FERHAT ELMAS 2006101102

CMPE 321 PROJECT 1 Designing Simple Storage Manager

Introduction

In this project, I am going to design a simple storage manager for a DBMS. This project will be a simulation of the paging which is result of memory hierarchy. I will give a schema of the general structures. I will try to explain how can be accomplished following basic table and record operations:

Table Operations Record Operations

-Create table -Insert record

-Delete table -Delete record

-List all tables -Search record(s)

-Update record

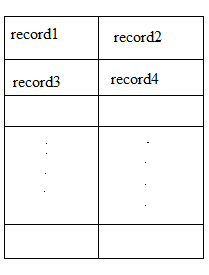
-List all records

I will give some pseudo codes for preceding operations. Finally, I will give advantages and disadvantages of design.

This project is interrelated to second project so, in this design, I made some vital assumptions in order to make implementation easy. Firstly I assumed that a size of a record will be at most 256 bytes because the size of the page is 256 bytes, which is given in the project specification. Each table will have two files that are .dat and .indx extension files. All information about the table structure, its fields and its primary key will be held in system catalog which is also kept in a file that has .sys extension.

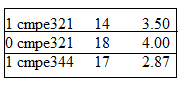
Finally, as limiting record size, I think I will limit max table number and max table field names to certain points to adjust metadata into just one page which finally makes implementation easier.

Page Format



There will not be header for a page. Since a page’s size is known, I can find its id via dividing the address of the file pointer by 256.

Record Format



I will add a simple deletion marker for records, goal of adding deletion marker is whether this record is deleted or not. Deletion marker will be just one byte flag.

In the above figure, we see a record in the second row but actually it was deleted because deletion marker is zero, and I can overwrite it.

Table Format

It is simply a sequence of pages and it won’t hold any special metadata.

Index Format

It is simply a sequence of primary key and page pointer pairs as following:

Ali🡪1

Gizem🡪1

Mehmet🡪3

Zehra🡪13

Ahmet🡪7

System Catalog Format

System catalog will have a header file that holds general information related to whole database such as number of tables, names of tables.

Then, it will hold specific information for each table such as table id, primary key, field names, field sizes, final page pointer, and counter of inspace.

I have some doubt for what I must put into system catalog, so I just give vital ones.

\*\*\*\*\* Before starting to explain operation algorithms, I should say that I tried to simplify implementation, so I tried not to use very complex algorithms. \*\*\*\*\*

Create Table

tablename - # of fields - field names - field sizes - primary key index

student - 3 - id department gpa - 10 20 4 - 1

These are necessary inputs to create a table and this information will be added in system catalog. First part of the input is name of the table which will be created and then number of its fields, fieldnames, fieldsizes and index of primary key must be given respectively.In addition to adding this information to system catalog, a .dat file and a .indx file will be created with the name of table. File that has dat extension will hold records and the other will hold indexing info.

if(exists(table\_name))

begin

print\_error(table\_name)

end

else

begin

page\_id🡨get\_page(system\_catalog)

create(data\_array, user\_inputs)

write\_page(catalog\_file, page\_id, data\_array)

create\_files(table\_name)

end

exists(name) checks whether there is a table with its parameter.

print\_error() prints user-friendly error messsage.

get\_page() returns empty page id which is used to hold metadata of table via accessing system catalog header.

create() prepares page to be written by given user inputs.

write\_page() writes prepared page and catalog\_file is the file handle of system catalog.

create\_files() creates .dat and .indx files of specified table.

Delete Table

The user gives the name of the table. Firstly, related .dat and .indx files are deleted easily by a file handle operation. Secondly table data is deleted from the system catalog header.

if(exists(table\_name))

begin

delete\_files(table\_name)

write\_catalog\_header(table\_name)

end

else

begin

print\_error(table\_name)

end

exists(name) checks whether there is table with given name.

delete\_files() deletes .dat and .indx files of the table.

write\_catalog\_header() deletes name of table from name list and decrements table counter but doesn’t delete table metadata physically, because I can do bookkeeping in header and write on old data, this overall decreases I/O operation.

print\_error() writes user-friendly error message.

List All Tables

I am planning to write two versions for this operation. One will write just table names via just reading system catalog header and the other one will write everything related to table, which is obviously more time-comsuming because of reading specific table metadata such as field names and field lengths.

Version1 Version2

read\_catalog\_header() read\_catalog\_header()

while(system\_catalog!=eof)

begin

read\_page(index)

end

read\_catalog\_header() reads header page and writes names of tables.

read\_page() reads page of related table and writes its meta data.

Insert Record

The user gives the table name for insertion. Then I need to find an empty subblock in a page. I hold a pointer to first page which has space for a record. I read this page. Actually , finding required page is as follows: I use heap file and I have a counter that counts deleted records within records. If this counter is zero, I add to the last page, otherwise by traversing index file I can find partially full inpage. After reading page, I know the size of a record since the sizes of the fields are known via the catalog. Then I search empty record space by skipping record-sized bytes and fill it by user inputs and write page back. Finally, I do some bookkeeping in catalog.

