CmpE 321 Spring - 2020 Project 1 Report

Ibrahim Ozgurcan Oztas
2016400198

March 31, 2020

Contents

Chapter 1

Introduction

This report is constructed for the documentation of a Database Management System(DBMS) that is developed by my efforts. In this report, a user can find simple yet concise explanations regarding the core structure of the database.

This DBMS has been called "The Mountain" by its creator, Ibrahim Ozgurcan Oztas. The main purpose behind this project is to design a DBMS that has been requested by his instructor at course CmpE 321, Introduction to Database Systems.

The construction process includes several parts such as "Observation", "Innovation" and "Origination".

In "Observation", I've gathered several insights from currently used Database Management Systems such as MySQL and PostgreSQL which are both relational databases and the core of these databases depends on Object-Relation Model(ORM). In this project, "ORM" is the sole approach to define and use databases.

In "Innovation", I've imagined the possibility: What if there's another place in control of DBMS which holds data that's been loaded to RAM? It would bring a minor increase in performance that may be enough for a few cases, where amount of data is so huge that the search algorithm takes very long time even though the new search would retrieve a record that's been inside the same page with a previously retrieved one.

In "Origination", I've installed a medium layer that works as a cache for the each database. This approach will handle the cost of retrieving a record in the same page of another record that is already been retrieved. That brings out the capability of more queries done in the same amount of time.

In next chapter, one can find the fact that this project has several assumptions to ease out the implementations and a few constraints either given by the instructor, or the revelations of realistic approach.

In further chapters, one can find the storage structures, DDL and DML operations, and the conclusion & assessment of the project.

Hope one has a wonderful time while reading!

Chapter 2

Assumptions & Constraints

For this project, the assumptions that are listed below are considered.

- **For Records:**
  - **Number of features in a record**, F_T, can be either 8 or 9 or 10. It is decided for the inserted type, when a new type is inserted in system catalogue.
  - **Data type** stored in features in a record is integer(4 bytes).
  - **Record header** includes **12 byte** character array that represents the latest changed time in format (YYYYMMDDHHmm).
  - **Record size** is calculated according the following equation: $B_{record} = B_{recordheader} + F_T \cdot B_{feature}$, where $F_T \in \{8, 9, 10\}$.
  - Since F_T can take 3 values, there can be **3 types of records** that consumes **44**, **48**, and **52** bytes, respectively.

- **For Pages:**
  - **Number of records in a page**, N_P, is decided constantly for all unique record type, which is **32 for all types**.
  - **Page Header** is same for all three unique types, that holds **4 different information**, explained below:
    - **Available record indicator**, A_R, which holds whether i^{th} record is valid or not, for $i = 1, 2, 3, ..., 32$. Hence, for each record I've allocated 1 bit to validate, **4 bytes** of storage are allocated for this information.
    - **Binary page fullness density**, b_f, which holds the data that corresponds whether this page is **completely full** or not. Hence, **1 byte** of information is reserved in page header for this information.
    - **Binary page emptiness density**, b_e, which holds the data that corresponds whether this page is **completely empty** or not. Hence, **1 byte** of information is reserved in page header for this information.
    - **Number of features in records**, F_T, which holds the type of records that has either 8 or 9 or 10 features. **Type A records** hold **8** features, **Type B records** hold **9** features and **Type C records** hold **10** features. For **"Type A"**, character **'A'**, for **"Type B"**, character **'B'**, and for **"Type C"**, character **'C'** are used.
    - **Indicator of contamination**, b_c, which holds the information about whether this page should be **completely purged (deleted)** or not. **1 byte** is sufficient for this information.
    - **Total amount of bytes** consumed by page header:
      $\mathbf{4 + 1 + 1 + 1 + 1 = 8}$
  - The **logical position(priority)** of page header information:
    $b_f b_e F_T b_c A_R$
  - **Page size** is calculated according the following equation:
    $B_{page} = B_{pageheader} + N_P \cdot B_{record}$, where B_{record} can take either **44** or **48** or **52**, and N_P is **32**, defined above.
    A page may contain **1416** or **1544** or **1672** bytes, according the record type that it holds.

