BOGAZICI UNIVERSITY, ISTANBUL, TURKEY
CmpE321, Spring 2020

Project 2

Implementation of Storage Manager System

Bekir Yıldırım - 2014400054

3 May 2020

Date Performed ..... 3 May, 2020
Instructor ..... Taflan Gündem

Contents

1 Introduction

A storage manager is a program that controls how the memory will be used to save data to increase the efficiency of a system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database. Can be thought as the interface between the DBMS and all the "physically" at the low levels. Storage manager is responsible for retrieving a record, and the other parts of the DBMS are only concerned with the records, not with files, pages, disk and so on. In this project, I am expected to design a storage manager system that supports DDL and DML operations. There should be a system catalogue which stores metadata and multiple data files that store the actual data.

1. DDL Operations
   - Create a type
   - Delete a type
   - List all types
2. DML Operations
   - Create a record
   - Delete a record
   - Search for a record by primary key
   - Update record by primary key
   - List all record for a type

This documents explains my design by showing my assumptions, constraints, storage structures and explaining the algorithms behind the DDL and DML operations in pseudo code. In according to assumptions and constraints, our storage structures and algorithm are arranged. The storage manager will keep according datas in *"typeName.txt"* and *"System-Catalogue.txt"* will be the system catalogue. The records in a file are divided into unit collections(pages) in abstraction.

2 Assumptions & Constraints

Before the design of a storage manager system, one needs to determine the constraints on which the design decisions will be made, since it is not possible make a design without any limits/constraints. Also, we should make some assumptions to construct useful storage structures and algorithm.

2.1 Assumptions

2.1.1 Type

- UTF-8 standard is used in the system, it means a char equal 1 byte.
- A data type can contain 10 fields provided by user exactly. Field values can only be integer as stated in the description.

- More fields are not allowed, yet if it contains less field, the remaining fields will be null.
- I choose the separate file style. Which means there will be a file for each entity type.
- Duplicate names of data types are not allowed.
- Type names shall be alphanumeric.

2.1.2 System Catalogue

- File name of the System Catalogue is *"SystemCatalogue.txt"*.
- System Catalogue file cannot be deleted by any user.
- The system shall not allow to create more than one system catalogue file or delete an existing one.

2.1.3 Page

- Page will be 2048 bytes.
- Page header stores; id of the page, type of records in that page, pointer address to next page in the file, number of record in that page, if that page is empty or not and maximum size of the page(which is 2048 bytes).

2.1.4 File

- Data files have the format *"typeName.txt"*.
- A data file can contain multiple pages.

2.1.5 Record

- Record consist of 10 fields. If it contains less, remaining fields considered as null.
- Record header stores; state of the record(full,empty or deleted) as also integer.

2.1.6 Field

- All of the field values are integers.
- Field names shall be alphanumeric.

2.1.7 Users

- User always enters valid input.

2.1.8 Disk Manager

- A disk manager already exists that is able to fetch the necessary pages when addressed.

2.2 Constraints

2.2.1 Type

- Type names are at most 10 characters long.
- Type names shall be alphanumeric.
- Max length of a type name ($\geq$ 8)

2.2.2 System Catalogue

- The system shall not allow to create more than one system catalogue file or delete an existing one.

2.2.3 Page

- A page can only contain one type of record.
- Pages must contain records.

2.2.4 File

- Max file size can be 65Kb.
- A file has at most 32 pages.
- Should contain reasonable amount of pages
- Not allowed to store all pages in the same file and a file must contain multiple pages.
- Although a file contains multiple pages, it must read page by page when it is needed. Loading the whole file to RAM is not allowed.
- When a file becomes free due to deletions, that file must be deleted.

2.2.5 Record

- Record consist of 10 fields. If it contains less, remaining fields considered as null.
- The primary key of a record should be assumed to be the value of the first field of that record.
- Records in the files should be stored in ascending order according to their primary keys.

2.2.6 Field

- Field names are at most 10 characters long.
- Field names shall be alphanumeric.
- Max number of fields a type can have ($\geq$ 3)
- Max length of a type name ($\geq$ 8)

3 Storage Structures

This design contains two main components which are System Catalogue and Data Files. It contains information about how many bytes are made up and how structure they have. All constraints reserved in *"Constraints.py"*

3.1 System Catalogue

System catalogue is responsible for storing the metadata. It's a blueprint for data types. Any change that can be done in the system via this file. It has multiple pages. Page header(8 bytes) of system catalogue has information about page id and number of records. number of pages has information about how many pages is included in system catalogue and also has information about record with header and their field names.

