THIRD ASSIGNMENT: REPORT:

1: File Management

i) and ii)

In this part of the assignment, we have designed the table functions. We have implemented the *table_test.c* file the following way: we set random *int* values to *int* fields and in *strings* we have saved "Hola mundo." in each one. When we execute the exercise, we have in each row one *string* with "Hola mundo." and 3 random *integers*, so it works as it should.

iii)

In this part of the exercise, we have implemented more types that can be added to the database, we have implemented the *long long int* and the *double* type, so in our new table we can add new data types. This made us change a bit the types functions, but it was not very difficult.

Output of table_test.c:

```
$ ./table_test
number of columns: 4

Row: 1
    Value 1: 3  Value 2: 9223372036854775807  Value 3: Hola mundo.  Value 4: 0.886000
Row: 2
    Value 1: 7  Value 2: 9223372036854775807  Value 3: Hola mundo.  Value 4: 6.915000
Row: 3
    Value 1: 3  Value 2: 9223372036854775807  Value 3: Hola mundo.  Value 4: 8.335000
Row: 4
    Value 1: 6  Value 2: 9223372036854775807  Value 3: Hola mundo.  Value 4: 0.492000
Row: 5
    Value 1: 9  Value 2: 9223372036854775807  Value 3: Hola mundo.  Value 4: 1.421000
```

As we can see in each row, the first value is an *int*, the second one is a *long long int* (and also notice that that is the biggest number that fits in a *long long int*), the third value is a *string* (the one used before) and the last one is as we can see a real number stored in the C type *double*.

2: Using local data

In this part, we have implemented the two files we were asked:

- *score*, in which we receive a *title* and a *score*, and add it to a local binary table, as well as the *isbn* and *book_id* [1].

- *suggest*, in which we receive a *score*, and search through all the books in the local table and print the title and the author of those which received that same score.

[1] See final notes for explanations about the *book_id* field

In order to be able to use these functions, we also implemented *initialize_table.c*, in which a table is created with *table_create()*. This only prints the header of the table into the file, but is needed to write data in score and read it in suggest

Output analysis:

- Score:

Score doesn't print anything on the screen. It just adds a book to a file (book_table.txt), which has a lot of unprintable characters. In order to see if score worked properly, we used the command:

```
> hexdump -C "book_table.txt"
```

which returns the following:

```
$ hexdump -C "book_table.txt"
    00000000  04 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00  |................|
    00000010  00 00 00 00 21 00 00 00  38 34 39 30 36 32 38 33  |....!...84906283|
    00000020  34 33 00 45 6c 20 43 6c  75 62 20 44 75 6d 61 73  |43.El Club Dumas|
    00000030  00 5a 00 00 00 00 00 00  00 24 00 00 00 38 34 32  |.Z.......$...842|
    00000040  30 34 38 33 35 30 38 00  50 61 74 65 6e 74 65 20  |0483508.Patente |
    00000050  64 65 20 43 6f 72 73 6f  00 5a 00 00 00 00 00 00  |de Corso.Z......|
    00000060  00 20 00 00 00 39 30 32  38 34 32 36 31 35 39 00  |. ...9028426159.|
    00000070  53 63 68 61 64 75 77 74  61 6e 67 6f 00 64 00 00  |Schaduwtango.d..|
    00000080  00 00 00 00 00                                    |.....|
    00000085
```

With this we can check if the *string* values (*isbn* and *title*) are stored correctly. However, *int* fields cannot be checked with this function, which reads hexadecimal, so we modified *table_test* into *score_test*. Here we read everything from the file, and print it through the terminal. It returns, with this file:

```
$ ./score_test


Row: 1
    Value 1: 8490628343  Value 2: El Club Dumas  Value 3: 90  Value 4: 0
Row: 2
    Value 1: 8420483508  Value 2: Patente de Corso  Value 3: 90  Value 4: 0
Row: 3
    Value 1: 9028426159  Value 2: Schaduwtango  Value 3: 100  Value 4: 0
```

As we can see, all the fields are stored correctly[2]

- Suggest:

Suggest prints in the screen books with the score given, so with these three books scored, it returns:

```
$ ./suggest 75


$ ./suggest 90


Arturo Perez-Reverte   El Club Dumas

Arturo Perez-Reverte   Patente de Corso

$ ./suggest 100


Arturo Pérez-Reverte   Schaduwtango
```

We did not implement another file to check *suggest*, as it can be easily checked knowing which books have been scored and being able to see all the scores with *score_test*

3: More work

i)

To implement indexes, we chose the implementation in which each record has a different key, and the positions are stored in an array, instead of creating a new record for each time a key appears

We implemented *index_test*, in which we create an index, fill it with some randomly generated keys and positions (keys are generated between 1 and 4, and the position between 0 and 99), and read all of them to check if everything was been stored properly.