- **For Files:**
  - **A file can hold** up to **100 pages**, at **maximum**.
  - **Size of file** may **differ** due to different types of records.
  - **File Header** includes **4 kind of information**, explained below:
    - **Number of pages in file**, N_Q, is the **amount of page** that the file currently stores. **3 bytes** of information is suffice to hold this information.
    - **Last accessed page**, L_Q is **the page** that has been **requested** by the user **most recently**. **3 bytes** of information is suffice to hold this information.
    - **Number of cached pages**, C_Q, is the cached pages that resides in medium layer temporarily. It can be **minimum 1**, and **maximum 5**. It is decided when a new type is registered in system catalogue. **1 byte** of information is suffice to hold this information.
    - **Binary caching indicator**, b_Q, is the information **whether caching is enabled** for this type or not. It is decided when a new type is registered in system catalogue. **1 byte** of information is suffice to hold this information.
    - **Total amount of bytes** consumed by file header:
      $\mathbf{3 + 3 + 1 + 1 = 8}$
  - **File size** is calculated according to the following equation:
    $B_{file} = B_{fileheader} + N_Q \cdot B_{page}$, where N_Q is the current number of pages in file and B_{page} is the byte amount that one page consumes.
    At maximum values, the files may contain **141.608(141.6KB)** or **154.408(154KB)** or **167.208(167.2KB)** bytes, respectively.
  - After that, **if necessary**, **another file may be created** for the same type.
  - **The naming convention of files** is :
    $< type\_name > - < file\_number > .db$, where $< file\_number >$ can take values from 0001_{10} to 1000_{10}.

- **For System Catalog:**
  - **The system catalog** can hold at most 20 types.
  - **Each type** can have **at least 8, at most 10** type features.
  - **Each type name** is described with **a string which is at most 20 characters**.
  - **Each type feature** is identified with **a string which is at most 20 characters**.
  - **Each type** has a **binary caching indicator,** b_Q, to hold the information of whether the caching system is enabled or not. 1 byte of space is reserved to hold this information.
  - **Each type** has a **number of cached pages,** C_Q, to hold the number of pages in medium layer, described in introduction part. 1 byte of space is reserved to hold this information.
  - For a **single type**, the **required byte amount** is calculated according the following equation:
    $BYTE_{type} = BYTE_{type\_name} + F_{type} \cdot BYTE_{feature\_name} + BYTE_{C_Q} + BYTE_{b_Q}$
  - For **the complete system catalog**, the **overall byte requirement** is calculated according the following equation:
    $BYTE_{catalog} = 20 \cdot BYTE_{type}$

Chapter 3

Storage Structures

System Catalogue: In this part, I've explained the overall structure of the system catalogue and its properties.

- In system catalogue, I've collected several data from the user, such as **number of cached pages** and **binary caching indicator** to decide it is enabled or disabled for that type, in addition to the names of the features of a type.

System Catalogue Table:

$< type\_name >$	F_{n1}	F_{n2}	F_{n3}	F_{n4}	.	.	.	F_{n8}	F_{n9}	F_{n10}	C_Q	b_Q
$< type\_name >$	F_{n1}	F_{n2}	F_{n3}	F_{n4}	.	.	.	F_{n8}	F_{n9}	F_{n10}	C_Q	b_Q
$< type\_name >$	F_{n1}	F_{n2}	F_{n3}	F_{n4}	.	.	.	F_{n8}	F_{n9}	F_{n10}	C_Q	b_Q
$< type\_name >$	F_{n1}	F_{n2}	F_{n3}	F_{n4}	.	.	.	F_{n8}	F_{n9}	F_{n10}	C_Q	b_Q

- The design of system catalogue is illustrated above to clarify the structure for all to understand. For each type, user has been abled to select different number of unique feature among 3 possibilities, 8, 9 or 10.
- The **first feature for each type** in system catalogue is decided to be the **primary key** of that type, in addition to the fact that **every value** of a feature of a record must be **integer**. Users are kindly reminded of that information whenever a new type is added to the system catalogue.
- At most **20** type can be added to the system catalogue.

