Project 4: Project Horadrim

CmpE 321 Introduction to Database Systems, Spring 2022

Project IV

Cahid Arda Öz - Karahan Sarıtaş
2019400132 - 2018400174

June 5, 2022

Contents

1 Introduction

In Project IV, we are expected to design a database system that supports *Horadrim Definition Language Operations* and *Horadrim Manipulation Language Operations*. Our design is expected to have a **system catalog** for our database, **files**, many **pages** in each **file** and many **records** in each **page**. We have to use **B+ Tree** for the indexing and primary keys of entities will be used as search-keys in our trees. To avoid generating the B+ trees again, we have to store our trees in a file and use them when the Horadrim software is run. Horadrim software should log all operations into a CSV file. We have to log time, operations and status of the operation as success/failure.

Overall, we have to implement a B+ Tree for indexing. Design our system catalog, file structure, page structure, page header, record structure and record header. We have to implement Horadrim Software and write a documentation about our design.

2 Assumptions & Constraints

#	**Assumption**
1	All type and field names are *alphanumeric* as stated in the description.
2	User always enters *alphanumeric* characters inside the test cases.
3	Condition item of the Filter Syntax doesn't contain any space characters within it.
4	Primary key name is always provided first in the Condition item of the Filter Syntax

#	**Constraint**
1	String values consists of at most 20 characters.
2	Integer values can only be positive.

3 Storage Structures

3.1 System Catalogue

System catalogue consists of information about the *types* created as the *Horadrim* grows. For each *type*, *primary key index* and *field information* are stored. For each *field*, name of the *field* and the *data type* are stored.**CatalogHandler.py** is responsible for the management of the system catalogue. Whenever *HoradrimSoftare.py* is run, a **CatalogHandler** instance is created. It opens the file and loads the content to RAM. When the execution is completed, it writes back the updated version of the catalogue to the same file. Our System Catalog is stored like our records in a file. For each type, first he primary key index is provided. Then number of fields is provided. For each field, field name and field type is listed one by one. Fields are indexed from zero to (number of fields - 1). When the horadrimSoftware is run, catalog file is mounted and a dictionary is created using the content of the catalog. Then the actual file is deleted. After the completion of execution, a new file is created with the updated version of the system catalog.

`input.txt`

```
create type angel 3 1 name str alias str affiliation str
create type evil 4 1 name str type str alias str spell str
create record angel Tyrael ArchangelOfJustice HighHeavens
create record angel Itherael ArchangelOfFate HighHeavens
list record angel
list record evil
list type
```

When the horadrimSoftware is run with the input above, content of the system catalog will be as follows (Since the indexing starts from 0 in Computer Science in general, we decrement one from the given primary key index):

```
1angel              primaryKeyIndex     0
1angel              numberOfFields      3
1angel              field0name          name
1angel              field0dtype         str
1angel              field1name          alias
1angel              field1dtype         str
1angel              field2name          affiliation
1angel              field2dtype         str
1evil               primaryKeyIndex     0
1evil               numberOfFields      4
1evil               field0name          name
1evil               field0dtype         str
1evil               field1name          type
1evil               field1dtype         str
1evil               field2name          alias
1evil               field2dtype         str
1evil               field3name          spell
1evil               field3dtype         str
```

3.2 File Design

In our design, files consist of pages. All the pages are the same with one another in terms of structure. There are no special pages such as a header page. Page structure is explained in the next section. Each type associated with a file and file size is determined to be 1 GB. Maximum number of pages in a file is calculated as $Max\_Page = File\_Size//Page\_Size$.

3.3 Page Design

Pages are blocks of content within the files. Each page corresponds to a leaf node in the B+ Tree. Each page consists of two sections: page header and the page body. Page size is set to 2048 Bytes. A diagram of the page structure can be seen below:

