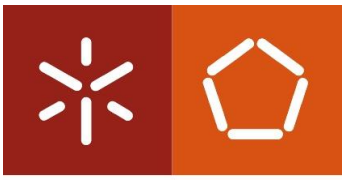

Universidade do Minho

Departamento de Informática

Sistema de Gestão de Vendas

Grupo Nº 39

Filipe Felício (A85593)

Henrique Ribeiro(A89582)

João Correia (A84414)

April 15, 2020

Contents

Chapter 1

Introduction

The following pages serve the purpose of in-depth reviewing the sophistication of our software implementation of a modular Sales Management System that is capable of processing sales data in a large scale, whilst also being able to quickly provide products and clients info, and answer multiple questions and challenges with optimal solutions.

With the key word on the last paragraph being *sophistication*, this report shall sustain the thesis of our choices in terms of Algorithms, Data Structures and Architecture being the most effective, reliable and appropriate, within each of the singular queries and their spot in the overall plain of the program, according to the fitting metrics of development.

In order to achieve peak performance in a multi-variable set of goals, such as time, scalability , encapsulation and memory use, this study will reflect the potential of our own approach to solve the given task with logical allocation of resources to give the best overall experience to a user of this application, given that on a micro dissection not all the choices were optimal we aim at the highest degree of optimization at the amortized macro analyses.

Chapter 2

Problem interpretation and challenges

The proposed project was to develop a sales management system (SGV) capable of emulating a mid sized business. There are three files that detail the emulated environment, they are:

- A file containing Clients' IDs;
- A file containing Products' IDs;
- A file describing sales information containing the ID of the bought product, the ID of the client who bough it, the price of an unit, the number of units bought, if the product was bought normally or through a sale and in which branch it was bought;

The system must also be able to answer 13 different queries, these are as follows:

1. Read the files referenced above, checking if the entries are valid and then process the information, loading it in it's data structures.
2. List all the products whose IDs start by a user provided character.
3. Given a product's ID and a month, return the number of sales and the total value billed for that product. The user must decide if he wants to see the results grouped globally or discriminated by branch
4. Return a list of products which were never bought, alphabetically ordered by their ID. The user can decide if he wants to check globally or by branch.
5. Return a list of clients who made purchases in all branches, also alphabetically ordered by their ID.
6. Determine the number of clients who never bought anything, as well as the number of products which were never bought.
7. Given a client's ID, create a table a table with the number of products bought by that client. The table must be divided by month and branch.
8. Determine the number and value of the sales that happened between a defined month interval.
9. Given a product's ID and a branch number, list the ID's of the clients who bought it, distinguishing them by if the client bought it normally or bought it at a discount.
10. Given a client's ID and a month, list the ID's of the products he bought more units. The list must be ordered by the number of units bought, in descending order.

11. List the N most sold products during the whole year. For each product, the number of sales and clients of that product in each branch must also be presented.

12. Given a client's ID, list the N products in which the client spent more money

13. Show a summary of what happened in query 1. This summary must include the path of each file read, as well as the number of lines read and the number of lines which were valid.

The main purpose of this project, as we understood it, was to learn the best practices of developing scalable mid-to-large scale software while keeping in mind crucial design principles, such as encapsulation and modularity.

One of the main challenges was the process of choosing the data structures, since it demanded a careful and methodical approach, as detailed on chapter 4.

Chapter 3

Project architecture and Modules

The project adopts the *MVC* (Model, View, Controller) architecture. A dependency diagram of our program can be found on attachment A.1. The details of each layer are exposed below.

3.1 Model

The model is the part of the software responsible for inserting, managing and manipulating data. In our program the modules that constitute the model are explained below, being the interface module the main module, since it is the one who encapsulates all the other ones.

3.1.1 Client

The Client module is responsible for holding information and answering simple questions regarding a single client, such as its id and if and where he has bought something.

3.1.2 Clients

This module is responsible for storing all the clients. The data structure used for this purpose was an hash table, provided by the GLib library. The reasons for why this was the chosen data structure are later presented on chapter 4.

