CmpE 321

Introduction to Database Systems

Spring 2020

Homework 2

Storage Management System Implementation

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. 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 they have numerical assignments in their file names. System derives the relation between them by checking their name. Pages in the files are stored consecutively so they can be accessed by their page indexes. When the index is provided system derives the right access with necessary calculations.(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 → 1700 bytes
- File size → 17000 bytes
- Type header contains type id, type state, type name, number of fields.
- Record header contains record state.
- Page header has: page state, number of types.
- Max number of fields a type can have → 10
- Max length of a type name → 10
- Max length of a field name → 10
- 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.
- User will not provide two records with the same primary key for the same type.
- There exist a "swap" function that performs swap operation between two records.

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. Files contains pages, and pages contains records. System catalog stores record types.

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 contains pages; pages 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). Name format of the meta files are syscatalog.<10-digit-number>.

File Pages

page header 1	type 1	...	type 15
page header 2	type 1	...	type 15
...	...	...	...
page header 10	type 1	...	type 15

- page 1 → 1680 bytes
- page 2 → 1680 bytes

  ⋮
- page 10 → 1680 bytes

3.1.2 Page

Each page contains 15 records. If a page is deleted or becomes full its state is changed accordingly.

Page Header

page id	page state

- page state → 1 byte
- number of types → 1 byte

Types

Type Header 1	field name 1	...	field name 10
Type Header 2	field name 1	...	field name 10
...	...	...	...
Type Header 15	field name 1	...	field name 10

- type 1 → 112 bytes
- type 2 → 112 bytes

  ⋮
- type 15 → 112 bytes

3.1.3 Type

Each type contains 10 field names. If a type becomes full its state changes accordingly.

Type Header

type id	type state	type name	number of fields

- type id → 4 bytes
- type state → 4 bytes
- type name → 10 bytes
- number of fields → 1 byte

Field Names

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

- field name 1 → 10 bytes
- field name 2 → 10 bytes

  ⋮
- field name 10 → 10 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 contains pages; pages 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 (during creating a new record). Form of the data file's name is <type-name>.<10-digit-number>.

Pages

page header 1	record 1	...	record 40
page header 2	record 1	...	record 40
...	...	...	...
page header 10	record 1	...	record 40

- page 1 → 1640 bytes
- page 2 → 1640 bytes

  ⋮
- page 10 → 1640 bytes

3.2.2 Page

Each page contains 40 records. If a page is deleted or becomes full its state is changed accordingly.

Page Header

page state	number of records

- page state → 1 byte
- number of records → 1 byte

Page Records

Record Header 1	field 1	...	field 10
Record Header 2	field 1	...	field 10
...	...	...	...
Record Header 40	field 1	...	field 10

- record 1 → 41 bytes
- record 2 → 41 bytes

  ⋮
- record 40 → 41 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 state

- record state → 1 byte

Record Fields

field 1	field 2	...	field 10

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

  ⋮
- field 10 → 4 bytes

4 Operations

4.1 DDL Operations

4.1.1 Create a type

First gets the names of the files in the related directory. Then checks if there is an available spot. If needed creates a new page. Iterates over pages until find an available one. Loads the page and adds the new type. Then updates page header accordingly.

```
Function createType(typeToCreate: Type):
    if isAvailableFileExist() != True then
        createNewFile()
    fileNameList ← getMetaFileList()
    currType ← typeToCreate
    for each fileName in fileNameList do
        file ← open(fileName)
        for i ← 0 to MAX_PAGE_NUM do
            page ← getPage(file, i)
            for j ← 0 to MAX_TYPE_NUM do
                if page.types[j].state == empty then
                    page.types[j] ← currType
                else if page.types[j].name > currType.name then
                    swap(page.types[j], currType)
            if fileName == fileNameList[-1] and page.state != 1 then
                page.numberOfTypes++ if page.numberOfTypes == MAX_TYPE_NUM
                 then
                    page.state ← 1
                    file.close()
                    return
        file.close()
```

