CmpE 321 Project 2

Spring 2019

Implementing Storage Manager

Aysu Sayın 2016400051

April 7, 2019

Contents

1 Introduction

In this project, a Storage Manager that we designed in the first project is implemented. Design is changed a bit due to the requirements of the second project. Following DDL and DML operations are implemented in Python3:

DDL:

- Create a type
- Delete a type
- List all types

DML:

- Create a record
- Delete a record
- Search for a record
- List all records for a given type
- Update a record

This document is for explaining the details of design and implementations using pseudo-code. I used primary dense indexing in my design. The name of the first files of a type are in the form of *typename.fl* and the name of the index files are *typename_index.fl*. The metadata is kept in *sys_cat.fl*. All these files are binary files.

2 Assumptions & Constraints

- All field values are integers.
- System Catalog can store infinite number of records in a file.
- Each character is 1 byte.
- Each field name is maximum 8 characters long.
- Each field name must be unique.
- A data type can have maximum of 6 fields. If there are less fields, they are considered to be null.
- A data type name is maximum 10 characters long.
- Each file keeps only one type of record.
- When a type is created, a file for that type is created by default.
- A file can have more than one pages.
- User gives valid input so there is no need to check for errors in the algorithms.
- All records are fixed-sized for all types.
- A file is maximum 105 MB.
- A page is maximum 1047 Bytes.
- A page is created with 40 records by default.(id's 0 to 39)
- A file can have maximum of 955000 pages.
- File names have the format: "*data_type_name*.fl".
- Index file names have the format: "*data_type_name*_index.fl".
- There is one index file for a data type.
- Primary key is only one field.
- Name of the system catalog file is "sys.cat".
- A sys.cat page is created with 7 records by default.
- Deleted pages are handled by garbage collector, which runs periodically.

3 Storage Structures

3.1 System Catalog

System catalog stores metadata. Since this is also a file, it consists of pages and records. A system catalog record is 59 bytes.

- Records:
  - Header
    - Data type name (10 Bytes)
    - Number of fields (1 Byte)
  - Field names (8 Bytes $\times$ 6 = 48 Bytes)

Type1 Name	# of fields	Field1 Name	...	Field6 Name
Type2 Name	# of fields	Field1 Name	...	Field6 Name
...				
TypeN Name	# of fields	Field1 Name	...	Field6 Name

3.2 Index File

Index file keeps address of a record (file name, page id and record id). The purpose of this is to improve the performance. I used dense indexing in my design and all records are sorted according to primary key field.

Key Field(4 Bytes)	File Name (10 Bytes)	Page ID (4 Bytes)	Record ID (1 Byte)

Table 1: A record of index file

3.3 Files

Files store the main data. Each file stores only one type of data and it is specified in the header. A file can be maximum of 1GB so it can store 635000 pages at maximum. When a file is full, a new file is created. The next file pointer in the file header is set to this new file. Also isEmpty shows if the file is full, empty, deleted or contains some pages in it.

- Header (22 Bytes)
  - Previous File Name (10 Bytes)
  - Page Number (1 Bytes)
  - isFull (1 Byte): : 00000001 if page is empty, 00000000 if the page is full, 00000010 if the page contains some records but not full, 00000011 if it is deleted.
  - Next File Name (10 Bytes)

- Pages

Page ID	isEmpty	Number Of Records	Next Empty Place
... Records ...			
Page ID	isEmpty	Number Of Records	Next Empty Place
... Records ...			

Table 2: File Structure. Highlighted part is file header.

3.3.1 Pages

Files are divided into pages because data is transferred to the main memory in pages. There are maximum 40 records in a page and it is maximum 1047 Bytes.