If(exists(table\_name))

begin

repeat(table\_index\_file!=eof)

begin

page\_data🡨get\_page(table\_index\_file).

for i🡨0 to i🡨255

begin

if(page\_data[i]=null)

begin

page\_id🡨get\_page\_id(page\_data[i])

page\_data🡨read\_page(page\_id)

search\_deleted\_record() //same as in the else part

insert(page\_data, user\_inputs)

write\_page(page\_data)

end

i🡨i+indexsize

end

table\_index\_file🡨table\_index\_file+256

end

end

else

begin

page\_data🡨get\_page(table\_data\_file)

for i🡨0 to i🡨255

begin

if(page\_data[i]=0)

begin

insert(page\_data, user\_inputs)

write\_page(page\_data)

end

i🡨i+recordsize

end

end

exists() checks whether there is any null record in index file, this is just counter in catalog.

get\_page(table\_index\_file) returns the 256 byte data (a page) sequantially.

get\_page(table\_data\_file) returns just one 256 byte data via using table catalog info for which page to insert.

page\_data keeps current page data

insert() changes the elements of the page\_data, this function uses the necessary information in system catalog (field types, key etc.)

write\_page() writes the elements of the page\_data.

recordsize and indexsize are calculated using system catalog

table\_data\_file and table\_index\_file are the file pointers of the table.

Delete Record

I search for given primary key in the related .indx file. I know the index sizes so by skipping index-sized bytes, I do search. When I find given primary key, I just convert this index to null. That means I don’t delete record physically, I traverse index file which is very small compared to data file and do manipulations on them, as a result less i/o operation.

begin

page\_data🡨get\_page(table\_index\_file).

for i🡨0 to i🡨255

begin

if(page\_data [i]=primary\_key)

begin

modify\_index()

modify\_del\_marker ()

end

end

table\_index\_file🡨table\_index\_file+256

end

get\_page() returns the 256 byte data (a page) from the table index file.

page\_data keeps current page data.

primary\_key is the key that is to be searched and given by user.

modify\_index() modifies the index that has the given primary\_key such that primary key would be null.

modify\_del\_marker() reads page which is specified by index and in this page finds specified record via checking its primary key and modifies its deletion marker.

table\_index\_file is the index file related table.

List All Records

When the table is specified by user, I will open the corresponding file. Then I will print records sequantially to the end of the file by reading page by page all file.

repeat(table\_data\_file!=eof)

begin

page\_data🡨get\_page(table\_data\_file).

for i🡨0 to i🡨255

begin

if(page\_data[i]=1)

begin

write\_record(page\_data[i])

end

i🡨i+recordsize

end

table\_data\_file🡨table\_data\_file+256

end

get\_page() returns the 256 byte data (a page) from the table file.

page\_data keeps current page data.

write\_record() writes the record which starts at specified point.

recordsize is simply calculated using system catalog

table\_data\_file is the file pointer of the table file.

Search record(s)

repeat(table\_index\_file!=eof)

begin

page\_data🡨get\_page(table\_index\_file).

for i🡨0 to i🡨255

begin

if(check(page\_data[i]))

begin

page\_list🡨page\_list+page\_data[i]

end

i🡨i+indexsize

end

table\_index\_file🡨table\_index\_file+256

end

repeat(page\_list=empty)

begin

page\_data🡨get\_page(table\_data\_file, page\_list)

for i🡨0 to i🡨255

begin

if(check(page\_data[i]))

begin

write\_record(page\_data[i])

end

i🡨i+recordsize

end

end

get\_page() returns a page data.

page\_data keeps current page data.

check() checks whether current record satisfies search condition.

page\_list holds list of pages which must be read form table data file.

write\_record() writes record that starts at specified point.

indexsize and recordsize are sizes of index and record that are calculated by using system catalog.

table\_data\_file is the file pointer of data file.

table\_index\_file is the file pointer of index file.

Get Record

I think there is no need for pseudo code for retrieval because it is one version of searching a record.

Update Record

This operation is very similar to retrieval.The only addition to retrieval is writing page that holds current page to data file.

Conclusion and Evaluation

In the explanation of operations, I tried to give detail as much as I can. I think its performance is very changeable according to database size. I used heap data file and sparse indexing to make implementation easier. Using heap data file structure makes insertion easier because I just need to hold a pointer final page but on the other hand it makes harder retrieval related operations. I designed indexing to fasten get, update, delete and search operations. However, these operations will be fast if database size reaches to a certain point. For preceding operations, I traverse index file and collect page list to open and then start to open pages to get what I want. For performance of all operations, system catalog is very important and especially for record operations, index file is also very important. Therefore, when manager starts, I will load system catalog to main memory to fasten retrieval operations such that I just need to read values of some variables and I will only write page when page is dirty , as a result less i/o. Same manipulations are also required for index file but with one difference, I just load related index file. Because of many operations related indexes, implemetation of indexes requires attention. B tree is usually used and I think I will also use B tree.

I will use a lot of pointers and file operations which are vitally important for implementation of the design. There are some blur points, but I think while implemantation, these points will be more clear.