4.1.2 Delete a type (by name)

Iterates over files, pages, types in reverse. Changes all of them with the previous one until override the one that should be deleted. Then updates page header accordingly.

```
Function deleteType(typeNameToBeDeleted: string):
    fileNameList ← getMetaFileList()
    reversedFileNameList ← reverseList(fileNameList)
    currType ← Type()
    for each fileName in reversedFileNameList do
        file ← open(fileName)
        for i ← MAX_PAGE_NUM-1 to -1 do
            page ← getPage(file, i)
            for j ← MAX_TYPE_NUM-1 to -1 do
                if page.types[j].state != empty then
                    swap(page.types[j], currType)
                    if currType.name == typeNameToBeDeleted then
                        file.close()
                        file ← open(reversedFileNameList[0])
                        page ← file.getLastPage
                        page.numberOfTypes–
                        if page.numberOfTypes == 0 then
                            page.state ← 0
                        if isEmpty(file) then
                            file.close()
                            delete(file)
                        return
        file.close()
```

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():
    fileNameList ← getMetaFileList()
    for each fileName in fileNameList do
        file ← open(fileName)
        for i ← 0 to MAX_PAGE_NUM do
            page ← getPage(file, i)
            for j ← 0 to MAX_TYPE_NUM do
                if page.types[j].state != empty then
                    print(page.types[j].name)
        file.close()
```

4.2 DML Operations

4.2.1 Create a record

First gets the names of the files of the related type. Then checks if there is an available spot. If needed creates a new page. Iterates over pages until find an available one. Loads the page and adds the new type. Then updates page header accordingly.

```
Function createRecord(recordToCreate: Record, typeName: string):
    fileNameList ← getDataFileList(typeName)
    if isAvailableFileExist() != True then
        createNewFile()
    currRecord ← recordToCreate
    for each fileName in fileNameList do
        file ← open(fileName)
        for i ← 0 to MAX_PAGE_NUM do
            page ← getPage(file, i)
            for j ← 0 to MAX_RECORD_NUM do
                if page.records[j].state == empty then
                    page.records[j] ← currRecord
                else if page.records[j].primaryKey > currRecord.primaryKey then
                    swap(page.records[j], currRecord)
            if fileName == fileNameList[-1] and page.state != 1 then
                page.numberOfRecords++ if page.numberOfRecords ==
                 MAX_RECORD_NUM then
                    page.state ← 1
                    file.close()
                    return
        file.close()
```

4.2.2 Delete a record (by primary key)

Iterates over files, pages, records in reverse. Changes all of them with the previous one until override the one that should be deleted. Then updates page header accordingly.

```
Function deleteRecord(primaryKeyToBeDeleted: int, typeName: string):
    fileNameList ← getDataFileList(typeName)
    reversedFileNameList ← reverseList(fileNameList)
    currRecord ← currRecord()
    for each fileName in reversedFileNameList do
        file ← open(fileName)
        for i ← MAX_PAGE_NUM-1 to -1 do
            page ← getPage(file, i)
            for j ← MAX_RECORD_NUM-1 to -1 do
                if page.records[j].state != empty then
                    swap(page.records[j], currRecord)
                    if currRecord.fields[0] == primaryKeyToBeDeleted then
                        file.close()
                        file ← open(reversedFileNameList[0])
                        page ← file.getLastPage
                        page.numberOfRecords–
                        if page.numberOfRecords == 0 then
                            page.state ← 0
                        if isEmpty(file) then
                            file.close()
                            delete(file)
                        return
        file.close()
```

4.2.3 List all records of a type

Iterates over files, pages, records of the related type. While iterating over them prints the fields of the records.

