CMPE 321 - 2020 Spring - Project1

Storage Management System

İsmet Sarı
2016400324

April 23, 2020

Contents

Contents

1 Introduction

The project is named storage management system. This system was designed to make some ddl operations, which are creating type, deleting type, and listing all types and to make some dml operations, which are creating record, deleting record, searching a record by using primary key, updating a record by using primary key and listing all records of a type. The these operations are explained in pseudo-code format in Algorithms section. We explained the whole system in four parts: Assumption and Constraints, Mathematical Demonstrations, Storage Structures, and Algorithms.

To make everything clear some assumptions and constraints are defined clearly such as how many files can be created for a type to store records etc. In addition to basic parts of storage, another storage structure file called indexFile is defined. In this new file we can store file id, page id, record id, and primary key of a record in order to find the record when the searching or updating is needed.

We make some mathematical operations to show how we specify these assumptions and constraints. Afterwards the visual figures of storage mechanism comes up and finally all the algorithms of all operations are written. Other than the algorithms of the ddl and dml operations, we defined other algorithms which is needed in dml and ddl operations such us linkfiles or control functions which controls if a type we try to add is already present or not.

In this Report we try to explain whole concept of the system and have made some assumption such as the user always enter correct inputs and you can see how the correct the input needs to be below but in any case we control the user input as much as possible as well. The parameters of a specific algorithm are given in procedure row of the algorithm. The system does not read the whole file instead it reads page by page so we just read the beginning of the file to see variables of the file such as the NumberOfRecords, NumberOfPages, etc.

2 Assumptions & Constraints

2.1 General Assumptions & Constraints

- User always enters correct inputs
- A disk manager already exists that is able to fetch the necessary pages when addressed.
- All fields shall be integers. However, type and field names shall be alphanumeric.
- All alpha numerical characters is defined in UTF-8 character set, which means that one character corresponds to 1 byte of size.
- User cannot create a record with a primary key which intersects with another primary key of another records of the same type.
- User cannot delete or update a record which is not present in data base.
- The primary key of a record is unique in a particular type which means that if a record of humans type is 36 then the primary key of another record of animals type can be 36 as well.
- All data must be organized in pages and pages must contain records.
- Records in the files should be stored in ascending order according to their primary keys.
- 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 system is not allowed to store all pages in the same file and a file must contain multiple pages. This means that your 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.

2.2 System Catalog Structures

- System catalog consist of all the information of types and total number of types.
- The system can store 2^{32} - 1 number of types so 4 bytes are allocated to store how many types are stored.

2.3 Type Structure

- A type can have at most 255 file to store all records so 1 byte is allocated to store the number of it.
- For every type we define an other file called indexFile.

- A type can have 64 field so 1 byte is allocated to store this value.
- After we create the type the order the fields cannot be changed so the field names stored in a specific type corresponds to the fields stored in a record of this specific type.
- We are assuming all inputs are valid so we can say that in the process of creating the type the all field names are not repeated.
- 10 bytes are allocated to store type name and fields names in Type structure
- 10 bytes are allocated to store fields names in Type structure

2.4 File Structure

- Each file must have allocated 1 byte to store number of Records in itself.
- Each file must have allocated 1 byte to store number of pages in itself.
- Each file must have allocated 1 byte to location of next file. If this is 255 then we do not have the next file.
- Each file must have allocated 1 byte to location of previous file. If this is 255 then we do not have the previous file.
- Each file is created for only one type.
- Each file has a type name of up to 10 character long to open a file on a system like open('humans'+'3').
- Creating of a file is dynamic, which means that created file does not cover its full size.
- Every file created for a particular type and it's fileid is stored in indexfile. FileID 0 means first file of a type.

2.5 indexFile Structure

- For each type there will be one indexFile to detect the location of a record easily.
- In indexFile the MaxNumberOfRecordPerFile variable can be max 256*255=65025 so 4 bytes are allocated to store it.
- To store total number of records of all files 4 bytes are allocated.
- The indexFile will be created as createFile('TypeName'+'index')
- The order of information of a record is FileID, PageID, RecordID, and PrimaryKey.

