BOGAZICI UNIVERSITY, ISTANBUL, TURKEY
CmpE321, Summer 2019

Project 1

Designing Storage Manager System

Bekir Yıldırım - 2014400054

21 July 2019

Date Performed .... 20 July, 2019
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 ”physcially” 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, data structures and explaining the algorithms behind the DDL and DML operations in pseudo code. The storage manager will keep according datas in typeName.txt and SystemCatalogue.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.

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 1500 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 1500 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.

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.

2.2.4 File

- Max file size can be 140Kb.
- A file has at most 100 pages.

2.2.5 Record

- Record consist of 10 fields. If it contains less, remaining fields considered as null.

2.2.6 Field

- Field names are at most 10 characters long.
- Field names shall be alphanumeric.

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.

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 Pages(4 bytes)

3.1.2 Page Header(8 bytes)

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

3.1.3 Record(115 bytes)

- Record Header(15 bytes)
  - Type Name(10 bytes)
  - # of Fields(not null)(4 bytes)
  - Deletion Status (isDeleted) (1 byte)
- Field Names(10*10=100 bytes)

3.2 Data Files

Data files store cuurent 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 32 records.

3.2.1 Pages(1500 bytes)

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

- Page Header (26 bytes)
  - Page ID (4 bytes)
  - Pointer to Next Page (8 bytes)
  - Pointer to Previous Page (8 bytes)
  - # of Records (4 bytes)
  - isEmpty(1 byte)
  - isFull(1 byte)
- Records

3.2.2 Records(46 bytes)

- Record Header(6 bytes)
  - Record ID (4 bytes)
  - isEmpty (1 byte)
  - isDeleted (1 byte)
- Record Fields (10*4= 40 bytes)

4 System Design

4.1 System Catalogue

# of Pages						
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 Page Header)*

4.2 Page Design & Page Header

Page Header					
Page ID	Pointer to Previous Page	Pointer to Next Page	# of Records	isEmpty	isFull
Record Header			Fields		
Record Id 1	isEmpty 1	isDeleted 1	Field 1	...	Field 10
Record Id 1	isEmpty 2	isDeleted 2	Field 1	...	Field 10
...	...	...	...	...	...
Record Id	isEmpty	isDeleted	Field 1	...	Field 10

Table 2: Design of a Page *(Starting with the Page Header)*

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

```
function createType
declare recordType
catalogue ← open("SystemCatalogue.txt")
recordType.name ← User Input
recordType.numberOfFields ← User Input
/* Now this is a for loop to fill field of record */
for int i=0 to recordType.numberOfField do
    recordType.fields[i].name ← User Input
/* Now this is an if...else conditional loop to fill empty fields of
   record */
if recordType.numberOfField <10 then
    for int i=recordType.numberOfFields+1 to 10 do
        recordType.fields[1].name ← null // Empty fields of record will be
            filled by null.
/* Now push created recordType to "SystemCatalogue.txt" */
catalogue.push(recordType)
catalogue.pageHeader.numberOfRecord++
createFile("recordTypeName.txt")
endFunction
```

5.1.2 Delete a type

Algorithm 2: Deleting Data Type

```
function deleteType
recordType.name ← User Input // Take record will be deleted by user
file ← findFile(recordType.name) // Find record file will be deleted by user
delete file
catalogue ← open("SystemCatalogue.txt")
/* Now this is a foreach to change delete status of record */
foreach page in catalogue do
    foreach record in page do
        if record.typeName == recordType.name then
            record.isDeleted ← 1
endFunction
```

5.1.3 List all types

Algorithm 3: List All Types

```
function listAllTypes
declare Types
catalogue ← open("SystemCatalogue.txt")
/* Now this is a for loop to push all types                    */
foreach page in catalogue do
    foreach record in page do
        if record.isDeleted == 0 then
            types.push(record.typeName)
return types // Return all types
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

```
function createRecord
recordType ← User Input
catalogue ← open(”SystemCatalogue.txt”)
numberOfFields ← file.recordType.numberOfFields)
recordFile ← open(”RecordType.txt”)
boolean lastPage ← 0
boolean newPage ← 0
/* Now this is a foreach to put current page as last page */
foreach page in RecordFile do
    if page.pageHeader.numberOfRecords <32 or page.isFull == 0 then
        lastPage ← page
        lastPage.pageHeader.numberOfRecords++
        lastPage ← 1
    else
        create newPage
        catalogue.numberOfPages++
        newPage ← 1
/* Now this is a foreach to fill records in last page and set empty
   status to 0 */
// If lastPage is not full
if lastPage == 0 then
    foreach record in lastPage do
        if record.isEmpty == 1 then
            for int i=0 to numberOfFields do
                record.fields[i] ← UserInput // Take input for fill fields of
                    record
            record.isEmpty ← 0
// If newPage is created
else if newPage == 0 then
    foreach record in newPage do
        if record.isEmpty == 1 then
            for int i=0 to numberOfFields do
                record.fields[i] ← UserInput // Take input for fill fields of
                    record
            record.isEmpty ← 0
            newPage.pageHeader.numberOfRecords++
endFunction
```

5.2.2 Delete a record

Algorithm 5: Deleting a Record

```
function deleteRecord
recordType ← UserInput
primaryKey ← UserInput // Take input to primaryKey to delete record
file ← open(”RecordType.txt”)
/* Now this is a foreach to delete record with given primary key      */
foreach page in file do
    foreach record in page do
        if record.isDeleted == 0 & record.id == primaryKey then
            page.pageHeader.numberOfRecord—
            record.isDeleted ← 1 // Set record.isDeleted status 1
            record.isEmpty ← 1 // Set record.isEmpty status 1
            if page.pageHeader.numberOfRecord == 0 then
                delete page
endFunction
```

5.2.3 Search for a record with Primary Key

Algorithm 6: Searching for a Record

```
function searchRecord
declare searchedRecord
recordType ← UserInput
primaryKey ← UserInput // Take input to primaryKey to select record
file ← open(”RecordType.txt”)
/* Now this is a foreach to search record with given primary key      */
foreach page in file do
    foreach record in page do
        if record.isDeleted == 0 & record.id == primaryKey then
            searchedRecord ← record // Record is finded with primary key.
return searchedRecord
endFunction
```

5.2.4 Update for a record with Primary Key

Algorithm 7: Updating for a Record

```
function updateRecord
declare updatedRecord
updatedRecord ← UserInput
recordType ← UserInput
primaryKey ← UserInput // Take input to primaryKey to update record
file ← open("RecordType.txt")
/* Now this is a foreach to search record with given primary key      */
foreach page in file do
    foreach record in page do
        if record.isDeleted == 0 & record.id == primaryKey then
            record ← updatedRecord // Record which is finded with primary
                key is updated.
endFunction
```

5.2.5 List all records of a type

Algorithm 8: Listing All Records for a Type

```
function listRecords
declare allRecords
recordType ← UserInput // Take input to list all certain records of a
    type
file ← open("RecordType.txt")
/* Now this is a foreach to search record which is not empty and not
    deleted                                                           */
foreach page in file do
    foreach record in page do
        if record.isDeleted == 0 & record.isEmpty == 0 then
            allRecords.push(record)
return allRecords // Return all records
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 100 pages. This make accesing 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. One down for our design could be the performance if the database is large, but since the project description says not use complex structure just for performance, we did not use more sophisticated structures like B+ trees.
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.