- Header (7 Bytes)
  - Page ID (4 Bytes)
  - isEmpty (1 Byte): 00000001 if page is empty, 00000000 if the page is full, 00000010 if the page contains some records but not full, 00000011 if it is deleted.
  - Number of records (1 Byte)
  - Next empty record's offset from header (1 Byte)
- Records (26 Bytes $\times$ 40 = 1040 Bytes)

3.3.2 Records

- Header (2 Bytes)
  - Record id (1 Byte) (Record id's are between 1 and 40.)
  - isEmpty (1 Byte): 00000001 if record is empty, 00000000 if the record is not empty.
- Fields (4 Bytes $\times$ 6 = 24 Bytes)

Page id	isEmpty	# of record	Next empty record	
Record id 1	isEmpty	Field1	...	Field6
...				
Record id 40	isEmpty	Field1	...	Field6

Table 3: Page Structure. Gray highlighted row is the page header and blue highlighted cells are the record header.

4 Operations

4.1 DDL Operations

4.1.1 Create a Type

Algorithm 1 Create a Type

```
1: typeName ← User Input
2: numOfFields ← User Input
3: fields ← User Input
4: file ← open "syscat"
5: file.writeToEnd(typeName.toBinary(size=10, fillWithSpaces=True)
6: file.writeToEnd(numOfFields.toBinary(size=1)
7: for each field in fields do
       file.writeToEnd(field.toBinary(size=8, fillWithSpaces=True)
8: end for
9: create new file "typename.fl"
10: push(typeName, pageNum=0, isFull=1, nextFileName =" ") to the header
    of "typename.fl"
11: create new file "typename_index.fl"
```

4.1.2 Delete a Type

Algorithm 2 Delete a Type

```
1: typeName ← User Input
2: file ← open "sys.cat"
3: for each record in file do
4:     if typeName = record.Name then
           file.delete(record)
5:     end if
6: end for
7: delete file "typename.fl" // Recursively deletes all the files that this file
   points to
8: delete file "typename_index.fl"
```

4.1.3 List All Types

Algorithm 3 List All Types

```
1: file ← open "sys.cat"
2: list = new List()
3: for each record in file do
4:     list.append(record.name)
5: end for
6: list.sort(ascending=True)
7: print(list)
```

4.2 DML

4.2.1 Create a Record

Algorithm 4 Create a Record

```
1: typeName ← User Input
2: dataFile ← open "typeName.fl"
3: indexFile ← open "typeName_index.fl"
4: fieldNo ← read number of fields from system catalog "sys.cat"
5: while dataFile.isEmpty = 0 do
6:    if dataFile.nextFile exists then
7:       dataFile ← dataFile.nextFile
8:    else
9:       create a new file "typeName_fileId.fl"
10:      dataFile.nextFile ← this newly created file
11:      dataFile ← dataFile.nextFile
12:   end if
13: end while
14: currentPage ← get pointer to first page from dataFile header
15: while currentPage.isEmpty = 0 do
16:   if currentPage.nextPage exists then
17:      currentPage ← currentPage.nextPage
18:   else if file.numOfPages < 635000 then
19:      create a new page
20:      increment the numOfPages in the file header by 1
21:      currentPage.nextPage ← this newly created page
22:      currentPage ← this newly created page
23:      if file.numOfPages = 635000 then
24:         file.isEmpty ← 0
25:      else
26:         file.isEmpty ← 2
27:      end if
28:   end if
29: end while
30: record ← currentPage.nextEmptyRecord
31: for i ← 0 to fieldNo do
32:   record[i] ← User Input
33: end for
34: record.isEmpty ← 0
35: currentPage.numOfRecords ← currentPage.numOfRecords +1 // Increment
    the number of records of that page by 1.
```

```
36: if currentPage.numOfRecords = 40 then
37:    currentPage.isEmpty ← 0 // Page is full
38: else
39:    currentPage.isEmpty ← 2 // Page contains some records but not full
40: end if
41: for i ← record.id to 40 do
42:    if record with id i isEmpty = 1 then
43:       currentPage.nextEmptyRecord ← record with id i
44:       break
45:    end if
46: end for
47: find the appropriate place for the record in indexFile(sorted according to
    primary key)
48: push primary key, dataFile.id, currentPage.id and record.id to indexFile
```