Page Design: In this part, I've explained the overall structure of the page design and its properties.

- In page design, I've imagined the fact that **pages** are constructed as **a stream of data** that holds a batch of records of a requested type. Also, pages are composed of **2 parts**, namely **page header** and **page data**.

Page Header Part of a Type:

b_f	b_e	F_T	A_R

Page Data Part of a Type:

H_1	$F_{n1,1}$	$F_{n1,2}$	$F_{n1,3}$	$F_{n1,4}$	$F_{n1,5}$	$F_{n1,6}$	$F_{n1,7}$	$F_{n1,8}$	$F_{n1,9}$	$F_{n1,10}$
H_2	$F_{n2,1}$	$F_{n2,2}$	$F_{n2,3}$	$F_{n2,4}$	$F_{n2,5}$	$F_{n2,6}$	$F_{n2,7}$	$F_{n2,8}$	$F_{n2,9}$	$F_{n2,10}$
H_3	$F_{n3,1}$	$F_{n3,2}$	$F_{n3,3}$	$F_{n3,4}$	$F_{n3,5}$	$F_{n3,6}$	$F_{n3,7}$	$F_{n3,8}$	$F_{n3,9}$	$F_{n3,10}$
H_4	$F_{n4,1}$	$F_{n4,2}$	$F_{n4,3}$	$F_{n4,4}$	$F_{n4,5}$	$F_{n4,6}$	$F_{n4,7}$	$F_{n4,8}$	$F_{n4,9}$	$F_{n4,10}$
H_5	$F_{n5,1}$	$F_{n5,2}$	$F_{n5,3}$	$F_{n5,4}$	$F_{n5,5}$	$F_{n5,6}$	$F_{n5,7}$	$F_{n5,8}$	$F_{n5,9}$	$F_{n5,10}$
H_6	$F_{n6,1}$	$F_{n6,2}$	$F_{n6,3}$	$F_{n6,4}$	$F_{n6,5}$	$F_{n6,6}$	$F_{n6,7}$	$F_{n6,8}$	$F_{n6,9}$	$F_{n6,10}$
H_7	$F_{n7,1}$	$F_{n7,2}$	$F_{n7,3}$	$F_{n7,4}$	$F_{n7,5}$	$F_{n7,6}$	$F_{n7,7}$	$F_{n7,8}$	$F_{n7,9}$	$F_{n7,10}$
H_8	$F_{n8,1}$	$F_{n8,2}$	$F_{n8,3}$	$F_{n8,4}$	$F_{n8,5}$	$F_{n8,6}$	$F_{n8,7}$	$F_{n8,8}$	$F_{n8,9}$	$F_{n8,10}$

Record Design: As shown above, a record is composed of 2 parts, **record header** and **record data**.

Record header only includes the information of last changed time in format **YYYYMMDDHHmm**, where **YYYY** represents year, **MM** represents month, **DD** represents day, **HH** represents hour in **24 hour format**, **mm** represents minute.

Chapter 4

DDL & DML Operations

Database Definition Language Operations: In this part, I've defined several operations that has effects on the system catalogue. You are here to find three distinct operation to define the system catalogue of the DBMS.

- **Create a Type:** Creating a Type includes an insertion operation into the system catalogue file, with several unique inputs, such as type features, binary caching indicator, and number of cached pages.

  If user tries to create a type which registered before, then the DBMS returns an error message indicating that the user had already registered this type.

- **Delete a Type:** Deleting a Type includes an removal operation from the system catalogue file, without any distinct input from user. The selected type is removed with its current records, without asking any permission.

  If user tries to remove a type which does not exist in the system catalogue, then the DBMS returns an error message indicating that there's no such type to be removed.

- **List all Types:** Listing all Types includes an itemization operation onto the system catalogue, without any input from user. All of the types are listed onto command prompt, or graphical user interface if enabled.