```
Function listAllRecords(typeToList: Type):
    fileNameList ← getDataFileList(typeName)
    for each fileName in fileNameList do
        file ← open(fileName)
        for i ← 0 to MAX_PAGE_NUM do
            page ← getPage(file, i)
            for j ← 0 to MAX_RECORD_NUM do
                if page.records[j].state != empty then
                    print(page.records[j].fields)
        file.close()
```

4.2.4 Update a record (by primary key)

First deletes the old version of the record, then creates the new version of the record.

```
Function updateRecord(primaryKeyToBeUpdated: int, newRecord: Record):
    fileNameList ← getDataFileList(typeName)
    reversedFileNameList ← reverseList(fileNameList)
    currRecord ← currRecord()
    deletionCompleted ← False
    for each fileName in reversedFileNameList do
        if deletionCompleted == True then
            break
        file ← open(fileName)
        for i ← MAX_PAGE_NUM-1 to -1 do
            if deletionCompleted == True then
                break
            page ← getPage(file, i)
            for j ← MAX_RECORD_NUM-1 to -1 do
                if page.records[j].state != empty then
                    swap(page.records[j], currRecord)
                    if currRecord.fields[0] == primaryKeyToBeDeleted then
                        deletionCompleted ← True
                        break
        file.close()
    currRecord ← newRecord
    for each fileName in fileNameList do
        file ← open(fileName)
        for i ← 0 to MAX_PAGE_NUM do
            page ← getPage(file, i)
            for j ← 0 to MAX_RECORD_NUM do
                if page.records[j].state == empty then
                    page.records[j] ← currRecord
                else if page.records[j].primaryKey > currRecord.primaryKey then
                    swap(page.records[j], currRecord)
        file.close()
```

4.2.5 Search for a record (by primary key)

Iterates over files, pages, types; of system catalog. When find the right one, prints the result and stops iterating.

```
Function searchRecord(primaryKeyToSearch: int, recordType: Type):
    fileNameList ← getDataFileList(typeName)
    for each fileName in fileNameList do
        file ← open(fileName)
        for i ← 0 to MAX_PAGE_NUM do
            page ← getPage(file, i)
            for j ← 0 to MAX_RECORD_NUM do
                if page.records[j].state != empty then
                    if page.records[j].fields[0] == primaryKeyToSearch then
                        print(page.records[j].fields)
                        return
        file.close()
```

5 Changes

This section is about the changes in the report. All changes in the report is highlighted in order them to make them easy to notice. Further explanation about the changes is provided in below bullets.

- File headers are removed from the project.
- Linked list like structure between files, and pages are changed. Instead index based address derivation is used.
- Data page headers and meta (system catalog) page headers has similar structures. (Before they were bit different due to singly/doubly linked list approach.)
- Record and type sizes/structures are updated according to HW-2 assumtions. (Also other sizes/structures updated depending these two.)
- Id fields are removed from the record headers and page headers.
- Pseudo-codes are updated according to new approach.
- Helper functions removed from the project.
- Introduction and Conclusion & Assessments parts are updated accordingly.

6 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. Those are file, page, record, and type. Pages of system catalog are contains page header and types. Similarly, pages of data storage units are contains page header and records.

Precedence relation between files are derived by checking their names. File name formats contains an 10 digit number and that way system can understand order of the files. Addresses of pages are derived by their indexes. When system working it knows which file index it is in therefore it is easy to calculate the address of the next and previous file easily. When system goes the address of a page, loads whole page into memory makes necessary changes and writes page back.

During creation operations system first checks if there is an available space exist, if there is no available space creates a new file and continues the process. In the deletion operations after deletion completed last file is checked. If it becomes empty, gets deleted. List and search operations are very strait forward, traverses and along the way prints the necessary parts/part. Update operation is basically performed by first deletion, then creation.

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 possible, and keeping it sorted and ordered. However in order to perform that deletion and creation operations performs a lot of swap operations, and it might cause an overhead.

To sum up, design is meets the desired functionalities. It has both powerful and improvable sides.