4.2.2 Delete a Record

Algorithm 5 Delete a Record

```
1: typeName ← User Input
2: keyField ← User Input
3: indexFile ← open "typeName_index.fl"
4: binary search indexFile to find keyField
5: dataFile ← open typeName file with the file id that is found
6: record ← the record, whose adress is found in indexFile, in dataFile
7: delete keyField, pageId, recordId from indexFile and set isEmpty to 1
8: delete each field of the record
9: record.isEmpty ← 1
10: page.numOfRecords ← page.numOfRecords - 1 // Decrement the number
    of records in the page header by 1
11: if page.numOfRecords = 0 then
12:    page.isEmpty ← 3 // Page is deleted
13:    page.nextEmptyRecord ← 1 // Set the next empty record to the first
       record of the page
14:    decrement the number of pages in the file header by 1
15:    if dataFile.numOfPages = 0 and dataFile name ≠ "typeName.fl" then
16:       dataFile.isEmpty ← 3
          // If this is an extra file (not the one which is created by default when
          the type is created), it needs to be deleted. Garbage collector handles
          the rest.
17:    else
18:       dataFile.isEmpty ← 2
19:    end if
20: else
21:    page.isEmpty ← 2
22:    if page.nextEmptyRecord > record.id then
23:       page.nextEmptyRecord ← record.id // if the next empty record is a
          record after this one, set this record to be the next empty record
24:    end if
25: end if
```

4.2.3 Search a Record

Algorithm 6 Search a Record

```
1: typeName ← User Input
2: key ← User Input
3: indexFile ← open "typeName_index.fl"
4: search indexFile to find keyField
5: dataFile ← open typeName file with the file id that is found
6: record ← the record, whose adress is found in indexFile, in dataFile
7: return record
```

4.2.4 List All Records

Algorithm 7 List All Records

```
1: typeName ← User Input
2: dataFile ← open "typeName.fl"
3: list = []
4: while true do
5:    for dataFile.isEmpty ≠ 1 or 3 and each page in dataFile do
6:       for page isEmpty ≠ 1 or 3 and each record in page do
7:          if record.isEmpty = 0 then
8:             list.append(record)
9:          end if
10:      end for
11:   end for
12:   if dataFile has nextFile then
13:      dataFile ← dataFile.nextFile
14:   else
15:      break
16:   end if
17: end while
18: list.sort(sortBy=record.key, ascending=True)
19: print(list)
```

4.2.5 Update A Record

Algorithm 8 Update A Record

```
1: typeName ← User Input
2: key ← User Input
3: fields ← User Input
4: indexFile ← open "typeName_index.fl"
5: search indexFile to find key
6: dataFile ← open typeName file with the file id that is found
7: record ← the record, whose adress is found in indexFile, in dataFile
8: i = 0
9: for field in fields do
10:    record.fields[i] = field
11:    i++
12: end for
```

5 Conclusions & Assessment

In my design, I used primary dense indexing for performance issues. It improves search performance with the cost of doing more operations at insertions and deletions. However, I didn't use binary search for searching a record in a index file. It could have been better if I used it. Also, this design is mostly appropriate for heavily search usage. Updating a value of a record is also faster since it is based on search. Since the main data files are not sorted or hashed, pages are used efficiently. The space of a deleted record is preferred at insertion, if it is possible. This prevents creating redundant pages.

Each type is stored in different files. A data file can have maximum of 100 pages. When it gets full, a new file is created to store the new records and next file pointer points to this record. So by starting from the file with name "$typeName$.fl", we can access all files. Thus, the names of the other files are not important. This makes implementation easier. Also index file stores the id of the file so it is accessed directly when searching.

Since the field sizes are fixed, this causes inefficiency in terms of storage. Designing a storage manager with variable length fields is possible. An extra "size" field needs to be added to the system catalog for each of the fields. However, this makes implementation more complex.

In conclusion, every design has ups and downs so my design is open to improvement.