CmpE 321
Introduction to Database Systems

Spring 2020

Homework 1
Storage Management System

Emilcan ARICAN
2016400231

Contents

1 Introduction

In this project I have implemented a storage management system. It basically contains two main parts which are system catalog and data storage units. Purpose of the system catalog is storing the meta data. In the scope of this project meta data is types that can be defined by user. System catalog is structured similarly with the data storage units in this project however it is simpler (ie. pages of the system catalog in a signly linked list manner). The other part is data storage units they hold actual data. There are three structures namely file, page, and record. Files consist of pages, and hold the information of the next file and the previous file in order to create a design like a doubly linked list. Similar approach is present in the pages as well. They keeps id of their next and previous pages, and they contains records. Records have the fields that is provided by the user. (More detailed information will be given in the related sections of the report.) There are several DDL and DML operations that is implemented. These are basically core functionalities of the storage management system. These are create/delete/list type, and create/delete/search/update/list record. Pseudo codes are provided in the related sections.

2 Assumptions and Constraints

2.1 Assumptions

- Page size $\rightarrow$ 1900 bytes
- File size $\rightarrow$ 15200 bytes
- Type header has type id, type state, type name, number of fields, name of the initial page.
- Record header has record id, record state.
- Page header has: (in the system catalog basis) page id, page state, number of types, id of the next page.
- Page header has: (in the data storage unit basis) page id, page state, number of types, id of the previous page, id of the next page.
- File header has file state, number of pages, name of the previous file, name of the next file, initial page id.
- Max number of fields a type can have $\rightarrow$ 8
- Max length of a type name $\rightarrow$ 16
- Max length of a field name $\rightarrow$ 16
- User always enters valid input.
- All fields shall be integers. However, type and field names shall be alphanumeric.
- A disk manager already exists that is able to fetch the necessary pages when addressed.
- I have a "state" enum where 0 corresponds to "available", 1 corresponds to "full", 2 corresponds to "deleted".
- "systemcatalog.meta" file should not be deleted.
- User will not provide two records with the same primary key for a spesific type
- There exist a "swap" function that performs swap operation between two records.
- There exist an unique file name generator function.
- There exist setInitialFile(recordType, fileName) function that sets the related field in the system catalog.

2.2 Constraints

- The data must be organized in pages and pages must contain records. Page and record structure must be explained in the report.
- Storing all pages in the same file is not allowed and a file must contain multiple pages. This means that system must be able to create new files as storage manager grows.Moreover, when a file becomes free due to deletions, that file must be deleted.
- 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.
- 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.

3 Storage Structures

This system has two storage structures system catalog, and data storage units. Data storage units are file, page, and record. File and page are designed as a linked list. In their header they have related fields so that they can refer/access their next and previous neighbours. Files contains pages, and pages contains records. System catalog stores record types and initial files for the specific type.

3.1 System Catalog

System catalog is responsible for storing metadata. It contains record types. General structure of the system catalog as the following: files forms a doubly linked list and contains pages; pages forms a singly linked list and contains types; types has field names and number of fields in them.

3.1.1 File

If a file becomes full, another file is created and added end of the linked list of the files (when a new type will be created). Starting node (initial file) is **systemcatalog.meta**.

File Header

file state	number of pages	name of the next file	name of the previous file	id of the initial page

- file state → 1 byte
- number of pages → 1 byte
- name of the next file → 16 bytes
- name of the previous file → 16 bytes
- id of the initial page → 4 bytes

File Pages

page header 1	type 1	...	type 10
page header 2	type 1	...	type 10
...	...	...	...
page header 8	type 1	...	type 10

- page 1 → 1740 bytes
- page 2 → 1740 bytes

  ⋮
- page 8 → 1740 bytes

3.1.2 Page

Each page contains 10 records. If a page is deleted or becomes full its state is changed accordingly. Id of the starting node (initial page) of the linked list of pages is stored in the header file header.

Page Header

page id	page state	number of types	id of the next page

- page id → 4 bytes
- page state → 1 byte
- number of types → 1 byte
- id of the next page → 4 bytes

Types

Type Header 1	field name 1	...	field name 8
Type Header 2	field name 1	...	field name 8
...	...	...	...
Type Header 10	field name 1	...	field name 8

- type 1 → 173 bytes
- type 2 → 173 bytes

  ⋮
- type 10 → 173 bytes

3.1.3 Type

Each type contains 8 field names. If a type is deleted or becomes full its state is changed accordingly.