Algorithm 1 Create a Type

```
1: new_type ← readInput()
2: type_list ← L_T
3: type_counter ← length(L_T)
4: for i ← 0 to type_counter do
5:         if type_list[i]['type_name'] == new_type['type_name'] then
6:                 return TypeCreationError
7:         else then
8:                 continue
9: join(type_list, new_type)
10: return type_list
```

- In the algorithm explained above, L_T is the list of types in system catalogue, and for simplicity in algorithm, $readInput()$ and $length(K)$ functions are explained in this part.
  - $readInput()$, is the function that reads from the user via command prompt. When creating a new type, user enter all necessary information constrained under the statements in chapter 2, which implies that all features of the type, the binary caching indicator and if necessary, the number of cached pages for the type, are required to create a new type.
  - $length(K)$, is the function returns the number of items in list K.

Algorithm 2 Delete a Type

```
1: deleted_type_name ← readInput()
2: type_list ← L_T
3: type_counter ← length(L_T)
4: for i ← 0 to type_counter do
5:         if type_list[i]['type_name'] == deleted_type_name then
6:                 remove(type_list, deleted_type_name)
7:                 removeData(deleted_type_name)
8:                 return type_list[i]
9:         else then
10:                continue
11: return TypeExistenceError
```

- In the algorithm explained above, L_T is the list of types in system catalogue, and for simplicity in algorithm, $readInput()$, $removeData(T)$, and $length(K)$ functions are explained in this part.
  - $readInput()$, is the function that reads from the user via command prompt. When deleting a type, user enters the name of the deleted type only. All records of deleted type and type itself are deleted, after input read.
  - $removeData(T)$, is the function that removes all files regarding to type T from the operating system. Any trace that implies type T is removed, after the function called.
  - $length(K)$, is the function returns the number of items in list K.

Algorithm 3 List all Types

1: $type\_list \leftarrow L_T$
2: $type\_counter \leftarrow length(L_T)$
3: $for\ i \leftarrow 0\ to\ type\_counter\ do$
4: $\quad current\_type \leftarrow type\_list[i]$
5: $\quad feature\_counter \leftarrow length(current\_type)$
6: $\quad displayLine(current\_type[0])$
7: $\quad for\ j \leftarrow 1\ to\ feature\_counter\ do$
8: $\quad\quad displayLine(current\_type[0] + current\_type[j])$
9: $return\ type\_list$

- In the algorithm explained above, L_T is the list of types in system catalogue, and for simplicity in algorithm, $length(K)$ function is explained in this part.
  - $length(K)$, is the function returns the number of items in list K.
  - $displayLine(Q)$, is the function which prints out the given parameter to the console output.
  - $displayLine(P + R)$, is the function which prints out the given parameters joined with 8 space characters to the console output.

Database Manipulation Language Operations: In this part, I've defined several operations that has effects on the files of the types that are listed in the system catalogue. You are here to find five distinct operations to manipulate the files regarding to the system catalogue.

- **Create a Record:** Creating a record includes an insertion operation to the current file of the specified type. While adding operation is being performed, several conditions are evaluated to progress further. One of them is whether there's a record that has the same primary key with the new record. To eliminate that problem, I've decided to make primary key auto-incremental for each type. With that, there'll be no duplicates in data files. In addition to that, all data must be sorted in ascending order, by the given constraints in project description. Hence, I've devised a sorting algorithm that handles the sorting process after insertion. Yet, it has brought severe burden to the DBMS due to the sorting condition after record creation operation. All details regarding the sorting algorithm are explained in further parts of this chapter.

- **Delete a Record:** Deleting a record includes an removal operation of a record from the current file of the specified type. While removing operation is being performed, it has to be known for sure that a non-existing record can not be deleted. Hence, there's a validation along with the process to be sure that the desired data can be deleted.

- **Search a Record(by primary key):** Searching a record via its primary key includes many condition operations to find out exact record that is requested to be listed. Since it is known for sure that the data will be hold in ascending sorted manner, a linear search would be suffice to reach the exact record.

- **Update a Record(by primary key):** Updating a record via its primary key includes many condition operations to find out exact record that is requested to be patched. It is assured that input validation for the updated record is necessary for all records in the specified file of that type.