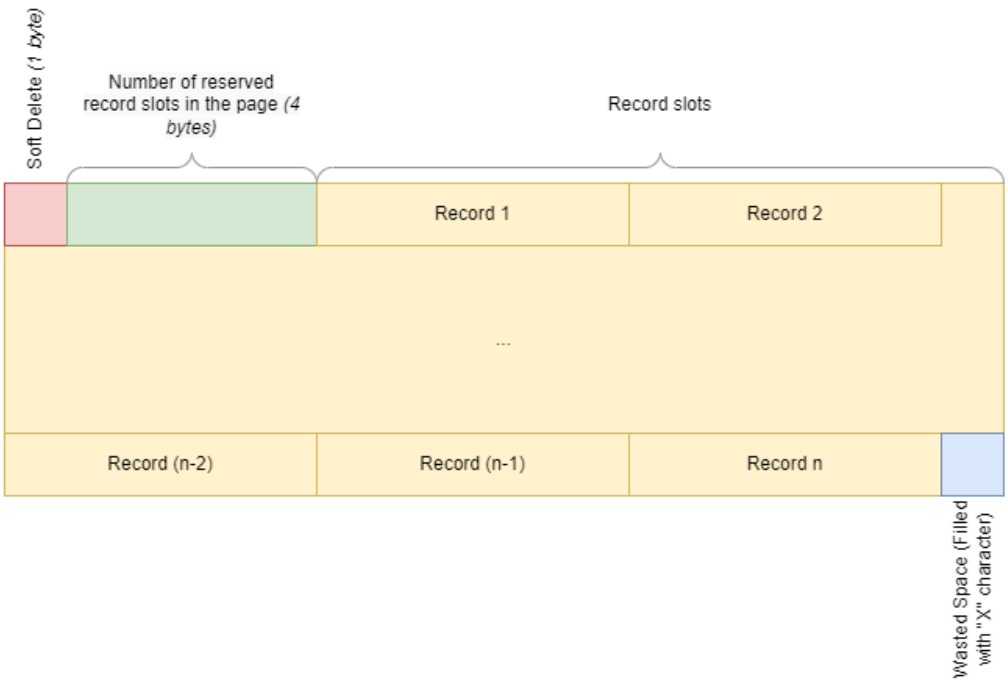

Figure 1: Page Design

First 5 bytes are *The Page Header*. It includes two information: A byte indicating whether

the page is in use and 4 bytes indicating the number of reserved record slots in the page. In our system, each page and record is soft deleted - meaning that they are not really removed from the storage. Rather, they are marked as deleted and corresponding byte is set to 0. If not deleted, it is set to 1.

We refer to the rest of the page as the page body. Page body is where we store our records. These records are not stored in a sorted order in the page. For more information about the records, see subsection 3.4. Number of slots in each page is decided with "$ceil(\frac{PageSize-HeaderSize}{RecordSlotSize})$". Wasted space size is then calculated with "*PageSize-HeaderSize-NumberOfRecordSlotsInPage*RecordSlotSize*". These calculations take place in the **get_page_settings** method of **CatalogHandler**

3.4 Record Design

When a record of a type is created in our database, we store the information in the file that belongs to the provided type. Each record is stored in a fixed-length record slot within the body of a page in the file. Design of the record slot can be seen below:

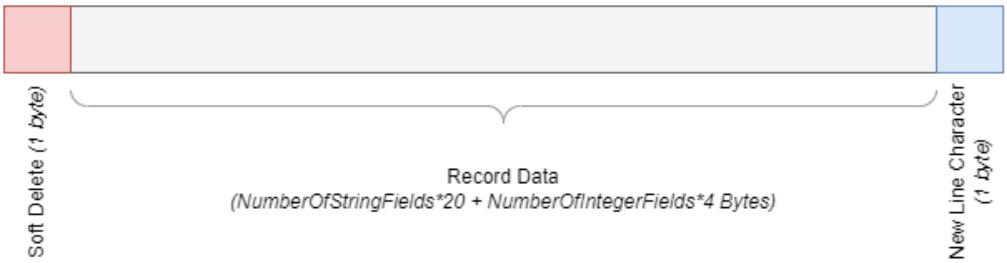

Figure 2: Record Design

These record slots consist of three sections:

- The header which is a single byte indicating whether the record slot is in use. This byte works the same way as the first byte in the page header. Byte is set to 1 or 0 depending on the availability of the slot.
- Record data where we store the information of the record. There are 4 bytes for each integer field and 20 bytes for each string field. First field in the record data is always the primary key.
- A byte which is always set to the newline character. This section was added to improve the readability of the file.