Type Header

type id	type state	type name	number of fields	name of the initial file

- type id → 4 bytes
- type state → 4 bytes
- type name → 16 bytes
- number of fields → 1 byte
- name of the initial file → 16 bytes

Field Names

field name 1	field name 2	...	field name 8

- field name 1 → 16 bytes
- field name 2 → 16 bytes

  ⋮
- field name 8 → 16 bytes

3.2 Data Storage Units

Data storage units store the actual data. It contains records provided by user. General structure of the data storage units as the following: files forms a doubly linked list and contains pages; pages also forms a doubly linked list and contains records; records has field values in them.

3.2.1 File

Each file contains only one type of record. If a file becomes full, another file is created and added end of the linked list of the files (when a new record will be created). Name of the starting node (initial file) of the linked list is stored in the header of the related type in the system catalog.

File Header

file state	number of pagess	name of the next file	name of the previous file	id of the initial page

- file state → 1 byte
- number of pages → 1 byte
- name of the next file → 16 bytes
- name of the previous file → 16 bytes
- id of the initial page → 4 bytes

File Pages

page header 1	record 1	...	record 50
page header 2	record 1	...	record 50
...	...	...	...
page header 8	record 1	...	record 50

- page 1 → 1864 bytes
- page 2 → 1864 bytes

  ⋮
- page 8 → 1864 bytes

3.2.2 Page

Each page contains 50 records. If a page is deleted or becomes full its state is changed accordingly. Id of the starting node (initial page) of the linked list of pages is stored in the header file header.

Page Header

page id	page state	number of records	id of the previous page	id of the next page

- page id → 4 bytes
- page state → 1 byte
- number of records → 1 byte
- id of the previous page → 4 bytes
- id of the next page → 4 bytes

Page Records

Record Header 1	field 1	...	field 8
Record Header 2	field 1	...	field 8
...	...	...	...
Record Header 50	field 1	...	field 8

- record 1 → 37 bytes
- record 2 → 37 bytes

  ⋮
- record 50 → 37 bytes

3.2.3 Record

Each record contains 8 fields. If a record is deleted or becomes full its state is changed accordingly.

Record Header

record id	record state

- record id → 4 bytes
- record state → 1 byte

Record Fields

field 1	field 2	...	field 8

- field 1 → 4 bytes
- field 2 → 4 bytes

  ⋮
- field 8 → 4 bytes

4 Operations

4.1 DDL Operations

4.1.1 Create a type

It starts from the "systemcatalog.meta" and finds an available page. (If needed creates a new page.) Iterates over page headers until find an available one. Loads the page and adds the new type. Then updates page header, and file header accordingly.

```
Function createType(typeToCreate: Type):
    fileName ← "systemcatalog.meta"
    fileHeader ← getFileHeader(fileName)
    while fileHeader.fileState = enum.full do
        if fileHeader.nextFileName = null then
            fileName ← generateNewMetaFileName()
            createFile(fileName)
            fileHeader.nextFileName  fileName
        fileHeader ← getFileHeader(fileHeader.nextFileName)
    pageHeader ← getPageHeader(fileHeader.initialPageId)
    while pageHeader.pageState = enum.full do
        pageHeader ← getPageHeader(pageHeader.nextPageId)
    page ← getPage(pageHeader.pageId)
    for $i$ ← 0 to 10 do
        if page.types[$i$].typeState ≠ enum.full then
            type ← page.types[$i$]
            type.typeId ← getNewTypeId()
            type.typeState ← enum.full
            type.typeName ← typeToCreate.typeName for $j$ ← 0 to
             typeToCreate.numberOfFields do
                type.fieldNames[$j$] = typeToCreate.fieldNames[$j$]
            break
    pageHeader.numberOfTypes + = 1
    if pageHeader.numberOfTypes = 10 then
        pageHeader.pageState ← enum.full
        fileHeader.numberOfPages + = 1
        if fileHeader.numberOfPages = 8 then
            fileHeader.fileState ← enum.full
```