- All ID's begins with zero such as PageID 0 means first page of a file.
- The FileID cover 1 byte since we can have at most 255 file for a type.
- The PageID cover 1 byte since we can have at most 255 page in a file.
- The RecordID cover 1 byte. We can store at most 255 record.
- 4 bytes are allocated to store primary key of a record.
- The primary key will be used to take the other aspects of a record (FileID, PageID, RecordID).
- The indexFile will be sorted according to primary key after any create,delete and update operations on a record.

2.6 Page Structure

- A page has allocated 2KB memory space but it could be 2040 and 1020 if the fields no is 1 or 2 respectively.
- A page has allocated 1 byte to store the number of records in itself
- Every page created in a particular file has a unique page id and it is stored in indexfile. PageID 0 means first record of a page.

2.7 Record Structure

- Every record created in a particular page has a unique record id and it is stored in indexfile. RecorID 0 means first record of a page.
- The primary key of a record is assumed to be the value of the first field of that record.

3 Constraints

- A type can have maximum 255 files,
- A type can have maximum 64.
- A file can have maximum 255 pages.
- A page can have maximum 170 and minimum 8 records.
- A record can have maximum 64 fields.
- A length of a type name can be maximum 10 character long.
- A length of a type fields name can be maximum 10 character long.

4 Mathematical Demonstrations

Some particular constraints are explained mathematically.

- A type at least has 1 field.

$\Longrightarrow$ A record min cover 4 bytes in memory.

$$\left\lfloor \frac{2\ KB}{4\ Bytes} \right\rfloor = 512 \xRightarrow{\text{After Restriction}} = 255.$$

The page memory is 2KB and if the record is as big as it can be then we can store 8 record in a page.

- A type at least has 64 field.

$\Longrightarrow$ A record max cover 256 bytes in memory.

$(64)* \underbrace{4\ bytes}_{The\ Field} = 256\ bytes.$

$$\left\lfloor \frac{2\ KB}{256\ Bytes} \right\rfloor = 8.$$

MaxNumberOfRecordsPerFile will be calculated as.

$$if \left\lfloor \frac{2\ KB}{4 * \#\ Fields} \right\rfloor > 255\ \ then \left\lfloor \frac{2\ KB}{4 * \#\ Fields} \right\rfloor = 255$$

$$255 * \left\lfloor \frac{2\ KB}{4 * \#\ Fields} \right\rfloor = MaxNumberOfRecordsPerFile.$$

Max value of this number is 65025 min 2040.

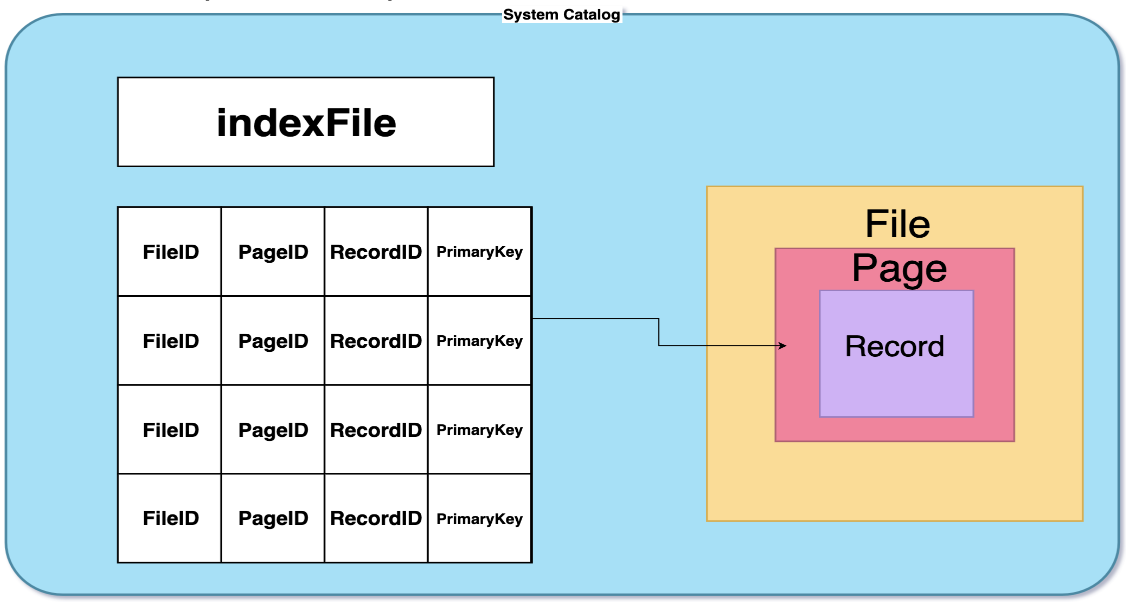
$$255 * \left\lfloor \frac{2\ KB}{4 * 64} \right\rfloor = 2040.$$

The page memory is generally 2KB. If the record is smallest then it has 1 field. 2048/4 = 512 then it would be 255. The page size will be 4*255=1020. If the record has 2 fields then 2048/8 = 256 then it would be 255 again. The page size will be 2*4*255=2040.

