Pablo Cuesta & Álvaro Zamanillo. Group 1251. Pair 1.

PRACTICE II. REPORT

1. Introduction

This report will cover how the different parts of the practice have been developed and the reasons for the choices taken.

For this practice, we have had to create a C program to interact with a database. For that purpose, we have used several functions included in the ODBC library. In general, we can divide the practice in two parts: the code related for the display for the menus and the related with the ODBC library.

2. Menu implementation and code organization

In the first part of the practice we mainly followed the recommendations given in the essay: we used the same code structured from the menu.c example; that is to say, a general menu and one submenu for each block of queries (customers, products, and orders).

In an attempt to have a more organized program, we decided also to write all the query functions related with the different blocks in separated .c and .h files.

3. Query functions

As mentioned before each block of queries is in a different .c (and .h) files. However, the functions in all of them are quite similar and follow the structure described below. (Variations from this structure are explained below for each option).

1. First of all we made a connection to the database using the function provided in *odbc.c odbc_connect.*

2. Then, we prepared the queries after having called the function *SQLAllocHandle*. We used the function *SQLPrepare* to prevent all kinds of possible injections. At the beginning we did not use *SQLPrepare*; instead , we created the query manipulating strings which had the inherent risk of injection.

3. Because we used *SQLPrepare* we had to bind the parameters with the input requested to the user. To do that, we used *SQLBindParameter*, to fill in the question marks written in the query written in *SQLPrepare*.

4. After this last step, the query was ready to be executed so we ran it *(SQLExecute*) and got the results. Once again we had to bind the columns fetched with a variable, because the query result was more than a single row. The functions used in this part were *SQLBindColumn* (passing a pointer to variables so that they change every time we fetch each row of the result) and *SQLFetch.*

5. After obtaining the result, we just had to free the handle that had been allocated (*SQLFreeHandle)* and disconnect (*SQLDisconnect*).

This structure is common for all the query functions, the difference is the number of parameters asked to the user and the query result (number of columns and its type). Furthermore, in most of the queries we also used the function SQLDescribeCol so as to print the name of the column when printing the results in the terminal, as well as the error control function SQL_SUCCEDED.

3.1 Products Queries

ProductsStock: This first function follows step by step the structure mentioned before. The query executed is indeed quite simple as it just involves a single table and one condition.
The query is the following, we only need to take the *quantityinstock* field of the table *products* where the *productcode* correspond to what the user enters:

```
SELECT quantityinstock
FROM products
WHERE productcode = ?;
```

ProductsFind: The key part of this query was finding the appropriate format to prepare the query. This query involves the like operator and the string that follows it, was the input asked. When binding the parameter we could not just write *… like %?% …* because the symbols % and ? together were not recognised. The proper way was writing *… like '%'||?||'%' …* We added a flag (in the while loop where we fetch the rows of the result) to know if the query result was empty (we used this flag for the rest of the functions in which we could get multiple rows).
The query for this option consists also on using on using one table, *products*, and taking the *productcode* and *productname* fields from the rows where the string entered by the user appeared in the name of the product:

```
SELECT productcode, productname
FROM products
WHERE productname like '%'||?||'%';
```

3.2 Orders Queries

OrdersOpen: This query did not need the *SQLPrepare* function because we were not asking the user for any input (no risk of injection). For this reason, we only had to execute the query using *SQLExecDirect* (no need to prepare the query, bind the parameters and then execute). We then describe the columns, and bind the columns to variables in order to fetch and print the rows. As mentioned before, if the result was empty, we inform the user that all orders have been shipped.
For this query, we select the *ordernumber* field of the orders which have *NULL* in the *shippeddate* field, because that means they have not been shipped yet.
At first, we also put the condition: *and status<>'Cancelled'* , in the *WHERE* statement, so as not to show the orders that have been cancelled.However, because this was not contemplated in the tests (.sh files) provided, we did not include it in the final query:

```
SELECT ordernumber
FROM orders
WHERE shippeddate is NULL
ORDER by ordernumber;
```

OrdersRange: The most important aspect of this function is that we were selecting shippedate in the query. This attribute has not the constraint *NOT NULL* like most of the others. Because of this, we had to include an extra variable when binding this column. This extra variable indicates the length of the string fetched. If a row has *NULL* value for that attribute, this variable is assigned value -1 so we do not try to print the string but just a blank space. It is also worth mentioning the function *check_date* (checks whether a string has the format "YYYY-MM-DD"). We created this function just for the user to be noticed if they had entered a date in the wrong format so they could enter it again instead of trying to run the query and getting an error (reported by odbc_extracterror).
This query simply takes the orders in which the date is between the dates entered by the user and returns the *ordernumber, orderdate* and *shippeddate* of each:

```
SELECT ordernumber, orderdate, shippeddate
FROM orders
WHERE orderdate>=? and orderdate<=?
ORDER by ordernumber;
```

OrdersDetails: This is the only function where we run more than one query and to do so in a simple way, we declared, handled, executed, and freed: two statements. We doubted if it was worth creating a view because both queries need the same result table. However, we thought that the appropriate thing would be to delete the view after getting the results and therefore we would have to add a new statement to drop the view and, at the end, it could end up being slower than executing two independent queries. For getting the price of the order we use a nested query where the inner part is what we could have replaced with a view.