4.1.2 Delete a type

Iterates over files, pages, records until find the type with the same id. Then changes its state to deleted. gapFixer is called in oerder to fix the gap appears after deletion (for detailed information please check helper operations). After that updates file header, and page header. I necessary it deletes the file becomes free after deletion.

```
Function deleteType(typeId: int):
    fileName ← "systemcatalog.meta"
    isDeleted ← False
    repeat
        fileHeader ← getFileHeader(fileName)
        pageHeader ← getPageHeader(fileHeader.initialPageId)
        repeat
            if pageHeader.pageState ≠ enum.deleted then
                page ← getPage(pageHeader.pageId)
                for i ← 0 to pageHeader.numberOfRecords do
                    if page.types[i].typeId = typeId then
                        page.types[i].typeState ← enum.deleted
                        isDeleted ← True
                        break
            pageHeader ← getPageHeader(pageHeader.nextPageId)
        until pageHeader.nextPageId ≠ null and isDeleted ≠ True
    until fileHeader.nextFileName ≠ null and isDeleted ≠ True
    pageHeader.numberOfTypes − = 1
    if pageHeader.numberOfTypes = 0 then
        pageHeader.pageState ← enum.deleted
        fileHeader.numberOfPages − = 1
    if fileHeader.numberOfPages = 0 then
        if fileHeader.previousFileName ≠ null then
            previousFileHeader ← getFileHeader(fileHeader.previousFileName)
            previousFileHeader.nextFileName ← fileHeader.nextFileName
            if fileHeader.nextFileName ≠ null then
                nextFileHeader ← getFileHeader(fileHeader.nextFileName)
                nextFileHeader.previousFileName ← fileHeader.previousFileName
            deleteFile(fileName)
```

4.1.3 List all types

Iterates over files, pages, types; of system catalog. While iterating over them prints the name of the types.

```
Function listAllTypes():
    fileName ← "systemcatalog.meta"
    repeat
        fileHeader ← getFileHeader(fileName)
        pageHeader ← getPageHeader(fileHeader.initialPageId)
        repeat
            page ← getPage(pageHeader.id)
            for $i \leftarrow 0$ to $page.types$ do
                if $page.types[i].typeState = enum.full$ then
                    print(page.records[i].typeName)
            pageHeader ← getPageHeader(pageHeader.nextPageId)
        until $pageHeader \neq null$
        fileName ← getFileHeader(fileHeader.nextFileName)
    until $fileName \neq null$
```

4.2 DML Operations

4.2.1 Create a record

First checks if file exists otherwise creates it. Iterates over page headers until find an available one. Loads the page and adds the new record. Then updates page header, and file header accordingly. In order to preserve the sorted state of the file, basicSortFixer is called. (for detailed information please check helper operations)

```
Function createRecord(recordToCreate: Record):
    recordType ← recordToCreate.type
    fileName ← getInitialFileName(recordType)
    if fileName = null then
        fileName ← generateNewFileName(recordType)
        createFile(fileName)
        setInitialFileName(recordType, fileName)
    fileHeader ← getFileHeader(fileName)
    while fileHeader.fileState = enum.full do
        if fileHeader.nextFileName = null then
            fileName ← generateNewFileName(recordType)
            createFile(fileName)
            fileHeader.nextFileName ← fileName
        fileHeader ← getFileHeader(fileHeader.nextFileName)
    pageHeader ← getPageHeader(fileHeader.initialPageId)
    while pageHeader.pageState = enum.full do
        pageHeader ← getPageHeader(pageHeader.nextPageId)
    page ← getPage(pageHeader.pageId)
    for i ← 0 to 49 do
        if page.records[i].recordState ≠ enum.full then
            record ← page.records[i]
            record.recordId ← getNewRecordId()
            record.recordState ← enum.full
            for j ← 0 to 7 do
                if j < getNumberOfFields(recordType) then
                    record.fields[j] ← recordToCreate.fields[j]
                else
                    record.fields[j] ← 0
            break
    pageHeader.numberOfRecords + = 1
    if pageHeader.numberOfRecords = 50 then
        pageHeader.pageState ← enum.full
        fileHeader.numberOfPages + = 1
        if fileHeader.numberOfPages = 8 then
            fileHeader.fileState ← enum.full
    basicSortFixer(pageHeader.pageId)
```