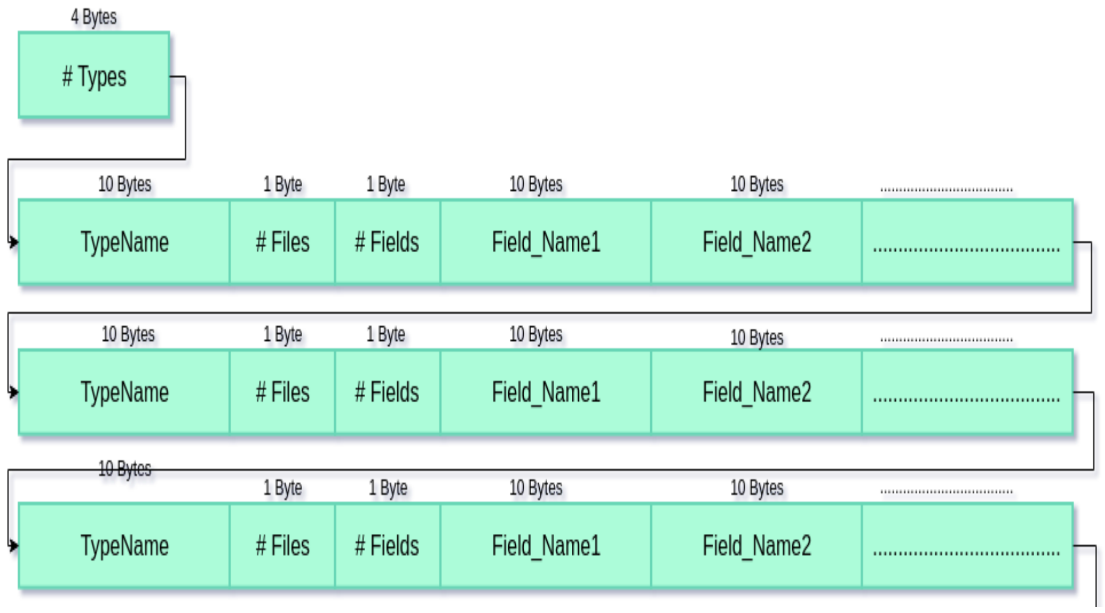
5 Storage Structures

5.1 System Catalog Structures

$\Longrightarrow$ The whole system is actually look like

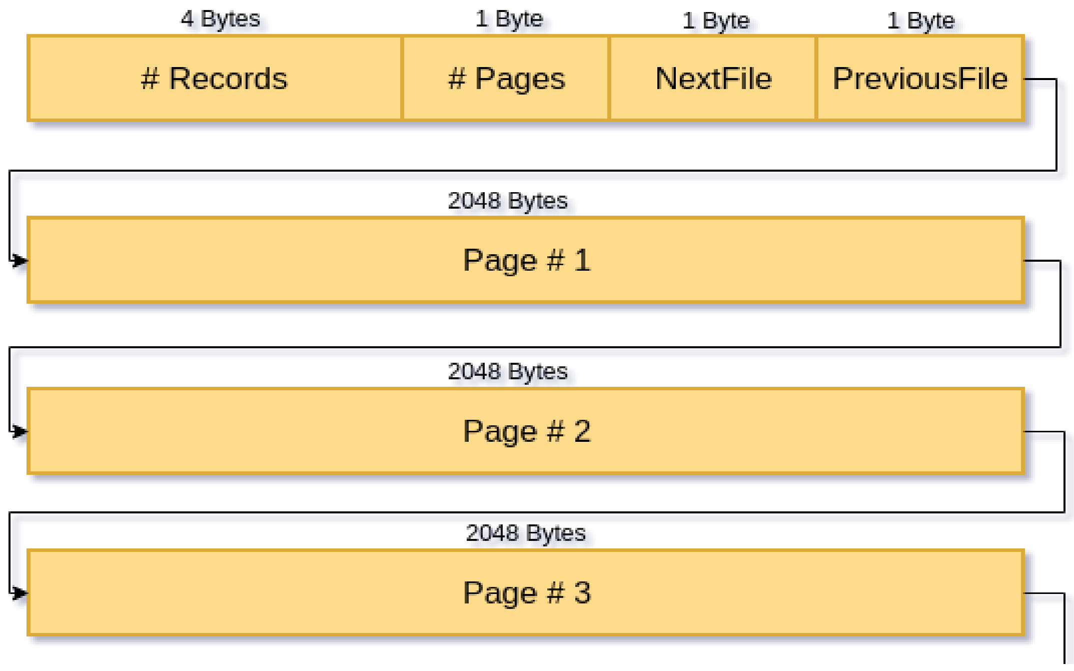


$\Longrightarrow$ The System catalog handles the operation of placing the all types like