- **List all Records of a type:** Listing all records of a type includes a complete walkthrough from first record to be read until last record is read completely. It is known for sure that listing all records result in reaching all records in that type costs linear complexity to the DBMS, hence it is coverable even if there's massive number of record registered in DBMS.

Algorithm 4 Create a Record:

```
1: user_input_array ← readInput()
2: file_name ← findLastFile(T)
3: pos_file ← openFile(file_name)
4: header_byte_array ← runOverBytes(position_file, 8)
5: total_page_count ← byteToInteger(header_byte_array[0 : 3])
6: file_type ← typeFinder(header_byte_array)
7: page_size ← pageSizeFinder(file_type)
8: record_size ← recordSizeFinder(file_type)
9: for i ← 0 to total_page_count do
10:     pos_file ← current_page_position + page_size
11:     current_file_position ← pos_file
12:     page_header_array ← runOverBytes(pos_file, 8)
13:     is_page_full ← byteToInteger(page_header_array[0 : 1])
14:     if is_page_full then
15:         continue
16:     else then
17:         first_empty_record ← bitChecker(page_header_array[4 : 8])
18:         pos_file ← pos_file + first_empty_record * record_size
19:         added_record ← oWB(pos_file, record_size, user_input_array)
20:         return added_record
21: current_file_position ← current_file_position + page_size
22: if total_page_count == 100 then
23:     added_file ← addNewFile(T)
24:     pos_file ← openFile(added_file)
25:     added_record ← oWB(pos_file, record_size, user_input_array)
26:     return added_record
27: else then
28:     allocated_bytes ← memAlloc(current_file_position, page_size)
29:     pos_file ← current_file_position + 8
30:     added_record ← oWB(pos_file, record_size, user_input_array)
31:     return added_record
```

- $readInput()$, is a function which includes inquiring information from user, that returns a list of strings.
- $findLastFile(T)$, is a function which includes returning the file that is the last storage of that type.
- $openFile(file\_name)$, is a function which includes returning the byte position of the file named as $file\_name$.
- $runOverBytes(position\_file, byte\_number)$, is a function which includes scanning over $byte\_number$ of bytes from starting point $position\_file$, and then returning the scanned byte(s) as a byte array.
- $byteToInteger(byte\_array)$, is a function which includes returning the integer value of a $byte\_array$.
- $typeFinder(header\_byte\_array)$, is a function which includes returning the type information embedded inside $header\_byte\_array$.
- $pageSizeFinder(file\_type)$, is a function which includes returning the page size of that specific type of record, given as $file\_type$.
- $recordSizeFinder(file\_type)$, is a function which includes returning the record size of that specific type of record, given as $file\_type$.
- $bitChecker(page\_header\_array)$, is a function which includes checking bit expansion of the given byte array, then returning the index of first zero bit starting from rightmost bit.
- $oWB(pos\_file, record\_size, user\_input\_array)$, is a function which includes replacing bytes in size of $record\_size$, from starting pos_file with $user\_input\_array$ data transformed into byte values, then changing the value of the bit in page header part for the newly added record. Then, it returns the byte array of newly added record.
- $addNewFile(T)$, is a function which includes creating a new file for storing records for type T, then returning its name.
- $memAlloc(current\_file\_position, page_size)$, is a function which includes extending the size of current file, starting from $current\_file\_position$ and extending it by an amount of $page\_size$, then returning the array of bytes.

Algorithm 5 Delete a Record:

1: $deleted\_record\_pkey \leftarrow readInput()$
2: $file\_list \leftarrow findAllFiles(T)$
3: $record\_size \leftarrow findRecordSize(T)$
4: $record\_file\_quantity \leftarrow 3200$
5: $file\_number \leftarrow deleted\_record\_pkey/3200$
6: $req\_file \leftarrow file\_list[file\_number]$
7: $page\_number \leftarrow (deleted\_record\_pkey - 3200 * file\_number)/32$
8: $page\_size \leftarrow findPageSize(T)$
9: $pos\_file \leftarrow openFile(file\_list[file\_number])$
10: $pos\_file \leftarrow pos\_file + page\_size * page\_number$
11: $page\_header\_array \leftarrow runOverBytes(pos\_file, 8)$
12: $record\_order \leftarrow modulo(deleted\_record\_pkey - 1, 32)$
13: $pos\_file \leftarrow pos\_file + record\_order * record\_size$
14: $deleted\_record \leftarrow oWB(pos\_file, record\_size, NULL)$
15: $return\ \ deleted\_record$