To compare the keys that are supposedly stored and the ones that are truly stored, we first print the pairs of keys and positions being inserted, and after all the process print all the positions of each key, as well as how many of them there are. So, when we execute it, in the terminal we see:

[2] See notes at the end for further explanation about the field *book_id*

```
$ ./index_test


Key: 4  Position: 86

Key: 2  Position: 15

Key: 2  Position: 35

Key: 3  Position: 92

Key: 2  Position: 21

Key: 3  Position: 27

Key: 3  Position: 59

Key: 4  Position: 26

Key: 1  Position: 26

Key: 1  Position: 36

Key: 4  Position: 68

Key: 4  Position: 29

Key: 3  Position: 30

Key: 3  Position: 23

Key: 4  Position: 35




Key: 1 (num_pos: 2)    26      36

Key: 2 (num_pos: 3)    15      21      35

Key: 3 (num_pos: 5)    23      27      30      59      92

Key: 4 (num_pos: 5)    26      29      35      68      86
```

With this output, we can visually check if all the keys and positions have been correctly stored.

ii) and iii)

In this part, we have modified the two files we were asked:

- *score_index*. The difference with *score* is that we also add the *score* as *key* and the position in which we write in the table to an index, in order to use it in *suggest_index*

- *suggest_index*. The difference with *suggest* is that, instead of looping through every book in the table, and check if its score is equal to the one given, we just ask the index for all the positions of books with that same score, and just loop through them. Inside the loop we do the same (asking ODBC for the author....)

In order to be able to use these functions, we implemented *initialize_index.c* (as well as *initialize_table.c* from the previous files), in which an index is created with *index_create()*.

This only prints the header of the index into the file, but is needed to write data in score and read it in suggest

Output analysis:

- Score_index:

Score_index, as *score* did, doesn't print anything on the screen. It just adds a book to a table (*book_table.txt*), and the position it has been written, as well as the score to an index.

To see if *score_test* worked properly, we used the same command we used with score:

```
> hexdump -C "book_table.txt"
```

which returns the following:

```
$ hexdump -C "book_table.txt"
00000000  04 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00  |................|
00000010  00 00 00 00 21 00 00 00  38 34 39 30 36 32 38 33  |....!...84906283|
00000020  34 33 00 45 6c 20 43 6c  75 62 20 44 75 6d 61 73  |43.El Club Dumas|
00000030  00 5a 00 00 00 00 00 00  00 24 00 00 00 38 34 32  |.Z.......$...842|
00000040  30 34 38 33 35 30 38 00  50 61 74 65 6e 74 65 20  |0483508.Patente |
00000050  64 65 20 43 6f 72 73 6f  00 5a 00 00 00 00 00 00  |de Corso.Z......|
00000060  00 20 00 00 00 39 30 32  38 34 32 36 31 35 39 00  |. ...9028426159.|
00000070  53 63 68 61 64 75 77 74  61 6e 67 6f 00 64 00 00  |Schaduwtango.d..|
00000080  00 00 00 00 00                                    |.....|
00000085


$ hexdump -C "book_index.txt"
00000000  00 00 00 00 02 00 00 00  5a 00 00 00 02 00 00 00  |........Z.......|
00000010  14 00 00 00 00 00 00 00  39 00 00 00 00 00 00 00  |........9.......|
00000020  64 00 00 00 01 00 00 00  61 00 00 00 00 00 00 00  |d.......a.......|
00000030
```

With this we can check if the *string* values (*isbn* and *title*) are stored correctly. However, *int* fields cannot be checked with this function, which reads *hexadecimal*, so we used the same *score_test* as previously. It returns, with this file:

```
$ ./score_test


Row: 1
    Value 1: 8490628343  Value 2: El Club Dumas  Value 3: 90  Value 4: 0
Row: 2
    Value 1: 8420483508  Value 2: Patente de Corso  Value 3: 90  Value 4: 0
Row: 3
    Value 1: 9028426159  Value 2: Schaduwtango  Value 3: 100  Value 4: 0
```

We can see that nothing has changed in the table, although we are now using indexes.[3]

- <u>Suggest_index:</u>

Suggest_index, as *suggest* did, prints in the screen books with the score given, so with these three books scored, it also returns:

```
$ ./suggest 75


$ ./suggest 90


Arturo Perez-Reverte   El Club Dumas
Arturo Perez-Reverte   Patente de Corso
$ ./suggest 100


Arturo Pérez-Reverte   Schaduwtango
```

We can see, with these checks, that although we are using indexes, the data itself is not modified. The only difference these files have in comparison to the previous version, is that they are more efficient when executed, because they just read what is needed, instead of the whole file

Finally: Write a report explaining your work, the problems that you have found and your solution. Have you implemented the indices as mentioned here, or have you found your own solution? Have you implemented the table the way I did, or have you found your own way (there are other ways... we'll talk about it in class)?

About the implementation of the index and table, we have used the implementation explained in the previous exercises.

[3] See notes at the end for further explanation about the field *book_id*

We had a lot of trouble implementing the table functions, because it was the first time we worked we binary files and we were not used to them. However, after a few hours we managed to master them, as well as the functions that work with them.

In the ODBC part we had a lot of trouble implementing the queries, so we used the function *sprintf* to write the query to a string, and we then had no more trouble.

In the *suggest* program, we already have the titles stored in our binary file, but, however, we get it again from the database, just because we think this method is more reliable that taking it from our own file, because we are not fully certain binary files work as expected.

The implementation of indexes was a bit tricky, because the functions were very difficult and with a truly long implementation. We faced a lot of troubles while implementing. The most significant and recurring was the *segmentation fault*. To fix them, we used the tool *Valgrind*. However, for many more complications we had, we had to use internet documentation to solve them.

Notes about *book_id*:

In the previous assignment, we did not use the *book_id*, and in this assignment, we have not either. This is because, as we cannot associate books with *book_ids* from the given files, the only solution was to implement is as a *serial* field. However, this would result in different values for every book. As we already have the *isbn* field as primary key (different for each entry), this would be meaningless.

As this was design choice, in both *score* and *score_index* we write the *book_id* field to 0, just to not completely remove a field from the table, because we would have to radically modify the table creation functions.