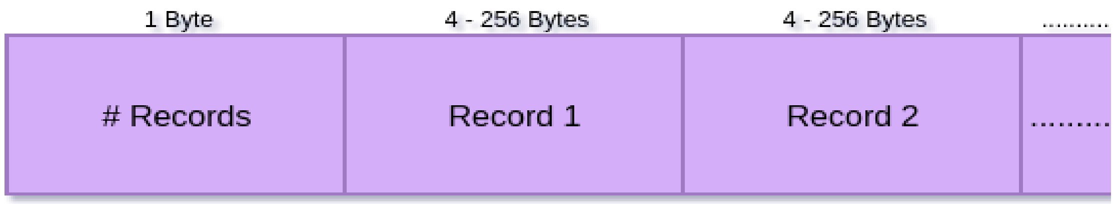
5.2 File Structures

⟹ The File Hierarchy is going to look like



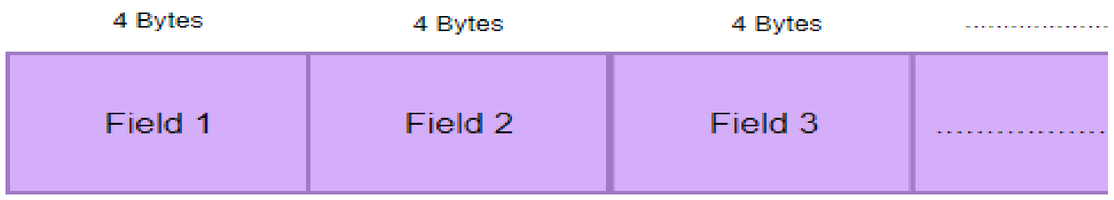
5.3 Page Structures

⟹ The Page Hierarchy is going to look like



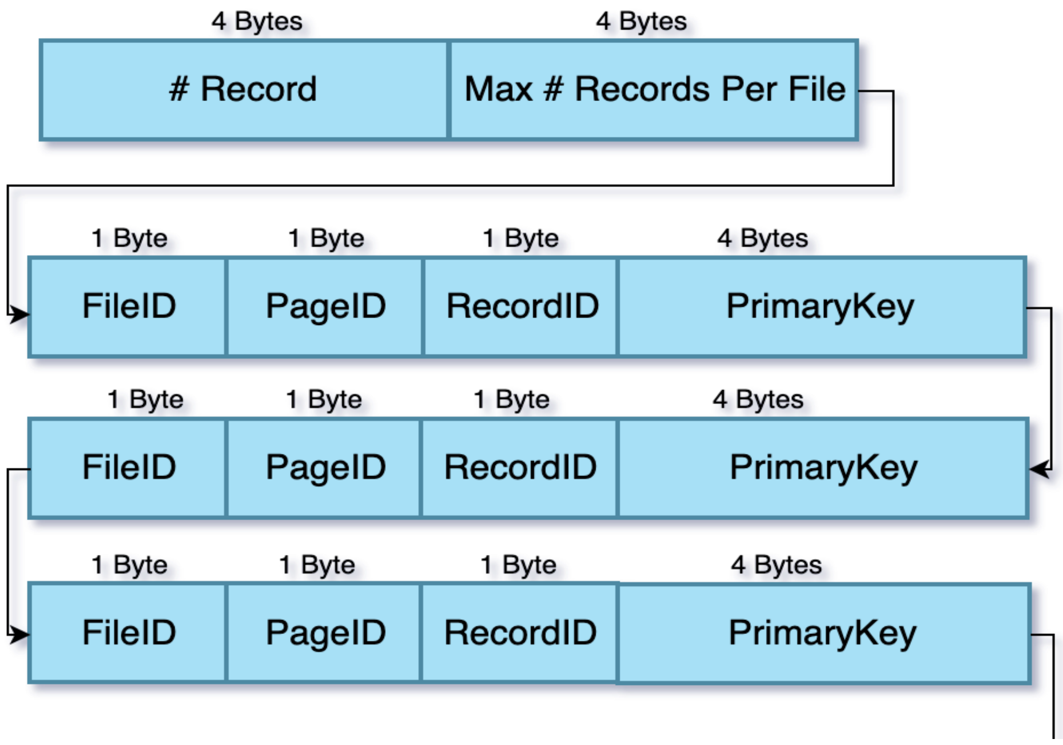
5.4 Record Structures

⟹ The Record Hierarchy is going to look like



5.5 indexFile Structures

⟹ The indexFile Hierarchy is going to look like



6 All Changes From Previous Report

- The File structure does not include isFull variable and we also don't store this variable in files.
- The TypeName can be maximum 10 character long.
- By executing operation we don't open files and make changes. Instead when opening system catalog as the program is at first executed, the all data structures are created and we make operation on these data structures and when finalize the program the all data structures are written back on their regarding files.
- The creating file is now dynamic which means that as we create record the file will be extended.
- The page size is not always 2KB anymore it can change based on the number of fields. If the record is smallest then it has 1 field. The number of records 2048/4 = 512 then it is decreased to 255. Then the page size will be 4*255=1020. If the record has 2 fields then 2048/8 = 256 then it is decreased to 255 again. The page size will be 2*4*255=2040. If we have 6 fields then floor(2048/24)=85 and our page size will be again 85*24=2040.
- Max number of files of a type is 255 instead 256.
- Max number of pages in a file is 255 instead 256.
- Max number of records in a page is 255 instead 170.
- Minimum number of fields is 1 instead 3.
- The length of type name is changed to 10.
- The length of field name is changed to 10.
- The File is being used full size there is no byte that is being wasted.
- I don't control anymore whether or not if the input is correct or not, so I don't accept any responsibility if the program is crashed due to incorrect input.
- We don't control whether or not a types fields no and its created record's fields no are the same.
- We don't control whether or not created type already exists.