3.5 B^+-Tree design

We implemented our own B^+-Tree from scratch to introduce more flexibility when it comes to the integration of our tree with the Horadrim Software. **bplustree.py** consists of two classes: *Node* and *BPlusTree*. It allows efficient insertion, deletion and search operations. A **bplustree** instance can be created with any **order**. **Order** determines the maximum number of keys a node can have and maximum number of children per node. If $order = M$, then maximum number of keys in a node can be $M - 1$ and maximum number of children per node can be M. Notice that the relationship with *order* and maximum/minimum number of keys allowed is a little bit different compared to the relationship provided in the book. We examined several resources to come up with the most proper definition of *order* and we decided to continue with the equation provided above. But of course, for the end user of the database, it's not important as long as the index works as expected. Our implementation complies with the typical *BPlusTree* properties listed below:

- Each node except root can have a maximum of M children and at least $ceil(\frac{M}{2})$ children.

- Each node can contain a maximum of *M – 1* keys and a minimum of *ceil($\frac{M}{2}$) – 1* keys.
- The root has at least two children and at least one search key.
- While insertion overflow of the node occurs when it contains more than M–1 search key values.

3.5.1 Search

Algorithm for the search operation:

1. Start from the root node. Compare the given key with the keys at the root.
2. If $k < k_1$, go to the very left child of the root node.
3. Else if $k \geq k_1$, compare it with k_2. If $k < k_2$, then go to the second child of the root from left.
4. If $k > k_2$, follow the same procedure for other keys. If given key is greater than all keys in the node, go to the very last child of the root.
5. Repeat the above steps until a leaf node is reached.
6. If k exists in the leaf node, return page index and *True* so that we can log it as success, otherwise return *False*.

3.5.2 Insertion

Algorithm for the insertion operation:

1. Until the current node is a leaf node, iterate through the nodes according to the primary key provided.
2. Insert the key into the leaf node in increasing order. Binary search is utilized in order to find the proper position although its advantage over linear search cannot be observed properly with small input sizes. If there is an overflow follow the following steps:
   - Split the node into two nodes and create an artificial parent.
   - Left node gets the $M/2$ number of keys. Right node gets the remaining keys. This ensures that right node has always equal or one more key than left node.
   - Copy the smallest search key value from second node to the parent node.
   - Insert the key of the artificial parent node to the real parent of the node.

   During the backtracking, an overflow may occur in a non-leaf node. If so, follow exactly the same steps above. Stop the execution if overflow doesn't occur in the parent node or if parent is *None*, meaning that we splitted the root node. If we splitted the root node, it means that new root is the artificial parent node we created. Assign the root of the tree accordingly.

3.5.3 Deletion

Algorithm for the deletion operation:

1. To delete a key, first iterate through the nodes following the same algorithm provided for *Search*.
2. After finding the leaf node N, remove the key from it.
3. If N meets the "minimum number of keys" criteria, then deletion operation is completed. Proceed to the last step.
4. If N doesn't meet the "minimum number of keys" criteria, then look for the adjacent siblings, that is, left and right adjacent nodes.

5. If one of them can spare a key without falling under the limit, then exchange a key and change the key of the parent node accordingly. If it is left sibling, remove the last key and prepend it to the keys of the current node. If it is right sibling, remove the first key and append it to the current node.

6. If none of the siblings is available to spare a key, then we have to apply *merging* operation. If possible, select the right sibling. If right sibling doesn't exist, go with the left sibling. Merge right node to the left node (either merge right node to current node N or merge current node N to left node) and remove the right node.

7. After *merging* is completed, backtrack recursively. We have deleted right node, therefore we have to remove corresponding key from the parent. Check if it causes parent node to fall under the "minimum number of keys" limit. If not, proceed to last step.

8. If parent node doesn't meet the "minimum number of keys" criteria, then follow the same steps starting from 4. step recursively.

9. Root node doesn't have to be complying with the "minimum number of keys" rule. If it gets deleted completely, this results in the decrease of height.

10. As a last step, check if any internal node contains the removed key in it. If so, swap it with the minimum key of the right subtree. By doing so, we always guarantee that keys in the internal nodes exist in the leaf nodes as well.

Our *BPlusTree* implementation exhibits the same behavior visualized here.

