Storage Manager System
Implementation

CMPE321

Project 2

Efe Önal

2016400267

April 7, 2019

Contents

Introduction

In this project I implemented the design I made in Project1 by making the necessary changes. In the implementation, records of each type is held in a file, there is no storage restriction for the files. Every file is read into the system page by page (1024 byte by 1024 byte) and necessary changes are made in the files throughout the execution. Also, a system catalog (SystemCatalog.txt) is updated and read in associated operations.

Assumptions & Constraints

- User always enters valid input.
- Fields are always integers.
- Type names can be alphanumeric.
- Primary key of a record is always the first field of that records.
- Field names and values cannot be longer than eight characters.
- Deleting a record is done by primary key.
- Searching a record is done by primary key.
- Pages are fetched into the system one by one.
- A page is 1024 bytes.
- There is no length restriction for files.
- A file is always named same as the type name.
- The data must be organized in pages and pages must contain records.
- A file must contain multiple pages, and a type consists of only one file.
- A record can not have more than 255 fields.
- First byte of a page holds the number of records in that page.

Data Structures

In this storage manager system, a file can hold only records of one type and the type name is the same as the file name. A type is actually a file that is managed by the system. Each file consists of multiple pages which in return consists of multiple records.

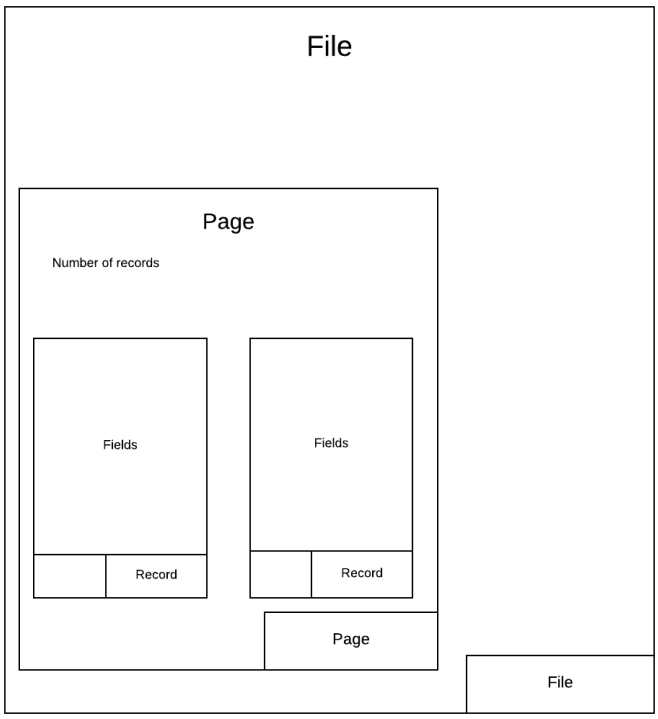

Figure 1: General Data Structure

System Catalog

The system catalog (SystemCatalog.txt) is a very important file which is separate from the files that hold the records. System catalog contains general information about the database: number of tables in the database, names of the tables, number of records in each file, number of fields of each type, names of the .txt files, and number of pages of the files. The system catalog is opened and updated after every write operation and operations are made according to the information in the system catalog.