- We don't control whether or not the type we want to delete is present.
- We don't control whether or not the record we want to add is already present.
- We don't control whether or not the record we want to make some operation on is present.

7 Conclusion

The overall Storage Management System has been defined by 4 steps: at first Assumptions Constraints is explained, and then we made some mathematical calculation to show how we specified the byte number of a variable then we made some visual figures in Storage Structures section to see whole concept of our database system,and finally Algorithms which is explained in section below. The design covers most of the basic operations of DML and DLL. It uses some fixed constant such as max number of records for a file,max number of files for types, and for our over-all system max number of types. These contradictions help us determine whether or not we can have capacity to add one more record or type to our system. In addition to this we define an indexFile. This aspect of the system give us time when we try to detect the location of a record we want to make some operation on. Using an index file assure us the system would spend less time then inspecting all the files one by one. Opening a page and file cost a lot for us so we just open this indexFile and read the records according to their last aspect which is primary key. Since the indexFile is already sorted we can find our records quite easily by using a basic searching algorithm.

As for disadvantages, this Storage Management System needs a lot free memory to store some variable in files. This variables save us time but occupy memory as well. However as we explained above having this kind of variables we store increase performance quite highly because we are decreasing disk accesses. The system should be designed in this way because people prefer to spend more on memory than waiting.

This system also does not guarantee that your operation would be safe after some possible system crashes such as power disconnections. To get the some of the system information we can create some back-up files periodically uses them when this kind of problems occur.

