CMPE 321 - 2022 Spring
Project 4: Project Horadrim

Ömer Özdemir- Burak Ferit Aktan
2018400372-2019400183

May 28, 2022

Contents

1 Introduction **3**

2 Assumptions & Constraints **3**
2.1 Assumptions 3
2.2 Constraints 3

3 Storage Structures **4**

4 Operations **5**
4.1 Horadrim Definition Language Operations 5
4.1.1 How create type works 5
4.1.2 How delete type works 5
4.1.3 How list type works 6
4.2 Horadrim Manipulation Language Operations 6
4.2.1 How create record type works 6
4.2.2 How delete record type works 7
4.2.3 How update record works 7
4.2.4 How search record works 8
4.2.5 How list record works 8
4.2.6 How filter record works 9

5 Conclusion & Assessment **10**

6 Comments **10**

7 Parameters of B+ Tree **12**

8 Functions of B+ Tree **12**

1 Introduction

In this project, we are wanted to create and manage database system called Horadrim. The project is implemented in python since it provides lots of open-source libraries which makes it easier to develop our project. Since B+ Tree makes it faster to delete/insert/search by giving up from memory 10 percent of the available data. B+ Tree is stored in a file since Horardim should be history sensitive. In other words, when database system is shut off, current B+ Tree will written into the corresponding file. For each table exist in the Horadrim, we create a B+ Tree for that specific table. So, there will always N number of B+ Tree file, where N is the number of table exist in the database. In system catalog, we store the current schema of the database. Horadrim Definition Language Operations helps us to create tables , delete existing ones and listing all tables. Horadrim Manipulation Language Operations helps us to make change on rows of a specific table.

2 Assumptions & Constraints

2.1 Assumptions

- Page size size
- Info stored in page header
- Info stored in record header
- Number of fields a type can have is greater than or equal to 6.
- Length of a type name is greater than or equal to 12.
- Length of a field name is greater than or equal to 12.
- All fields are alphanumeric. Also, type and field names are alphanumeric
- User always enters alphanumeric characters inside the test cases
- Physical storage controller of Horadrim is handled by Operating System.

2.2 Constraints

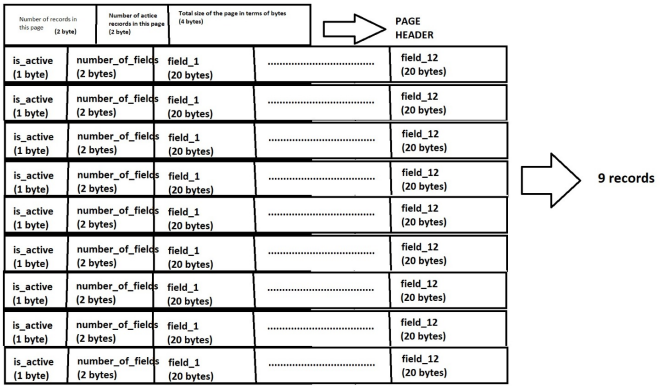
1. Primary key is used for indexing B+ Tree
2. Each type (table) is represented by its own B+ Tree indexing file.
3. Create and delete operations changes the structure of B+ Tree, thus its B+ Tree file.
4. B+ Tree is created for each type for once and used again and again.

5. The data (row records of a table) is stored in a file and each file includes 2 or more pages.

6. Whenever an operation is done, the corresponding file is opened. Then, necessary page is chosen and brought to the memory. Then, after the operation is done, we write it back to the corresponding file and change the its corresponding B+ Tree if necessary.

7. All pages are distributed among multiples pages. So, data files are created and deleted as necessary.

3 Storage Structures



Records are stored in files.

Each file has 3 pages.

Files, records and pages are fixed size. Sizes are determined according to maximum possible input/data lengths. Unused bytes are filled with dollar sign.

Each page consists of a page header and 9 records.

Each page has a header of 8 bytes. First 2 bytes are number of records in that page, following 2 bytes are number of active records in that page, following 4 bytes are size of the page in terms of bytes.