- $readInput()$, is a function which includes inquiring information from user, that returns a list of strings.
- $findAllFiles(T)$, is a function which includes collecting the names of all files for type T into a list, and returning the list.
- $findRecordSize(T)$, is a function which includes returning the size of a record given in type T.
- $findPageSize(T)$, is a function which includes returning the size of a page given in type T.
- $openFile(file\_name)$, is a function which includes returning the byte position of the file named as $file\_name$.
- $runOverBytes(pos\_file, byte\_amount)$, is a function which includes returning the byte array starting from pos_file and iterates it over $byte\_amount$, then returning the byte array.
- $modulo(a, b)$, is a function which includes taking $a\%b$ and returns the resulting value.

- $oWB(pos\_file, record\_size, NULL)$, is a function which includes replacing bytes in size of $record\_size$, from starting pos_file with $NULL$ value transformed into byte values, then changing the value of the bit in page header part for the deleted record. Then, it returns the byte array of deleted record.

Algorithm 6 Search a Record(via primary key):

1: $record\_pkey \leftarrow readInput()$
2: $file\_list \leftarrow findAllFiles(T)$
3: $record\_size \leftarrow findRecordSize(T)$
4: $record\_file\_quantity \leftarrow 3200$
5: $file\_number \leftarrow record\_pkey/3200$
6: $req\_file \leftarrow file\_list[file\_number]$
7: $page\_number \leftarrow (record\_pkey - 3200 * file\_number)/32$
8: $page\_size \leftarrow findPageSize(T)$
9: $pos\_file \leftarrow openFile(file\_list[file\_number])$
10: $pos\_file \leftarrow pos\_file + page\_size * page\_number$
11: $page\_header\_array \leftarrow runOverBytes(pos\_file, 8)$
12: $record\_order \leftarrow modulo(record\_pkey - 1, 32)$
13: $pos\_file \leftarrow pos\_file + record\_order * record\_size$
14: $searched\_record \leftarrow runOverBytes(pos\_file, record\_size)$
15: $return\ \ searched\_record$

- $readInput()$, is a function which includes inquiring information from user, that returns a list of strings.
- $findAllFiles(T)$, is a function which includes collecting the names of all files for type T into a list, and returning the list.
- $findRecordSize(T)$, is a function which includes returning the size of a record given in type T.
- $findPageSize(T)$, is a function which includes returning the size of a page given in type T.
- $openFile(file\_name)$, is a function which includes returning the byte position of the file named as $file\_name$.

- $runOverBytes(pos\_file, byte\_amount)$, is a function which includes returning the byte array starting from pos_file and iterates it over $byte\_amount$, then returning the byte array.
- $modulo(a, b)$, is a function which includes taking $a\%b$ and returns the resulting value.

Algorithm 7 Update a Record(via primary key):

1: $updated\_record\_pkey \leftarrow readInput()$
2: $updated\_record\_input\_array \leftarrow readInput()$
3: $file\_list \leftarrow findAllFiles(T)$
4: $record\_size \leftarrow findRecordSize(T)$
5: $record\_file\_quantity \leftarrow 3200$
6: $file\_number \leftarrow updated\_record\_pkey/3200$
7: $req\_file \leftarrow file\_list[file\_number]$
8: $page\_number \leftarrow (updated\_record\_pkey - 3200 * file\_number)/32$
9: $page\_size \leftarrow findPageSize(T)$
10: $pos\_file \leftarrow openFile(file\_list[file\_number])$
11: $pos\_file \leftarrow pos\_file + page\_size * page\_number$
12: $page\_header\_array \leftarrow runOverBytes(pos\_file, 8)$
13: $record\_order \leftarrow modulo(updated\_record\_pkey - 1, 32)$
14: $pos\_file \leftarrow pos\_file + record\_order * record\_size$
15: $updated\_record \leftarrow oWB(pos\_file, record\_size, updated\_record\_input\_array)$
16: $return\ updated\_record$

- $readInput()$, is a function which includes inquiring information from user, that returns a list of strings.
- $findAllFiles(T)$, is a function which includes collecting the names of all files for type T into a list, and returning the list.
- $findRecordSize(T)$, is a function which includes returning the size of a record given in type T.
- $findPageSize(T)$, is a function which includes returning the size of a page given in type T.