8 Algorithms

Algorithm 1 insertRecord in insertFile

```
procedure INSERTRECORD(Fields,TheindexFile,Type,MaxNumberOfRecordsPerPage)

    $TheindexFile.NumberOfRecords++$
    $M \leftarrow MaxNumberOfRecordsPerPage$
    $index = FindRecordIndex(Fields[0], indexFile)$
    if $index = 0$ then
        TheindexFile.insert(0,RecordinindexFile(0,0,0,Fields[0]))
        Type.Files[0].Pages[0].Records.insert(0,Record(Fields))
        if Type.Files[0].NumberOfRecords less then 255*M then
            if Type.Files[0].Pages[0].NumberOfRecords less then M then
                Type.Files[0].Pages[0].NumberOfRecords++
            end if
            Types.Files[0].NumberOfRecords++
        end if
        return 1
    end if
    FID=TheindexFile.Records[index-1].Fileid
    PID=TheindexFile.Records[index-1].Pageid
    RID=TheindexFile.Records[index-1].Recordid
    indexFile.Records.insert(insert,RecordinindexFile(FID,PID,RID,Fields[0]))
    Type.Files[FID].Pages[PID].Records.insert(RID+1,Record(Fields))
    if Type.Fields[FID].NumberOfRecords less than 255*M then
        if Type.Files[FID].Pages[PID].NumberOfRecords less than M then
            Type.Files[FID].Pages[PID].NumberOfRecords++
        end if
        Type.Files[FID].NumberOfRecords++
    end if
    return index
end procedure
```

Algorithm 2 Link Two Files

```
procedure LINKFILES(TypeName,FirstFileID,SecondFileID)
    $FirstFile \leftarrow Types[TypeName].Files[FirstFileID].Previous$
    $SecondFile \leftarrow Types[TypeName].Files[SecondFileID].Previous$
    $FirstFile.NextFile \leftarrow SecondFileID$
    $SecondFile.PrevFile \leftarrow FirstFileID$
end procedure
```

Algorithm 3 Cut The Link Between Two Files

```
procedure CUTTHELINKPREVIOUSCURRENT(TypeName,FileID)
    $CurrentFile \leftarrow Types[TypeName].Files[CurentFile].Previous$
    if $CurrentFile.PrevFile \neq 255$ then
        $PreviousFile \leftarrow Types[TypeName].Files[PreviousFile].Previous$
        $PreviousFile.NextFile \leftarrow 255$
    end if
end procedure
```

Algorithm 4 Create Type

```
procedure CREATING_TYPE(TypeName,NumberofFields,FieldsNames,PrimaryKey)

    /*There is a method defied in Types in SystemCatalog called append that
    adds the new type at the end of the Types array in SystemCatalog*/

    $SystemCatalog.Types.append(TypeName, 0, NumberofFields, FieldsNames)$
    $SystemCatalog.NumberOfTypes++$

    //as creating file the all variables are set their defaults values
    // all numbers are 0 and boolean is false.
    $createFolder(TypeName)$                    ▷ Creating the folder to store files

    $MaxNumber \leftarrow floor(2048/(4 * NumberOffFields))$

    $indexFile \leftarrow createFile(TypeName + "index")$

    $indexFile.MaxNumberOfRecordsPerFile \leftarrow MaxNumber$

end procedure
```

Algorithm 5 Delete Type

```
procedure DELETING_TYPE(TypeName)
    /*There is a method defied in Types in SystemCatalog called
    remove that removes the type from the Types array in SystemCatalog*/

    $SystemCatalog.Types.remove(TypeName)$
    $SystemCatalog.NumberOfTypes--$

    $DeleteFolder(TypeName)$
    $DeleteFile(TypeName + index)$
end procedure
```

Algorithm 6 List All Types

```
procedure LIST_ALL_TYPES( )

    $NumberofTypes \leftarrow SystemCatalog.NumberofTypes$

    for $i = 0 \rightarrow NumberofTypes$ do
        $print(SystemCatalog.types[i].typename)$
    end for
end procedure
```