4 Operations

In this section, we will provide an overview of the definition and manipulation language operations of the Horadrim Software. First two steps of all operations are the same. In order to not write these common steps for each operation, I will provide them down below:

1. Each line of the input is read and the tokens in the line are extracted with the iterator created with the **line_iterator** method of **InputLoader** (see **InputLoader.py**).

2. Special tokens such as the operation name, primary key etc. are then identified according to the Horadrim Syntax from the list of tokens generated by the iterator. This step takes place in the **horadrimSoftware.py** after the tokens are extracted from the lines.

In the next sections, we will provide an overview for each operation after the two steps explained above. Additionally, **CatalogHandler** is defined in the **CatalogHandler.py** file, the **FileHandler** is defined in the **FileHandler.py** file and **OutputWriter** is defined in the **OutputWriter.py** file.

4.1 Horadrim Definition Language Operations

4.1.1 Create

type_exists method of **CatalogHandler** is called. If the type already exists, execution of the line fails.

If the type doesn't exist in the system, then the field information of the new type is processed and the **add_type** method of **CatalogHandler** is called. This method does the following:

- Add the type information to the catalog in the memory
- Create a B+ Tree for the type and save it in the memory
- Create a file on the disk for the type and allocate the first page. This step is handled with the **mount_file** and **add_page** methods of **FileHandler**.

Once these operations are completed, success is logged to the log file.

4.1.2 Delete

type_exists method of **CatalogHandler** is called. If the type doesn't exist, execution of the line fails.

IF the type exists, **remove_type** method of **CatalogHandler** is called. This method does the following:

- JSON file that contains the B+ Tree of the type is removed from the disk.
- **FileHandler** is called to mount the file belonging to the type and remove it.
- Information about the type is removed from the catalog in the memory.

Once these steps are completed, success is logged to the log file.

4.1.3 List

get_types method of **CatalogHandler** is called. The list of types is sent to the **write_types** method of **OutputWriter**, which prints the list of types to the output file. Then success is logged to the logfile.

4.2 Horadrim Manipulation Language Operations

In all of the manipulation language operations, we first check whether the type exists in our catalog with the **type_exists** method of **CatalogHandler**. If the type doesn't exist, then we log that the execution of the line failed.

If the type exists, then we get the B+ tree of the type from memory using the **get_tree** method of **CatalogHandler**. We also make the **FileHandler** instance mount the file of the type using the **mount_file** method.

4.2.1 Create

In the create record operation, we pass the list of fields in the line to the **fix_fields_for_record** method of **CatalogHandler**. This method simply moves the primary key to the first index in the list of fields. This is necessary since we always store the primary key first in our records. Then the **insert** method of **BPlusTree** instance (which belongs to the type) is called to insert the primary key to the tree.

- Fail Case: If the tree already contains the primary key provided, then the **insert** doesn't change anything in the system and simply returns False. We log that the execution of the line has failed.
- Success Case: If the primary key doesn't exist in the database, then the primary key is inserted to the B+ Tree of the type. Then, within the **insert** method, the **write_record** method of **FileHandler** is called to write the record to the file in the disk. **insert** returns True and success result is logged to the log file.

4.2.2 Delete

In the delete record operation, we call the **delete** method of **BPlusTree**. This method searches the tree to find the primary key provided.

- Fail Case: If the primary key is not found in the tree, **delete** returns false and the fail is logged to the log file.
- Success Case: If the primary key is found in the tree, primary key is removed from the tree and the file page that corresponds to the page index associated with the leaf node where the primary key was found. When a primary key is removed from the tree together with the record that corresponds to it, there is a change that other records are affected. This happens because the leaf node in question (call it node A) sometimes violate the minimum key count constraint (see subsection 3.5). When this constraint is violated two options are possible:

- A key from neighboring leaf nodes may be transferred to the node A. In this case, **delete_record** and **write_record** methods of **FileHandler** are called to remove the transferred record from the page corresponding to the neighboring node and add the transferred record to the page corresponding to node A.
- Neighboring nodes can not lend a key to node A. A merge is required. The merge is applied in the tree and then the **merge_pages** method of **FileHandler** is called. This method soft-deletes one page, transfers the records within that page to another page.

Merge operations are carried out until the B+ Tree satisfies the B+ tree constraints. When the operations are finished, success is logged to the console.