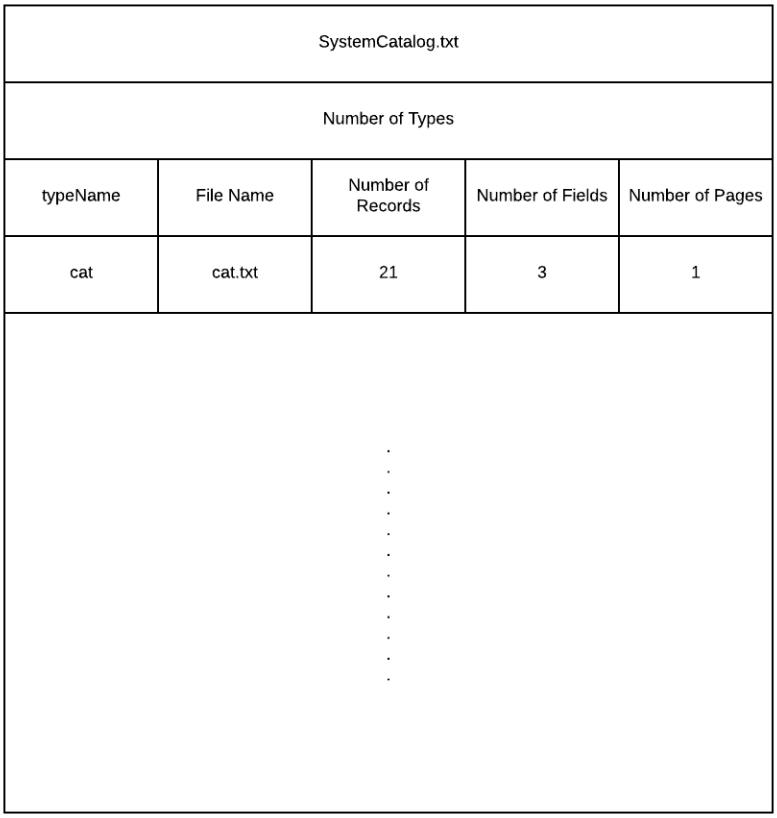

Figure 2: System Catalog Structure

File Structure

In this DBMS, each type is held in a file that consists of multiple files and there is no storage limit for files. Every file consists of pages which hold the records. The files are only consecutive bytes which are the the fields of records. Only, the first byte in every 1024 bytes states how many bytes after the first byte are parts of a record, number of records.

If the user creates a new table, then a new file is automatically created by the system and the system catalog is updated. Also, when a type is deleted, associated file is also deleted.

When a new record is added to a table, if the last page of the file is full, a new page is created and written into the file.

In the case of a search, the associated file is read into the system page by page. For updates and deletes, the page is written back into the file.

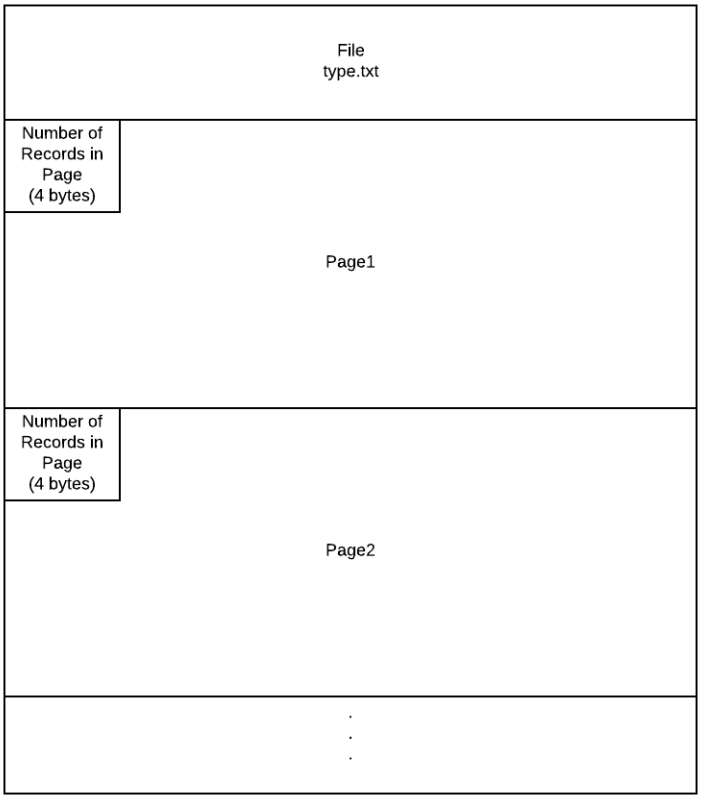

Figure 3: File Structure

Page Structure

A page is a byte that indicates the number of records and fields of those records. If the number of records in a page is the same as the maximum number of records for the page which is calculated by the system, then a new page is created when a new record is created by the user. Again, system catalog is updated according to the operations made.