3.1.3 Product

The product module holds information about a product, such as its id and where the product was bought. It has an API that is very similar to the Clients', being capable of answering simple questions.

3.1.4 Products

The module is responsible for storing all the products. Because it has been built with scalability in mind, as with the Clients module, it uses an Hash table for storing and retrieving data.

3.1.5 Sales

This module is responsible for holding the information about a product's billing and profits. After researching the problem, the need for creating a module representing a single sale wasn't found, so this module is the module directly responsible for answering information about the all the sales of a product, namely the number of times it has been sold, the total amount of units sold and the total value of sales of that product. This is done by using multi-dimensional arrays, using the indexes to separate information by month, branch and type of sale.

3.1.6 TotalSales

This is the module for storing all the sales structures. As with the products and clients modules, this one also uses an hash table. It will serve as a middleman for creating and updating sales information regarding a product whilst also providing functions to check if a sale is valid. No reference to clients is present in this module as that is responsibility of the branch module.

3.1.7 Branch

This module is responsible for storing the information about the sales made in a certain branch of our business. It is capable of establishing a relation between all the products a client bought in a certain branch and all the clients who bought a product. This is done by using plethora of structures who boil down to nested hash tables. The first layer is composed of two hash tables. The first one, for each product, contains a second hash table for each client who bought it. It then contains a structure with information about the client's purchases of that product in that branch. The same logic is applied to the second pair of nested hash tables, which establish a relation between every client and all the products they have bought.

3.1.8 Interface

This is the module where the SGV is implemented. This structure will encapsulate all modules referenced above: A products catalogue, a clients catalogue, a totalSales structure and an array of branches. This module also contains all the interactive queries, therefore it will interact with the controller when the user requests information from the application.

3.2 Controller

This is the layer responsible for interaction with the user. It calls the View layer to present the options that the user may call. After an option is chosen, the Controller will then communicate with the Model layer to either change the internal representation of the data or to provide the answer to the chosen query.

3.2.1 Controller

This module implements functions capable of navigating our intricate View layer whilst also being responsible for validating the user's input and handling the implemented queries, calling the Model layer and obtaining the QueryOutput structure, which will be used to present the query's answer.

3.3 View

This layer is responsible for presenting formatted information to the user, through the use of tables, diagrams and other graphics. It will also be responsible for presenting large amounts of data in a format the user can easily navigate.

3.3.1 View

The view module presents the application's menu to the user with some imaginative techniques to provide an immersive, easy-to-use experience. It will also present the user with the answers to all the queries in a navigable format, thus serving also as a String *paginator*.

3.4 Transversal modules

These are the modules who will be used by more than one of the above layers and, as such, cannot be labelled as one of them.

3.4.1 QueryOutputs

This module consists of a group of structures which will be used to return the answer's to the interactive queries. Although some queries could use the same structure to return their answers, we opted for implementing a different structure for each query. Doing so, it becomes easier to rewrite code in case we decide to change the way some query works or what information it will return.

3.4.2 ProductInfo

This module is used in query getTopSelledProducts() to keep track of a product's number of units bought and clients. Although it could be declared as an ancillary structure inside another module, for a reason of modularity and encapsulation, it was declared as different module altogether since it is called upon by several different modules.

3.4.3 Config

This module contains user defined macros for our program. The value of these macros can be modified to increase or decrease the application's dimension, making it possible to attain software scalability without having to know how the software is implemented. Some examples of these macros are: the size of a client's and product's id, the number of different types of sales and the number of branches.

Chapter 4

Data Structures

When designing mid to large scale software, one of the most crucial decisions consists on which data structures to use in order to strike a balance between memory usage, speed and scalability. As described on chapter 3, both the clients module and the products module implement the glib version of Hash tables. This choice was based in both analytical and empirical evidence, as shown below. Two data structures seemed promising for our software, they were:

- A Hash table
- An Array of 26 Balanced Trees

The program has two phases where the data structures are consulted or modified: When the information is loaded to the data structures, right at the beginning of the program, and when answering the proposed queries.