- $openFile(file\_name)$, is a function which includes returning the byte position of the file named as $file\_name$.
- $runOverBytes(pos\_file, byte\_amount)$, is a function which includes returning the byte array starting from pos_file and iterates it over $byte\_amount$, then returning the byte array.
- $modulo(a, b)$, is a function which includes taking $a\%b$ and returns the resulting value.
- $oWB(pos\_file, record\_size, user\_input\_array)$, is a function which includes replacing bytes in size of $record\_size$, from starting pos_file with $user\_input\_array$ data transformed into byte values, then changing the value of the bit in page header part for the updated record. Then, it returns the byte array of updated record.

Algorithm 8 List all Records:

```
1: list_type ← readInput()
2: file_list ← findAllFiles(list_type)
3: record_size ← findRecordSize(list_type)
4: list_length ← length(file_list)
5: for i ← 0 to list_length do
6:     current_file ← file_list[i]
7:     current_pos_file ← openFile(current_file)
8:     file_header ← runOverBytes(current_pos_file, 8)
9:     page_amount ← byteToInteger(file_header[0 : 3])
10:    for j ← 0 to page_amount do
11:        current_record ← runOverBytes(current_pos_file, record_size)
12:        display(current_record)
```

- $readInput()$, is a function which includes inquiring information from user, that returns a list of strings.
- $findAllFiles(T)$, is a function which includes collecting the names of all files for type T into a list, and returning the list.
- $findRecordSize(T)$, is a function which includes returning the size of a record given in type T.

- $openFile(file\_name)$, is a function which includes returning the byte position of the file named as $file\_name$.
- $runOverBytes(pos\_file, byte\_amount)$, is a function which includes returning the byte array starting from pos_file and iterates it over $byte\_amount$, then returning the byte array.
- $byteToInteger(byte\_array)$, is a function which includes returning the integer value of a $byte\_array$.
- $display(current\_record)$, is a function which includes printing the value of $current\_record$ to the command prompt as a stream of strings.

Chapter 5

Conclusions & Assessment

In this design explained in former chapters, is a theoretical construction composed on the given assumptions & constraints, and the creator's imagination. Yet, it has several pros and cons in a trade-off manner. In this chapter, the further information is based on the advantages and disadvantages that this design has brought in reality.

Pros:

- Dynamic approach to the record types has brought different types of record metadata and it enhances the efficiency of storage for different needs.
- Overriding the default caching protocol via user has created a different supplying methods of records for different kind of demands by a user. If a user wants that there will be no caching, then it is for sure that for each fetching action processed on the database will take quite a time for a massive data chunk for queries requiring same data each time.
- Number of pages resides in cache is a great feature in my point of view, since it is known that caching depends on temporal locality and spatial locality. Increasing number of pages in RAM would likely increase the spatial locality and will have benevolent effects on overall efficiency of the DBMS.

- Sorted data in all types in DBMS has brought a quite load-off of searching a record in the database. For the usual approach, creating a record costs in complexity of $O(n^2)$, if insertion sort is used to create and sort the data. But in my approach, since page headers hold the information of each record place in every page whether a record is registered or unregistered to the database. Then, every action progresses in linear complexity $O(n)$, which is quite an efficiency.

Cons:

- Increased chance of bugs, since different features are included in the DBMS and while coding, it might brought up more bugs than a default project.
- Complexity of coding process is quite higher than a usual approach of designing a DBMS, since imitating the workflow of RAM for each database, defining a caching alongside with the dynamic record structure, and the algorithms behind all DDL & DML operations inside DBMS will cost quite a time to code, analyze and test than a regular project.