The first query (where we get the sum is the following): we only have to sum the price paid in a given order (*ordernumber* provided by the user), so we make a nested query where the price of each product in the entered order is calculated (*priceeach*quantityordered*) and then all of these prices are added to get the total amount.

```
SELECT Sum(price) AS sum
FROM   (SELECT ( priceeach * quantityordered ) AS price
        FROM   orderdetails
               natural JOIN orders
        WHERE  ordernumber = ?
        ORDER  BY orderlinenumber) AS x ;
```

The second query is exactly the same as the nested query in the first, but instead of selecting the price, it is necessary to select *status, orderdate, productcode, quantityordered* and *priceeach*.

```
SELECT status, orderdate, productcode, quantityordered,
priceeach
FROM   orderdetails natural JOIN orders
WHERE  ordernumber = ?
ORDER  BY orderlinenumber ;
```

3.3 Customers Queries

CustomersFind: For this query we had the same syntax issue from ProductFind and the like operator. Apart from that the query is again quite simple as it only involves one table:

```
SELECT customername, contactfirstname, contactlastname,
  customernumber
FROM   customers
WHERE  contactfirstname like '%'||?||'%' or contactlastname
like '%'||?||'%'
ORDER BY customernumber ;
```

CustomersListProducts: The key part of this query function is the paging which will be discussed in the next section. As for the query, it needed the function sum and therefore aggregation-*GROUP BY* (so that each product appears only once, with the total quantity ordered by the customer, even if it was in different orders):

```
SELECT productname,
sum(quantityordered) AS total_number_of_units
FROM orders natural join customers
     natural join orderdetails
natural join products
WHERE customernumber = ?
```

```
GROUP BY productcode, productname
ORDER BY productcode;
```

CustomersBalance: In this option, we do not use *SQLBindCol*, as there is only one possible row, with only one column. So we use the function *SQLGetData*, as well as the variable *col_ind* to check if the value is *NULL*, which means that the *customernumber* entered by the user is not in the database.
The query implemented consists of two nested queries, the first takes the total sum of items*price_of_each_item ordered by the customer (in non-cancelled orders), and the second takes the total amount of money already paid by that customer. Finally, the difference is returned:

```
SELECT ( y.paid - x.total ) AS Balance
FROM (SELECT Sum(priceeach * quantityordered) AS total
      FROM   (orders NATURAL JOIN orderdetails)
      WHERE  customernumber = ? and
             orders.status<>'Cancelled') AS x,
      (SELECT Sum(amount) AS paid
      FROM   payments
      WHERE  customernumber = ?) AS y;
```

4. Paging

For the paging we thought of two different approaches, but one, using files, was discarded soon.

The idea of the files was copying all the results of the query in a file. Then, for each time we wanted to print a page, open the file, go to the line where started that page and print ten rows. This would have been so inefficient because we would have been opening and closing the file constantly and after leaving the function, we would have had to remove the file. Having said this, it is clear that using files was not the right approach.

What we have used is an array of strings: we save one row per string. Because we cannot know the number of rows in the result of the query, we use realloc whenever we run out of space in the array of strings. As we had to implement the paging for two different queries (with different results), we generalized the code in two external functions *fetch_and_store* (that given a statement (*SQLSTMT*) already executed, returns an array of strings containing the rows of the result, or *NULL*, if there is no results) and *page_results* (where the paging of the results stored in the previously returned array is handled). It is worth mentioning the double pointer to void argument of *fetch_and_store* which we use to deal with the different types of the fields from each query result (in *CustomersFind* and *CustomersListProducts*).

Once we have all the rows saved in the array, we just acces them by blocks to print them page by page. In order to clean the screen and positionate the cursor, we have used several ANSI escape sequences. The characters ‘<’ and ‘>’ are requested to the user in order to return to the previous page or to go to the next one. The character ‘q’ is used to exit the results and go back to the Customers submenu. (These commands are explained in each ‘page’ so that it is clear to the user what they have to do to see the results or to go back).

NOTE: We have executed each option with valgrind, but all of the warnings and errors that it reports are coming from functions that we have not coded.