3.1.1 # of Records in The File(4 bytes)

3.1.2 Page Header(8 bytes)

- Page ID (4 bytes)
- # of Records in The Page (4 bytes)

3.1.3 Record(115 bytes)

- Record Header(16 bytes)
  - Type Name(10 bytes)
  - # of Fields(not null)(4 bytes)
  - Deletion Status (isDeleted) (1 byte)
  - Exit Status (isExist) (1 byte)
- Field Names(10*10=100 bytes)

3.2 Data Files

Data files store current datas. In our designed storage manager system, data files are separated into the number of types. Each data file can store one type of record. Data files have the name *"typeName.txt"*. Each page in data file an store at most 48 records.

3.2.1 # of Records in The File(4 bytes)

3.2.2 Pages(2048 bytes)

Page headers store information about the specific page it belongs to and points to next and previous page.

- Page Header (8 bytes)
  - Page ID (4 bytes)
  - # of Records (4 bytes)
- Records

3.2.3 Records(42 bytes)

- Record Header(2 bytes)
  - isEmpty (1 byte)
  - isDeleted (1 byte)
- Record Fields (10*4= 40 bytes)

4 System Design

4.1 System Catalogue

# of Records						
Page Header						
Page ID			# of Records			
Record Header			Field Names			
Type Name 1	# of Fields 1	isDeleted 1	Field Name 1	Field Name 2	...	Field Name 10
Type Name 2	# of Fields 2	isDeleted 2	Field Name 1	Field Name 2	...	Field Name 10
...	...	...	...	...	...	
Type Name	# of Fields	isDeleted	Field Name 1	Field Name 2	...	Field Name 10

Table 1: Design of a System Catalogue *(Starting with the # of Records)*

4.2 Data File Design & Page Header

# of Records				
Page Header				
Page ID		# of Records		
Record Header		Fields		
isEmpty 1	isDeleted 1	Field 1	...	Field 10
isEmpty 2	isDeleted 2	Field 1	...	Field 10
...	...	...	...	...
isEmpty	isDeleted	Field 1	...	Field 10

Table 2: Design of a Data File *(Starting with the # of Records)*

5 Operations

5.1 DDL Operations

Database Design Language *(DDL)* operations are related to System Catalogue most of the time.

5.1.1 Create a type

Algorithm 1: Creating Data Type

```
1  function createType
2  declare type
3  catalogue ← open("SystemCatalogue.txt")
4  type.name ← User Input
5  type.numberOfFields ← User Input
6  for i ← 0 to type.numberOfField do
7      type.fields[i].name ← User Input
8  if Page.findTypeByName(type.name) != None then
9      return
10 for i ← 0 to 32 do
11     buffer ← getPage(catalogue,i)
12     type_number ← Page.findInteger(buffer, 4)
13     if type_number != 17 then
14         for j ← 0 to type_number do
15             temp ← Page.findType(buffer, j)
16             if temp.name == type.name and temp.isDeleted then
17                 Page.putType(buffer, j, type)
18                 putPage(catalogue, i, buffer)
19                 return
20         for j ← 0 to 17 do
21             Page.putType(buffer, j, type)
22             Page.putInteger(buffer, 4, type_number + 1)
23             self.putPage(catalogue, i, buffer)
24             return
25 catalogue.pageHeader.numberOfRecord++
26 endFunction
```

5.1.2 Delete a type

Algorithm 2: Deleting Data Type

```
function deleteType
catalogue ← open("SystemCatalogue.txt")
name ← User Input
i ← 0
file ← name + i + ".txt"
while File.existFile(file) do
    remove(file)
    i+=1
for i ← 0 to 32 do
    buffer ← getPage(catalogue,i)
    type_number ← Page.findInteger(buffer, 4)
    if type_number == 0 then
        return
    for j ← 0 to type_number do
        temp ← Page.findType(buffer, j)
        if temp.name == name and not temp.isDeleted and temp.isExist then
            temp.isDeleted ← True
            Page.putType(buffer, j, temp)
            putPage(catalogue, i, buffer)
            return
catalogue.pageHeader.numberOfRecord- -
endFunction
```

5.1.3 List all types

Algorithm 3: List All Types

```
function listType
catalogue ← open("SystemCatalogue.txt")
type_list ← [ ]
for i ← 0 to 32 do
    buffer ← getPage(catalogue,i)
    type_number ← Page.findInteger(buffer, 4)
    for j ← 0 to type_number do
        temp ← Page.findType(buffer, j)
        if not temp.isDeleted and temp.isExist then
            type_list.append(temp)
return type_list
endFunction
```

5.2 DML Operations

Database Manipulation Language *(DML)* are generally is related to data files.

