
- Deleting a record of a non-existing type.
- Deleting a non-existing record.
- Calling to list a type that doesn't exist.
- If no entry for a type exists in system, listing operation for records give error.
- Listing records of a non-existing type.
- If the given primary key doesn't get along with given primary key in update operation.
- Updating a record of a non-existing type.
- Updating a non-existing record.
- Searching the record of a type that doesn't exist.
- Searching a record that doesn't exist.
- Filtering for a non-existing type.
- Filtering if variable name in condition doesn't exist as primary key or if comparison operators aren't one of the following: ">", "<", "=".
- Updating the primary key when updating a record.

3 Storage Structures

3.1 System Catalog:

The attributes for catalogs are chosen to serve as communication between different elements of database.

System Catalog For Relation Fields: System catalog for relation fields has five attributes: *field_name*, *field_type*, *relation* to which the field belongs, *position* of the field for a given record of the relation. *primary_key* is True if the field is the primary key of the relation it belongs to. An example can be seen below:

field_name	**field_type**	**relation**	**primary_key**	**position**
animal_name	string	cats	True	1
animal_name	string	dogs	True	1
life_expectancy	integer	cats	False	2

Catalog For Each Relation: System catalog for indexes has two fields: *relation* indicating the relation name index belongs, *search_key_field* indicating the field for building the index (in our design, this attribute can be found from reaching the primary key of each relation but I included this as another attribute anyway for ease of access). The info in this table is accessible in system catalog too but it yields info for frequently searched data and it speeds up the process of searching if a type exists. An example can be seen below:

relation_name	**primary_key_index**
cats	1
dogs	3

3.2 Design Spesifications:

- For data entries in leaf nodes in index trees, we use Alternative 2.
- Indexes in system are all unclustured.
- Data files have heap structure.
- No two records of different relations exist in same file.
- A new file exists for each different index and each relation has one index which is based on their primary keys.

3.3 File Design:

- File size is 100 MB.
- File header size is 2048 B (a single page).
- Files are not deleted when they are empty.
- **File Header:** Content of the file header is as below:
  - *file_type* : type of data it stores (relation records, index, system catalog) -
  - *unused_file_size* : size of the file unused
  - *primary_key_field_number* : the number for the field being primary key
  - No new file is created in case of an overflow.

3.4 Page Design:

- Page size is 2948 B.
- Pages are not deleted if empty.

- Page header size is 128 B.
- Each page can have 8 records at most.
- Each file has 51200 pages.
- **Page Header:** Content of the page header is as below:
  - *filled_record_data* : An integer value whose binary representation is a bitwise encoding for the empty and full records. It is smallendian: The first digit implies the first record place, second digit the second place and so on. The digit being one means that place is full and zero means empty.

3.5 Record Design:

- Size is fixed for each relation independent of how many fields a relation has. The record size is arranged for the maximum size it can occupy.
- Size of a record is 240 B.
- Record header size is 0 B – no record header exists.
- Each record can have at most 12 fields and each field has 20 characters capacity at most.
- **Record Header:** Content of the record header is as below:
  - None

3.6 B+ Tree Design:

- Built with primary keys of relations.
- The priorities are determined according to keys' alphanumeric priorities.
- Order is 50 for all trees. I used B+ tree from an external Github repository. It lacked direct deletion method. I do the deletion by calling to sequential private methods of two different classes. First method call returns the leaf node the key is located and written recursively. If the key doesn't exist it returns the node that it should have been located if existed. The other call is done on the node returned from previous method call. It deletes a given key from the node. The repository provides .db files for recording the b plus tree being created and dynamically updates the .db file when it is being updated. It gives error if not closed properly. There are serializer options for key fields, default being Integer. We chose String serializer and modified the external code accordingly. We also changed the default size for key field from 8 to 20 characters.

4 Operations

The operations are parsed and thus directed to the correct if statement. I won't mention the error cases I covered for the operations since they are covered in section 2.3. I utilized mmap library to read and modify exact locations. Also, at the end of each operation independent of its being failure or success, I log the information to "horadrim-Log.csv" file. I use csv library for this.

4.1 Horadrim Definition Language Operations

- **Create Type**

  The equivalent of creating a new type for user is creating a new relation from database point of view. Creating a new relation happens with:

  1. Adding necessary information to system catalogs: "relation_metadata.txt" and "system_catalog.txt".
  2. Creating a b+ tree with its .db file, creating a .rec file for record entries. Apart from executions in main method for this operation, helper function "create_type(type_name, index_of_primary_key)" is called to create .rec file for record entries.

- **Delete Type**

  The equivalent of deleting a type for user is removal of a relation from database point of view. Deleting a relation happens with:

  1. Deletion of index file .db that matches with the relation being deleted.
  2. Deletion of .rec file of the type.
  3. Deleting the corresponding entries from "system_catalog.txt" and "relation_metadata.txt".

- **List All Types**

  1. Read all lines of relation from "relation_metadata.txt", sort them in ascending order and print to "output.txt" file.

4.2 Horadrim Manipulation Language Operations

Create a Record

- Add to corresponding .rec record entry file.
- Add to .db b+ tree index file.
- We call "insert_record(type_name, fields, index_of_key)" method for modifying .rec file and adding record key to b+ tree.

- After putting the record to the first empty place we find in the file (by utilizing mmap), we modify the page header, marking the place we inserted as 1 (not available). We also found the first empty place by parsing each page header for *filled_record_data* field's value.
- We then decrease *unused_file_size* field from file header.
- If the page we are inserting to hasn't been written on before, we create the info for page header first (initiate *filled_record_data*).

Delete a Record

- Delete the record from .rec file by extracting where it is located from b+ tree.
- Delete the corresponding key-value pair from b+ tree.
- From page header, modify *filled_record_data* field's value accordingly.
- Increase value of *unused_file_size* in file header.

Search For a Record

- We get the record location info from b+ tree by providing the given primary key value. Then we parse the value of index key to locate the record. We format the field values and print it to "output.txt".

Update a Record

- We first find the location of the record from corresponding b+ tree, then we format the given updated version and insert to its place.

List All Records of a Type

- We get the location for all record entries from b+ tree because our data record files are unclustered and has holes in it so this way it becomes more efficient. We then format the records and print to "output.txt" but we do this on-the-fly.

Filter Record

- We first validate that the variable name given in condition is actually the corresponding primary key field's name of the type.
- We get all the keys from the b+ tree by calling its .__iter__() method. Then we examine the values for keys that satisfy the conditional statement given and find the record corresponding to this value, format the record and print to "output.txt".
- The ordering of the output records are ascending order according to their primary keys.

5 Conclusion & Assessment

- The repository I found from Github is really successful, it applies all theoretical aspects we learned about b+ trees. It needed some modifications for me to utilize but I learned its code structure while trying to modify it to my use. In it, the author provides different types of serializers, but in our case the key could be both integer and string. I use string serializer for all relations, but if I had more time I would create two instances of the repository code, change their names slightly and create the b+trees with distinct types of serializers by extracting the type of primary key from the given index when creating a type for the first time. To compensate for this, we sort the keys while doing search and filter, with an input already close to be sorted (fully sorted if the primary key is a string type). We could have chosen variable length fields, but it was harder to implement and we didn't have that much time. That would have been more space-efficient.
- We gave more space than required to page and field headers to ease read and writes. They don't take much extra space though.
- There are some unutilized fields on page and file headers but we put them anyways in case it would be useful out of the scope of this project's requirements.
- We don't create new pages for the case of file overflow. We just behave like in NOP command.
- We could have written the code more modular, could have written more extra methods to compile frequently used sequential procedures.