4.2.3 Update

In the update record method, we first call the **fix_fields_for_record** method of **CatalogHandler** to fix the list of fields (see subsubsection 4.2.1). Then the **search** method of **BPlusTree** is called to get the page index of the record being updated.

- Fail Case: If the record doesn't exist in the B+ Tree, then the page index variable is None. We log that the execution of the line has failed.
- Success Case: If the page index variable is not None, then we have the page index of the page. **delete_record** and **write_record** methods of **FileHandler** are called consecutively to first delete the old record from the page and then to write the new record. Then we log success.

4.2.4 Search

In the search operation, we provide the primary key and the type name to the **search** method we defined in the beginning of the **horadrimSoftware.py**. We have written this method because there are two different syntax for the search operation. Other syntax is when filter operation is called with the "=" condition (see subsubsection 4.2.6).

search method gets the **BPlusTree** instance of the type. **search** method of **BPlusTree** is called to get the page index of the record whose primary key matches the provided primary key.

- Fail Case: If the page index is None, then the record is not found. We log that the execution of the line failed.
- Success Case: If the page index is not None, then it is the page index of the page where the record is stored. Fields in the record are read with the **read_record** method of **FileHandler**. Then the list of fields is fixed for the output with the **fix_fields_for_output** method of **CatalogHandler**. This method does the opposite of **fix_fields_for_record** (see subsubsection 4.2.1). Fixed fields are written to the output file with the **write_fields** method of **OutputWriter**. Success is logged to the log file.

4.2.5 List

In the list operation, we first use the **leaf_nodes** method of **BPlusTree** which returns a list of page indexes that correspond to the leaf nodes in the B+ Tree of the type.

- Fail Case: If the list of page indexes is empty, then there are not items to list in the type. We log fail.
- Success Case: If the list of page indexes is not empty, then we have pages with active records. For each page index, we call the **read_page** method of **FileHandler** which returns the information listed below:
  - Whether the page is active are soft deleted.
  - Number of records in the page
  - A list denoting whether the record slots in the response are active. All true if the **read_page** is called with ignore_deleted_records=False.

– List of record data

List of record data is sorted, then each record data is fixed with **fix_fields_for_output** method (see subsubsection 4.2.4) of **CatalogHandler** and written to the output file with **write_fields** method of **OutputWriter**. Success is logged to the log file.

4.2.6 Filter

First, the condition token in the input is split to three pieces: Primary key name, condition and the value. If the condition is "=", then we simply call the **search** method defined in **horadrimSoftware.py** (See subsubsection 4.2.4)

If the condition is "<" or ">" then we do the exact same thing done in the list operation (see subsubsection 4.2.5) except for one difference. Instead of fixing and writing every record returned in **read_page**, we check whether the primary key of the record satisfies the condition first. We fix and write if and only if the condition is satisfied.

5 Conclusion & Assessment

For the deletion of our records and pages in files, we preferred implementing soft delete procedure. It made our work easier since we didn't have to reorganize the files after each delete operation. By the usage of soft delete mechanism we avoid spending time on restoring the pages/records. However it also requires us to iterate through our pages/records and check the corresponding byte to determine if it's available or not. Since we can't guarantee that our pages/records will be in ascending order, we use a simple linear search to find the next available page or record. Linear search might take some time as the input size goes to infinity. However, this effect is not quite noticeable for the small input sizes.
One of the advantages of our project design is that majority of the design choices such as *page size*, *maximum number of fields*, *maximum name length of a type* etc. are not "hard-coded", meaning that one can run the *HoradrimSoftware.py* by changing all of these values from settings.py. Such flexibility made our work easier while we were testing our code with small input sizes.
Rather than giving a fixed record size (maximum record size that one can have), we calculate the size of the records using the data types of the fields. This way, we avoid allocating unnecessary amount of data to the records. Corresponding *B+ Tree* is created by calculating the order from page size and record size dynamically. Since pages of different types are separate, it doesn't lead to any problem during the database operations.
We haven't stored the id of the records in the leaf nodes, rather we stored only the id of the page. This means that after finding the page with the id we have to scan the records to find the one we are looking for based on the search key. This is one of the downs of our design since it increases the I/O operations needed to fetch the records.