5.2.1 Create a record

Algorithm 4: Creating a Record

```
1  function createRecord
2  declare record
3  record.name ← User Input
4  for i ← 0 to record.numberOfField do
5      record.fields[i].values ← User Input
6  if Page.findTypeByName(type.name) == None then
7      return
8  call binarySort(0, record, 0)
9  endFunction
10 function binarySort(pageID, record, fileno)
11 filename ← record.name + fileno
12 if not File.existFile(filename) then
13     createDataFile(filename, record)
14     return
15 buffer ← getPage(filename, pageID)
16 record_number ← Page.findInteger(buffer, 4)
17 current ← binarySearch(buffer, 0, record_number-1, record.fieldValues[0],
     record.name)
18 next ← Record()
19 while current <record_number do
20     next ← Page.findRecord(buffer, current, record.name)
21     Page.putRecord(buffer, current, record)
22     putPage(filename, pageID, buffer)
23     current+=1
24     record ← Page.assign(next)
25 if current == 48 then
26     if pageID == 31 then
27         call binarySort(0, record, fileno+1)
28     else
29         call binarySort(pageID+1, record, fileno)
30         return
31 else
32     putFileHeader(filename, 1)
33     Page.putInteger(buffer, 4, record_number+1)
34     Page.putRecord(buffer, current, record)
35     putPage(filename, pageID, buffer)
```

5.2.2 Delete a record

Algorithm 5: Deleting a Record

```
1  function deleteRecord(name)
2  if Page.findTypeByName(name) == None then
3      return
4  i ← 0
5  filename ← name + i
6  record ← Record()
7  while File.existFile(filename) do
8      for j ← 0 to 32 do
9          buffer ← getPage(filename, j)
10         record_number ← Page.findInteger(buffer, 4)
11         for k ← 0 to record_number do
12             record ← Page.findRecord(buffer, k, name)
13             record_key ← record.fieldValues[0]
14             if record_key == key and not record.isDeleted and record.isExist then
15                 record.isDeleted← True
16                 Page.putRecord(buffer, k, record)
17                 putPage(filename, j, buffer)
18                 putFileHeader(filename, -1)
19                 file_header ← getFileHeader(filename)
20                 number ← Page.findInteger(file_header, 0)
21                 if number == 0 then
22                     remove(filename)
23                     return
24     i+=1
25     filename ← name + i
26 endFunction
```

5.2.3 Search for a record with Primary Key

Algorithm 6: Searching for a Record

```
1  function searchRecord(name, key)
2  if Page.findTypeByName(name) == None then
3      return
4  i ← 0
5  filename ← name + i
6  record ← Record()
7  while File.existFile(filename) do
8      for j ← 0 to 32 do
9          buffer ← getPage(filename, j)
10         record_number ← Page.findInteger(buffer, 4)
11         for k ← 0 to record_number do
12             record ← Page.findRecord(buffer, k, name)
13             record_key ← record.fieldValues[0]
14             if record_key == key and not record.isDeleted and record.isExist then
15                 return record
16     i+=1
17     filename ← name + i
18 endFunction
```

5.2.4 Update for a record with Primary Key

Algorithm 7: Updating for a Record

```
1  function updateRecord(record)
2  if searchRecord(record.name, record.fieldValues[0]) == None then
3      return
4  i ← 0
5  filename ← name + i
6  temp ← Record()
7  while File.existFile(filename) do
8      for j ← 0 to 32 do
9          buffer ← getPage(filename, j)
10         record_number ← Page.findInteger(buffer, 4)
11         for k ← 0 to record_number do
12             temp ← Page.findRecord(buffer, k, name)
13             temp_key ← record.fieldValues[0]
14             if temp_key == key and not temp.isDeleted and temp.isExist then
15                 Page.putRecord(buffer, k, record)
16                 putPage(filename, j, buffer)
17                 return
18     i+=1
19     filename ← name + i
20 endFunction
```

5.2.5 List all records of a type

Algorithm 8: Listing All Records for a Type

```
1  function listRecords
2  if Page.findTypeByName(name) == None then
3      return
4  record_list ← [ ]
5  i ← 0
6  filename ← name + i
7  record ← Record()
8  while File.existFile(filename) do
9      for j ← 0 to 32 do
10         buffer ← getPage(filename, j)
11         record_number ← Page.findInteger(buffer, 4)
12         for k ← 0 to record_number do
13             record ← Page.findRecord(buffer, k, name)
14             if not record.isDeleted and record.isExist then
15                 record_list.append(record)
16     i+=1
17     filename ← name + i
18 return record_list
19 endFunction
```

6 Conclusions & Assessment

In this documentation a storage manager design is proposed where size of each structure is fixed. This creates an inefficiency in terms of memory usage while it makes the storage manager easier to implement. In my design each file can hold at most one record type and can have at most 32 pages. This make accessing a record with its primary key faster but insertion is slower since we have to access a specific page to insert a record. Since we did not do any error checking, if a user enters a wrong input, this storage manager cannot handle it. I decided to use binary sort and search to create record. One down for our design could be the performance if the database is large, program might need to resort so much of opening and closing files and pages, which could be a lot of overhead on CPU.
To sum up, this is really simple storage manager design and it has it owns pros and cons. But mostly, it is very efficient while accessing a record but not so much while insertion. But we can modify this design and improve it. Hence, implementing it would also be easier with necessary modifications that can be realized on the run.