Each record consists of a record header and a record body.

Record header is 3 bytes. First one is valid bit (will be used to logical delete), following two bits are number of fields in the record.

Record body consists of 12 fields, each field 20 bytes. In each field, we write data of a field of a record. If number of fields are less than 12, we fill remaining fields with dollar sign. If length of the field is less than 20 bytes, we fill remaining bytes with dollar signs.

length of a type name = 20

length of a field name = 20

number of fields a type can have = 12

record size = 3 bytes record header + 12 fields * 20 bytes per field = 243 bytes

page size = (3 pages)*((8 bytes page header) + (9 records per page) * (243 byte record size) = 2195 bytes

file size = (3 pages per file) * (2195 bytes page size) = 6585 bytes

Assume name of a table is "computer". The first file storing data of this table is named as "computer_1.txt". When it is filled up we add new files like "computer_2.txt".

Delete operations are done as logical delete. When all records in a file are deleted, file is also deletes.

System catalog has a very similar structure to the main storage structure. The main difference is system catalog is just one file. If it's filled up new pages are added to the end of it.

Pages of the system catalog are consists of 8 bytes header (same structure), each page has 8 records. Records are consist of 23 bytes of header (1 byte active_bit, 20 bytes table name, 2 bytes primary key index) and a record body of 12 fields, each are 23 bytes (20 bytes for field names, 3 bytes for field types - like str,int)

4 Operations

4.1 Horadrim Definition Language Operations

Operation	Input Format	Output Format
Create	create type <type-name><number-of-fields><primary-key-order><field1-name><field1-type><field2-name>...	None
Delete	delete type <type-name>	None
List	list type	<type1-name> <type2-name> ...

4.1.1 How create type works

Step 1: System catalog is updated for the given type-name with field-names and primary-key by adding it.

Step 2: First file is created for the data record of that type.

4.1.2 How delete type works

Step 1: All files where records are store is deleted.

Step 2: System catalog is updated for the given type-name by deleting that type.

4.1.3 How list type works

Step 1: Open the system catalog

Step 2: Do a search on system catalog for type-names.

Step 3: Close the system catalog.

4.2 Horadrim Manipulation Language Operations

Operation	Input Format	Output Format
Create	create record <type-name><field1-value><field2-value>...	None
Delete	delete record <type-name><primary-key>	None
Update	update record <type-name><primary-key><field1-value><field2-value>...	None
Search	search record <type-name><primary-key>	<field1-value><field2-value>...
List	list record <type-name>	<record1-field1-value><record1-field2-value>... <record2-field1-value><record2-field2-value>... ...
Filter	filter record <type-name><condition>	<record1-field1-value><record1-field2-value>... <record2-field1-value><record2-field2-value>... ...

4.2.1 How create record type works

Step 1: Open the available data record file

Step 2: Read the necessary page

Step 3: Delete that page from the file

Step 4: Insert that new record to that page (in memory)

Step 5: Write that new page starting from deleted page's beginning

Step 6: Close the record file

Step 7: Open the B+ Tree index file

Step 8: Insert record related things to the B+ Tree

Step 9: Close the B+ Tree index file

4.2.2 How delete record type works

Step 1: Open the data record file

Step 2: Read the necessary page

Step 3: Delete the necessary page (in file)

Step 4: Delete the record from that page (in memory)

Step 5: Write the new page to data record file, starting from deleted page's beginning

Step 6: Close the record file

Step 7: Open the B+ Tree index file

Step 8: Read the B+ Tree index file

Step 10: Mark that record in B+ Tree as deleted

Step 11: Overwrite that new B+ Tree to B+ Tree index file

Step 12: Close the B+ Tree index file

4.2.3 How update record works

Step 1: Open the B+ tree index file

Step 2: Read the B+ tree index file

Step 3: Get the path of data record file from B+ Tree

Step 4: Open the data record file

Step 5: Read the necessary page

Step 6: Delete the necessary page (in file)

Step 7: Find that new record in the page (in memory)

Step 8: Update that record in the page (in memory)

Step 9: Write that page to data record file starting from the same line where deletion happened

Step 10: Close B+ Tree index file

Step 11: Close data record file

4.2.4 How search record works

Step 1: Open the B+ tree index file

Step 2: Read the B+ tree index file

Step 3: Get the path of data record file from B+ Tree

Step 4: Open the data record file

Step 5: Read the necessary page

Step 6 : Find that new record in the page (in memory) and print it.