4.2.2 Delete a record

First gets the list of types. Then for each type iterates over files, pages, records until find the record with the same id. Then changes its state to deleted. gapFixer is called in oerder to fix the gap appears after deletion (for detailed information please check helper operations). After that updates file header, and page header. I necessary it deletes the file becomes free after deletion.

```
Function deleteRecord(recordId: int):
    isDeleted ← False
    typeList ← getListOfTypes("systemcatalog.meta")
    for each recordType in typeList do
        if isDeleted = True then
            break
        fileName ← getInitialFileName(recordType)
        repeat
            fileHeader ← getFileHeader(fileName)
            pageHeader ← getPageHeader(fileHeader.initialPageId)
            repeat
                if pageHeader.pageState ≠ enum.deleted then
                    page ← getPage(pageHeader.pageId)
                    for i ← 0 to pageHeader.numberOfRecords do
                        if page.records[i].recordId = recordId then
                            page.records[i].recordState ← enum.deleted
                            isDeleted ← True
                            break
                pageHeader ← getPageHeader(pageHeader.nextPageId)
            until pageHeader.nextPageId = null and isDeleted ≠ True
        until fileHeader.nextFileName = null and isDeleted ≠ True
    pageHeader ← gapFixer(fileName)
    pageHeader.numberOfRecords − = 1
    if pageHeader.numberOfRecords = 0 then
        pageHeader.pageState ← enum.deleted
        fileHeader.numberOfPages − = 1
    if fileHeader.numberOfPages = 0 then
        if fileHeader.previousFileName ≠ null then
            previousFileHeader ← getFileHeader(fileHeader.previousFileName)
            previousFileHeader.nextFileName ← fileHeader.nextFileName
        else
            setInitialFile(recordType, fileHeader.nextFileName)
        if fileHeader.nextFileName ≠ null then
            nextFileHeader ← getFileHeader(fileHeader.nextFileName)
            nextFileHeader.previousFileName ← fileHeader.previousFileName
        deleteFile(fileName)
```

4.2.3 Search for a record (by primary key)

First related initial file name is gathered. Then iterates over files, pages, records in order to find matching first field. Then returns the record.

```
Function searchRecord(primaryKey: int, recordType: Type):
    fileName ← getInitialFileName(recordType)
    repeat
        fileHeader ← getFileHeader(fileName)
        pageHeader ← getPageHeader(fileHeader.initialPageId)
        repeat
            if pageHeader.pageState ≠ enum.deleted then
                page ← getPage(pageHeader.pageId)
                for i ← 0 to pageHeader.numberOfRecords do
                    if page.records[i].fields[0] = primaryKey then
                        return page.records[i]
            pageHeader ← getPageHeader(pageHeader.nextPageId)
        until pageHeader.nextPageId ≠ null
    until fileHeader.nextFileName ≠ null
```

4.2.4 Update a record (by primary key)

First locates the record that will be updated. Then modifies it. Finally calls basicSortFixer (for detailed information please check helper operations) function to maintain sorted state.

```
Function updateRecord(primaryKey: int, newRecord: Record):
    fileName ← getInitialFileName(recordType)
    repeat
        fileHeader ← getFileHeader(fileName)
        pageHeader ← getPageHeader(fileHeader.initialPageId)
        repeat
            if pageHeader.pageState ≠ enum.deleted then
                page ← getPage(pageHeader.pageId)
                for i ← 0 to pageHeader.numberOfRecords do
                    if page.records[i].field[0] = primaryKey then
                        record ← page.records[i]
                        for j ← 0 to 8 do
                            if getNumberOfFields(newRecord.type) > j then
                                record[i].fields[j] ← newRecord.fields[j]
                            else
                                record[i].fields[j] ← 0
            pageHeader ← getPageHeader(pageHeader.nextPageId)
        until pageHeader.nextPageId ≠ null
    until fileHeader.nextFileName ≠ null
    basicSortFixer(fileName)
```

4.2.5 List all records of a type

First related initial file name is gathered. Then iterates over files, pages, records. While iterating prints the records.

```
Function listAllRecords(typeToList: Type):
    fileName ← getInitialFileName(typeToList)
    repeat
        fileHeader ← getFileHeader(fileName)
        pageHeader ← getPageHeader(fileHeader.initialPageId)
        repeat
            page ← getPage(pageHeader.id)
            for $i \leftarrow 0$ to $page.numberOfRecords$ do
                if $page.records[i].recordState = enum.full$ then
                    print(page.records[i])
            pageHeader ← getPageHeader(pageHeader.nextPageId)
        until $pageHeader \neq null$
        fileName ← getFileHeader(fileHeader.nextFileName)
    until $fileName \neq null$
```

4.3 Helper Operations

4.3.1 Gap Fixer

After deleting a record it might create a gap between records. Since it may cause problems about sorting it should be fixed. In order to fix this problem this operation moves the deleted record to the end of the records.

```
Function gapFixer(fileName: String):
    fileHeader ← getFileHeader(fileName)
    pageHeader ← getPageHeader(fileHeader.initialPageId)
    isGapFixed ← False
    repeat
        page ← getPage(pageHeader.pageId)
        for $i$ ← 0 to 48 do
            if page.records[$i$].recordState = enum.deleted
            and page.records[$i+1$].recordState = enum.full then
                swap(page.records[$i$], page.records[$i+1$])
            else
                isGapFixed ← True
                break
        if page.records[49].recordState = enum.deleted then
            tempRecord ← page.records[49].recordState
            pageHeader ← getPageHeader(pageHeader.nextPageId)
            page ← getPage(pageHeader.pageId)
            if page.records[0].recordState = enum.full then
                swap(tempRecord, page.records[0])
                pageHeader ← getPageHeader(pageHeader.previousPageId)
                page ← getPage(pageHeader.pageId)
                swap(tempRecord, page.records[49])
                pageHeader ← getPageHeader(pageHeader.nextPageId)
            else
                break
    until pageHeader = null or isGapFixed = True
    return pageHeader
```