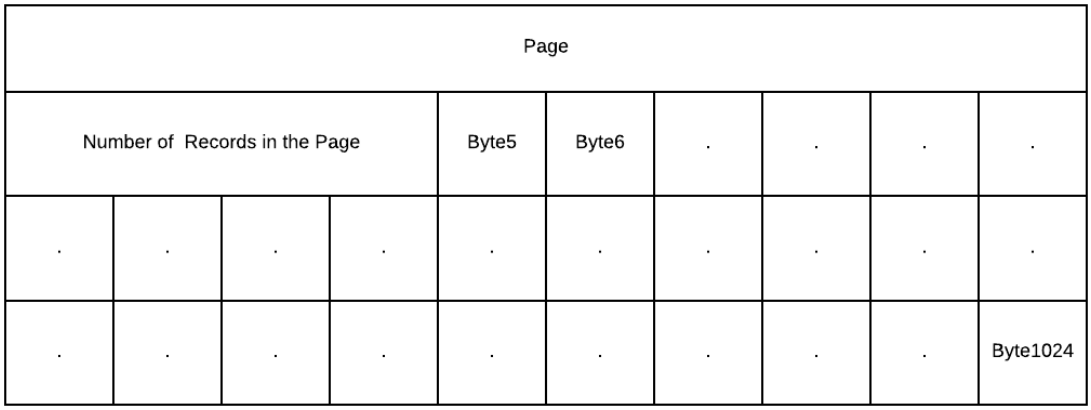

Figure 4: Page Structure

Record Structure

A records is only consecutive bytes (integer fields) in a file.

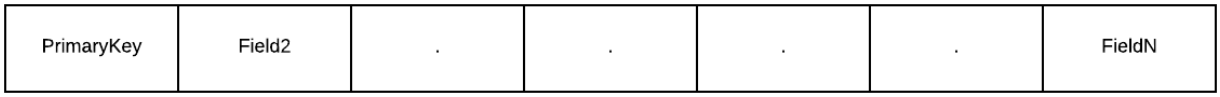

Figure 5: Record Structure

Operations

DDL Operations

In this function we create the associated file and add the new type to the Types vector of the system.

Algorithm 1 Create Type

```
function createType(string typeName, int numOfFields )
    for i ← 0 to this.Types.size() do
        if typeName == this.Types[i].typeName then
            return −1
        end if
    end for
    newType(typeName, numOfFields)
    newType.sizeOfOneRecord ← 4 ∗ numOfFields
    Types.push(newType)
    ofstreamoutfile(typeName + ".txt")
    outfile.close()
    this.numberOfTypes + +
    this.updateSystemCatalog()
    return
end function
```

In this function we delete the file of the type and update fields of the system as well as the system catalog. Also the type is erased from the Types vector.

Algorithm 2 Delete a Type

```
function deleteType(string typeName)
    for i ← 0 to this.Types.size() do
        if typeName == this.Types[i].typeName then
            string filename ← this.Types[i].typeName + ".txt"
            remove(filename.c_str())
            this.Types.erase(this− > Types.begin() + i)
            this.numberOfTypes − −
            this.updateSystemCatalog()
            return
        end if
    end for
    return
end function
```

In this function we first sort the Types vector by making use of the customized comparator less_than_key. Then we pushed all elements of this vector to another vector and returned it.

Algorithm 3 List All Types

```
function listAllTypes( )
    vector < string > myVector
    sort(this.Types)
    for i ← 0 to this.Types.size() do
        myVector.push(this.Types[i].typeName);
    end for
    return myVector
end function
```

DDL Operations

This function is called by createRecord(string typeName, vector<int> fields) function of SystemCatalog class. It checks all pages one by one and when it finds a page that is not full, it puts the new record in this page, writes the page back to the file and returns. If all pages are full, a new page is created.

Algorithm 4 Create a Record

```
function createRecord(vector < int > fields)
    vector < int > myVector
    Record myRecord(this.typeName, this.numOfFields)
    for i = 0 to this.numOfPages do
        int pivot ← 0
        myVector ← readPage(i)
        if myVector[0] < maxNumOfRecords then
            for k ← 0 to k < myVector[0] do
                for j ← 0 to j < this.numOfFields do
                    Update records fields
                    pivot + +
                end for
                push the record to records
                clear record fields
                for m ← 0 to numOfFields do
                    push new fields to myRecord
                end for
                this.records.push(myRecord)
                writePage(i)
                this.numOfRecords + +
                this.records.clear()
                return

            end for
            for i ← 0 to this.numOfFields do myRecord.fields.push(Fields[i])
            end for
            createNewPage
            this.records.push(myRecord)
            writePage(this.numOfPages − 1)
            this.numOfRecords + +
            this.recordds.clear()
            return
```

This function lists all records of a type. It does so by filling the output vector with each page and returning the output vector. It is called by listRecordsOfType(string typeName) function of SystemCatalog class.