Step 7 : Close the B+ Tree index file.

Step 8: Close data record file

4.2.5 How list record works

Step 1: Open the B+ Tree index file

Step 2: Read the B+ Tree index file

Step 3: If search is not done, go to Step 4. Otherwise, Go to Step 9.

Step 4: Search over the keys of B+ Tree and for each key, Go to Step 5

Step 5: Open the data record file and Go to Step 6

Step 6: Read the necessary page and Go to Step 7

Step 7: Find the record in page and print it. Then, Go to Step 8

Step 8: Close the data record file. Then, Go to Step 3

Step 9: Close B+ Tree index file.

4.2.6 How filter record works

Step 1: Open the B+ Tree index file

Step 2: Read the B+ Tree index file

Step 3: If search is not done, go to Step 4. Otherwise, Go to Step 9.

Step 4: Search over the keys of B+ Tree and check condition. If condition holds, Go to Step 5

Step 5: Open the data record file and Go to Step 6

Step 6: Read the necessary page and Go to Step 7

Step 7: Find the record in page and print it. Then, Go to Step 8

Step 8: Close the data record file. Then, Go to Step 3

Step 9: Close B+ Tree index file.

5 Conclusion & Assessment

Using B+ Tree for easy access to the necessary data is nice. By increasing the memory complexity 10 percent of the available data for the corresponding B+ Tree, instead of having linear time complexity for operations, we decrease the time complexity to log(N) where N is the number of data available in the tree. Using another person's code leave us no optimization or improvement on B+ Tree, that was something we could have improved. Since our B+ Tree does not support removing, our B+ Tree could have been improved in terms of memory complexity. However, since allocation and de-allocation requires a system call and also deleting a node requires to balance the B+ Tree, we gain from the time complexity a lot. However, using someone's B+ Tree helped us to develop Horadrim in a short amount of time. Also, using python to develop Horadrim decreased our development time, thanks its libraries and easy debugging features. If we had known the data coming to us in advance, we could have chosen something to be index instead of primary key . This would hugely improved the time complexity thanks to clustered design.

6 Comments

Why do we need indexing?
In todays database systems, instead of create,update,delete kind of instructions, select queries are used a lot and those queries most commonly includes range of record based on some field like primary key. Therefore, since hashing makes searching O(1), using hashing was a nice idea at a first glance. However, if search is done based on some range (< and >) like primary-key instead instead of equality condition (=), then we would need to go over all the keys in hash. So, if we know that queries are most commonly based on some range based on some field like primary key, then using B+ Tree is better than hashing (since when we reach a leaf node with equality search, then, after that, we can go to the previous and next leaf nodes with the help of pointers at one step instead of starting from the search from the root again) . Since in Horadrim, we use primary keys to run queries and not equality (=) but also > and < filters are used, therefore, using B+ Tree for indexing makes sense as stated in the description. Since each type (a.k.a table) have its own primary key, we should use different B+ Trees for each. So, the number of B+ Trees should be equal to the number of types (a.k.a tables). Also, it is required to whenever a record is updated (like deletion or insertion), we should update the B+ Tree since database records are changed.