Algorithm 7 Create Record

```
procedure CREATING_RECORD(TypeName,NumberofFields,Fields)
    $indexFile \leftarrow SystemCatalog.indexFiles[TypeName]$
    $MRPrF \leftarrow indexFile.MaxNumberOfRecordsPerFile$
    $MRPrP \leftarrow indexFile.MaxNumberOfRecordsPerFile/255$
    $index \leftarrow indexFile.insertRecord(Fields[0]), File \leftarrow NULL$
    for $i = index \rightarrow indexFile.NumberOfRecords$ do
        if $++indexFile.Records[i].RecordID > MRPrP$ then
            $indexFile.Records[i].RecordID = 1, NewPage \leftarrow NULL$
            $F \leftarrow indexFile.Records[i].FileID$
            $OLDFile \leftarrow SystemCatalog.Types[TypeName].Files[F]$
            $OldPageID \leftarrow indexFile.Records[i].PageID++$
            $OLDpage \leftarrow OLDFile.Pages[OldPageIP]$
            if $indexFile.Records[i].PageID > 255$ then
                $ID \leftarrow ++indexFile.Records[i].FileID$
                $indexFile.Records[i].PageID = 1$
                if $Types[TypeName].Files == ID$ then
                    // the it means that we don't have file so create
                    $NewFile \leftarrow createFile(TypeName + ID)$
                    $NewFile.NumberOfPages++$
                    $NewFile.NumberOfRecords++$
                    $NewPage.NumberOfRecords++$
                    $linkfiles(TypeName, OLDFile.FileID, NewFile.FileID)$
                    $NewPage \leftarrow Types[TypeName].Files.[ID].Pages[0]$
                else
                    $NewPage \leftarrow OLDFile.Pages[indexFile.Records[i].PageID]$
                end if
                $NewPage.AddRecordFront(OLDpage.PopLastRecord)$
            end if
        end if
    end for
end procedure
```

Algorithm 8 Delete Record

```
procedure DELETING_RECORD(TypeName,FileID,PageID,RecordID)
    $PrimaryKey \leftarrow Page.Records[RecordID - 1]$
    $MaxRePerFile \leftarrow File.MaxNumberOfRecordsPerFile$
    $MaxRePerPage \leftarrow File.MaxNumberOfRecordsPerPage$
    $Record, index \leftarrow indexFile.findrecord(PrimaryKey)$
    $del\ RecordinitsFile$
    indexFile.NumberofRecords–
    for $i = 0 \rightarrow indexFile.NumberofRecords$ do
        R=indexFile.Recods[i]
        R.RecordID–
        if R.RecordID¡0 then
            R.RecordID=MaxRePerPage
            R.PageID–
            if R.PageID¡0 then
                R.pageID=254
                R.FileID–
                newR=popfirstelementofFile[R.FileID+1]
                appendcurrentfile(newR)
            end if
            newR=popfirstelementofPage[R.PageID+1]
        end ifappendcuurentPage(newR)
    end for
    //Here we traverse all files and pages
    // and assign the size of all pages to pagesnumberofrecords
    //and assign the total number of pagesnumberofrecords to
    // File.NumberOfRecords
    UpdateStateNumbers()
    //After that Update function
    for All Files do
        for All pages in a file do
            if pages.numberOFrecords==0 then
                del page
                File.NumberOfPages–
            end if
        end for
        if File.NumberOfRECORDS==0 then
            cutlinkbetweenpreviousone(file)
            del file
            Types[TypeName].NumberOfFiles–
        end if
    end for
end procedure
```

Algorithm 9 Update Record

```
procedure UPDATING_RECORD(TypeName,PrimaryKey,NewFields)
    Record=BinarySearch(indexFile,PrimaryKey)
    FID=Record.FileID
    PID=Record.PageID
    RID=Record.RecordID
    Record=Types[TypeName].Files[FID].Pages[PID].Records[RID]
    Record.Fields=NewFields
end procedure
```

Algorithm 10 Search a Record By Primary Key

```
procedure SEARCHING_RECORD(TypeName,PrimaryKey)
    Record=BinarySearch(indexFile,PrimaryKey)
    FID=Record.FileID
    PID=Record.PageID
    RID=Record.RecordID
    Record=Types[TypeName].Files[FID].Pages[PID].Records[RID]
    Print(Record.Fields)
end procedure
```

Algorithm 11 List All Records Of A Type

```
procedure LIST_ALL_RECORDS_OF_A_TYPE(TypeName)
    Type=SystemCatalog.Types[TypeName]
    for $a = 0 \rightarrow Type.NumberOfFiles$ do
        File=Types.Files[a]
        for $b = 0 \rightarrow File.NumberOfPages$ do
            Page=File.Pages[b]
            for $c = 0 \rightarrow Page.NumberOfRecords$ do
                print(Page.Records[c].Fields)
            end for
        end for
    end for
end procedure
```