4.3.2 Basic Sort Fixer

Due to update and create record operation, sorted state of the records might violated. This operation created to fix this issue. When an update or create record operation is used, it results in only one misplaced record. This function iterates over records in the file. First goes beginning to end, and if primary key of the next field is lesser than current one switches them. Then goes end to beginning, and makes the opposite of that. By doing so misplaced record finds it place. It works at O(n) complexity where n is number of records at the file. (This trick only works for 1 misplaced record and system prevents more misplaced records emerging)

```
Function basicSortFixer(pageId: int, recordIndex: int):
    pageHeader ← getPageHeader(pageId)
    repeat
        page ← getPage(pageHeader.pageId)
        for $i$ ← 0 to 48 do
            if $page.records[i+1].recordState = enum.full$ then
                if $page.records[i].fields[0] > page.records[i+1].fields[0]$ then
                    swap(page.records[$i$],page.records[$i+1$])
        if $pageHeader.nextPageId \neq null$ then
            tempRecord ← page.records[49]
            pageHeader ← getPageHeader(pageHeader.nextPageId)
            page ← getPage(pageHeader.pageId)
            if $page.records[0].fields[0] < tempRecord.fields[0]$ then
                swap(page.records[0], tempRecord)
                pageHeader ← getPageHeader(pageHeader.previousPageId)
                page ← getPage(pageHeader.pageId)
                swap(page.records[49], tempRecord)
                pageHeader ← getPageHeader(pageHeader.nextPageId)
    until $pageHeader.nextPageId = null$
    repeat
        page ← getPage(pageHeader.pageId)
        for $i$ ← 49 to 1 do
            if $page.records[i].recordState = enum.full$ then
                if $page.records[i].fields[0] < page.records[i-1].fields[0]$ then
                    swap(page.records[$i$],page.records[$i-1$])
        if $pageHeader.previousPageId \neq null$ then
            tempRecord ← page.records[0]
            pageHeader ← getPageHeader(pageHeader.previousPageId)
            page ← getPage(pageHeader.pageId)
            if $tempRecord.fields[0] < page.records[49].fields[0]$ then
                swap(tempRecord, page.records[49])
                pageHeader ← getPageHeader(pageHeader.nextPageId)
                page ← getPage(pageHeader.pageId)
                swap(tempRecord, page.records[0])
                pageHeader ← getPageHeader(pageHeader.previousPageId)
    until $pageHeader.previousPageId = null$
```

5 Conclusions Assessment

Under the conditions of constraints and assumtions stated above, storage management system design is completed. System consists of system catalog and data storage units. In the basis of data storage units, files are forms a doubly linked structure by storing name of the previous and next files. Every type its own linked file structure. Every file manages the pages in them in a similar manner. Pages also creates doubly linked page structure. By holding ids of their previous and next neighbours. Id of initial page is stored in the file header. This structure eases to access certain point in the storage management system that I build. In the basis of system catalog, structure is nearly the same however, pages of system catalog forms signly linked structure. Initial files of the doubly linked file structures is stored in the type headers inside of the system catalog. During a type/record is created if necessary a new file is created as well. During the deletion of a type/record soft delete is performed and state of the type/record is changed. If a file becomes empty after deletion, this file will deleted (except systemcatalog.meta). After deletion of a record there is a possibility that a gap might be appear between records. Therefore a gap fixing operation is used. This approach makes system more reliable and helps to allocate memory more efficiently. After update and create record operations sortedness of the records is violated. So basic sort fixing operation is used in oerder to solve that. It works sequences with only one misplaced record. Since system fixes it after each update and create operation there is no chance to have more than one misplaced record. Pseudo code of the mentioned DDL, DML, and helper operations are provided in the report with basic explanations of them. This design focuses on the allocating least memory posibble, and keeping it sorted and ordered. However in order to perform that helper operations (gap fixer, basic sort fixer) are used very often and it may cause an overhead in the system. Linked list structure I created provides a comfortable maneuvering experience in the system, however in order to achieve this I had to store extra ids and file names. Every link also uses storage. To sum up, design is meets the desired functionalities. It has both powerful and improvable sides.