What is the structure of B+ Tree and why do we prefer it over hashing?
Each node in the B+ Tree corresponds to a page (chunk of data like 4096byte). So, reach to any non-leaf or leaf node means 1 Disk I/O. Each Non-leaf nodes are used to reach to the leaf node (they just give us the direction which path

should we choose on the tree to reach to the leaf). Each Leaf node stores (search-key,pointer to the orginal record stored on secondary storage) pair where search-key help us to identify a specific data. For example, we can use Primary-key for search-key since it uniquely identifies exactly one node and thus, we can sort the tree based on that search-key. Pointer to the original record stored on secondary storage is the place where we should look at the secondary storage for the data records. So, by sorting the tree based on that field, we improve the best case time complexity and average time complexity over hashing. This is because, first we will reach to some leaf node, then going left will mean decreasing that search-key value and going right will mean increasing that search-key value. Hashing was not able to provide this feature because hashing, in case of search-key collision, it will use some probing like linear,quadratic etc, which means that we cannot for sure know the exact location of a search-key, which means hashing cannot sort the data. After reach to the record file by the help of pointer, the file might be sorted (a heap file) or sorted file. In case of heap file, we would need to scan the entire file and check whether search-index matches. However, in the case sorted file, we would know for sure where that record is.

What are the alternatives for data entries in B+ Tree?
While using B+ Tree, we can choose the store the original record in B+ Tree, which is called alternative 1. While using B+ Tree, we can choose to store a pointer to in which file, in which page, in which line record lies, which is called alternative 2 if that pointer is only one, called alternative 3 if that pointer is more than one.

What is the purpose of record header?
In data file, we have record header and record data. Record data gives information to identify record data efficiently. The information it will give to us is determined by us. For example, record header tells the type of the record data, whether the record data is valid or not (logical deletion), number of fields it have etc.

Why we should read record files page by page?
Some number of —record header—record data— is called a page. The some number is up to us. We can choose it as 8, for example. The less the number we choose, the less memory it takes to load that page into the memory. The less the number we choose, the more I/O operation will needed in total. So, there is a trade-off. If memory is not enough to load that whole file into memory then the database will not. Fortunately, if we choose to load files page by page and where pages are really small like 4kb or 8kb, then all todays computers would be able to run the database. Therefore, loading the record files page by page makes more sense than loading it all at once or something like that. Also, each page will have its own page-header which gives the information that we choose it to give. For example, it might tell us the number of —record header—record data— pairs in the page, what kind of record in the page, first x bytes are record

header the rest is record data,

Why don't we need to store record files in order if we are using B+ Tree? In horardim, we did not need to store record data in any order like ascending or descending because B+ Tree will store the data sorted anyways.

7 Parameters of B+ Tree

order –> The number of indexes in a node. For example, if it is 50, then one node will have 50 indexes in it

page_size –> The size of a node. For example, if Order=50 and each index occupies 20byte, than 2000byte would the maximum allowed which is lower than page_size, which is allowed. If page_size were lower than 2000, then we get an error since space is not enough to hold those indexes.

value_size –> Size of the file pointer(value in leaf) in bytes

key_size –> Size of the index (search-key) in bytes

serializer –> determines the serialization technique for the key. By default it is int, so search-key will be int. If one wants to use string as index for B+ Tree, then serializer=StrSeriliazer() should be given as parameter.

cach_size is like buffer management, we put the most frequently pages onto the main memory. By default its size is 64 pages.

8 Functions of B+ Tree

tree.get(Key) –> Gives the value of the leaf node for the given Key

tree[Key] –> Gives the value of the leaf node for the given Key

tree.insert(Key,Value) –> The given Key,Value pair is inserted into the B+ Tree where value is the file pointer and key is the search-index.

When we open the tree, a wad file is created which is for allowing parallel execution of the B+ Tree. However, since in our pythom program we do not use any external library to utilize our multicores in our computers, we were not able to benefit it. So, it basically tries to minimize data inconsistency due to parallel execution.

Serializer.py, first convert the key into bytes. Then, encodes the key in utf-8 format and decodes it when necessary in utf-8 format again. Only integer and

string keys are allowed.

const.py is just used for constant values.

__init__.py is used to mark bplustree files as package

entry.py is used to define structure of entries in a node, deletion of a node, make a modification on it etc.

node.py is used to identify the sturcture of a node (a page), make modification on it, deletion of it etc.

memory.py is used to handle writing to file from memory and reading from the file to memory.