Algorithm 5 List All Records of a Type

```
function listRecords
    vector output
    line
    pivot ← 1
    myVector
    recordNumberOfPage
    myRecord(this.typeName, this.numOfFields)
    for i ← 0 to this.numOfPages do
        myVector ← readPage(i)
        recordNumberOfPage ← myVector[0]
        for k ← 0 to recordNumberOfPage do
            for p ← 0 to this.numberOfFields do
                myRecord.fields.push(next field)
                pivot + +
            end for
            this.records.push(myRecord)
            myRecord.fields.clear()
        end for
        for y ← 0 to records.size() do
            for x ← 0 to this.nnumOfFields do
                Fill the line
            end for
            Fill output by lines
        end for
        pivot ← 1
        this.records.clear()
    end for
    sort(output)
    return
end function
```

This function searches all pages of a file and if it finds the record, it returns the fields of the record. It is called by searchRecord(string typeName, int primaryKey) function of SystemCatalog class.

Algorithm 6 Search a Record

```
function searchRecord(primaryKey)
    vector output
    myRecord(this.typeName)
    for j ← 0 to numOfPages do
        myVector ← readPage(j)
        for k ← 0 to myVector[0] + 1 do
            for l ← 0 to do
                myRecord.fields.push(nextFieldIntheVector)
            end for
            this.records.push(myRecord)
            myRecord.fields.clear()
        end for
        for m ← 0 to this.records.size() do
            if record is record to be deleted then
                for f ← 0 to numOfFields do
                    fill output vector
                end for
            end if
        end for
        this.records.clear()
    end for
    return
end function
```

This function searches all pages one by one and when it finds the record, it erases the record decreasing numOfRecords and writes the page back. It is called by deleteRecord(string typeName, int primaryKey) function of SystemCatalog class which also calls updateSyatemCatalog() function.

Algorithm 7 Delete a Record

```
function deleteRecord(primaryKey)
    pivot ← 1
    vector < int > myVector
    reccordNumberOfPage
    myReocrd(this.typeName, this.numOfFields)
    for i ← 0 to this.numOfPages do
        myVector ← readPage(i)
        recordNumberOfPage ← myVector[0]
        for k ← 0 to recordNumberOfPage do
            for p ← 0 to this.numOfFields do
                myRecord.fields.push(myVector[pivot])
                pivot + +
                this.records.push(myRecord)
                myRecord.fields.clear()
            end for
            for l ← 0 to this.records.size() do
                if records[l].fields[0] == primaryKey then
                    this.records.erase(this.records.begin() + l)
                    this.numOfRecords − −,
                    break
                end if
            end for
            writePage(i)
            this.records.clear()
            return
        end for
        return
```

This function searches all pages one by one and when it finds the record, it changes the records fields by input fields vector and writes back the page. It is called by updateRecord(string typeName, int primaryKey) function of SystemCatalog class which also calls updateSyatemCatalog() function. =0

Algorithm 8 Update a Record

```
function updateRecord(primaryKey, vector fields)
    pivot ← 1
    vector < int > myVector
    reccordNumberOfPage
    myReocrd(this.typeName, this.numOfFields)
    for i ← 0 to this.numOfPages do
        myVector ← readPage(i)
        recordNumberOfPage ← myVector[0]
        for k ← 0 to recordNumberOfPage do
            for p ← 0 to this.numOfFields do
                myRecord.fields.push(myVector[pivot])
                pivot + +
                this.records.push(myRecord)
                myRecord.fields.clear()
            end for
            for l ← 0 to this.records.size() do
                if records[l].fields[0] == primaryKey then
                    records[l].fields.clear()
                    for q ← 0 to fields.size() do
                        this.records[l].fields.push(fields[q])
                    end for
                    break
                end if
            end for
            writePage(i)
            this.records.clear()
            return
        end for
        return
```

Conclusions & Assessment

I have implemented a simple storage manager system. DML and DDL operations are explained with pseudo code.

Unfortunately, in my design the only structure with a header is page and pages have only one byte of a header: number of records in the page. If more detailed headers were created for files, pages, and records, further implementations would be easier.

Also, in type operations an output vector is used to return the output although the types vector could be returned right away.