While the Hash table's insertion and search time complexity is $O(1)$, meaning it will have a stellar performance when inserting and validating information, it becomes less effective when sorting data, requiring the use of a sorting algorithm like the one used in the program, *qsort()*. The trees will be slower when inserting and validating data since their operations are $O(logn)$, but, due to their ordered insertion algorithm, will provide faster answers to queries where the answer must be ordered.

To test both data structures, the products module was implemented both using an hash table (*products.c*) and using an array of trees (`experimental_code/productsTrees`). To measure the impact, three times were measured: the load time, the time to answer query 2 and query 4, these are the processes that are most impacted by what data structure is implemented. The results can be found on tables C.1 and C.2.

As expected, the load time is faster and the second query is slower when using the Hash table, however, the fourth query is actually slower when using the tree array. This is due to a completely different reason than time complexities: the glib's API. The GHashTable API has a wider variety of functions when compared to the GTree API, which results in better optimizations when using functions such as `g_hash_table_iter` and `g_hash_table_get_keys` that don't have any equivalents in the GTree API. This is another reason we decided to use Hash tables: it provides a better API for scalable software. We also see that as the program grows, the load time increases accordingly while the queries' answer time remains the same. If scalability is in one's best interest, the use of hash tables comes as the most sensible choice, since it provides a better load time while sacrificing little in query time.

Chapter 5

Performance metrics

When developing software, there are a myriad of different metrics that can be used to evaluate its performance. In the context of this project, while having in mind the specification detailed on chapter 2, the aspects we found more relevant to measure were memory management and execution time, with a special emphasis on the first one, since our application is not aided by databases.

5.1 Memory management

As referenced above, the program does not use databases, which means that all the data must be loaded to memory. With this concern, we developed the application with the goal of being able to still fit large amounts of data. That means that we programmed our modules to make use of the heap, since the stack has limited size. While sacrificing some speed in the load process, this ensures our application is capable of handling bigger amounts of data.

One of the greatest risks associated with this kind of approach when programming in C is the threat of memory leaks. The problem with memory leaks is the fact that a program can run with these errors whilst appearing to execute normally and producing the expected results.

5.2 Memory usage

In order to check our program for memory leaks, the *Valgrind* tool was used. No memory leaks were found, except for 18,612 bytes which came Glib. These were completely unavoidable as the very own act of simply including the glib header file caused them.

Not only does *Valgrind* warn of memory leaks, it also informs the user of the specific parts of the software that cause these issues. This tool was very useful, especially when used alongside *GDB*, when it came to debbuging.

With the use of the *memusage* tool, we are able to verify the max amount of data allocated in the heap for each sales document read. From the B.1 graph, we can see that for 1 million sales the max amount of allocated memory were 339 MB. From the B.2 and B.3 graphs, we can see that the max amount of allocated memory was 625 and 880 Mb, for 3 and 5 million sales, respectively. This demonstrates the effectiveness of our careful memory management, as memory usage grows sub linearly to the growth in the number of sales.

5.3 Execution time

Another issue will be the potential mishandling of data structures by programmers, since data structures such as Hash tables or binary trees may be poorly implemented, or, in the case of structures like Fibonacci heaps, might not match their theoretical performance in a practical setting.

It is crucial then, to test our use of these data structures to make sure our performance matches these structures' theoretical level of efficiency and run time.

We used the the *time.h* library to evaluate the time it took for our queries to execute. These results can be seen in the D.1 table.

Chapter 6

Conclusion

In conclusion, all the proposed queries are implemented efficiently with execution times that don't worsen as data size increases.

Managing to reduce the execution time for the queries whilst maintaining a small program load time would be a challenge worth pursuing.

Another challenge would be to implement our own version of the data structures instead of the ones provided by GLib, seeing as they could be better equipped to handle the specific problems of our SGV system since we would be able to fully customize them in what way we would see fit.

Trying to implement data bases in our program would also be of our interest as it would allow the application to scale exponentially more since it would not be bound by the RAM size of the computer running it.

Appendix A

Dependency Graph

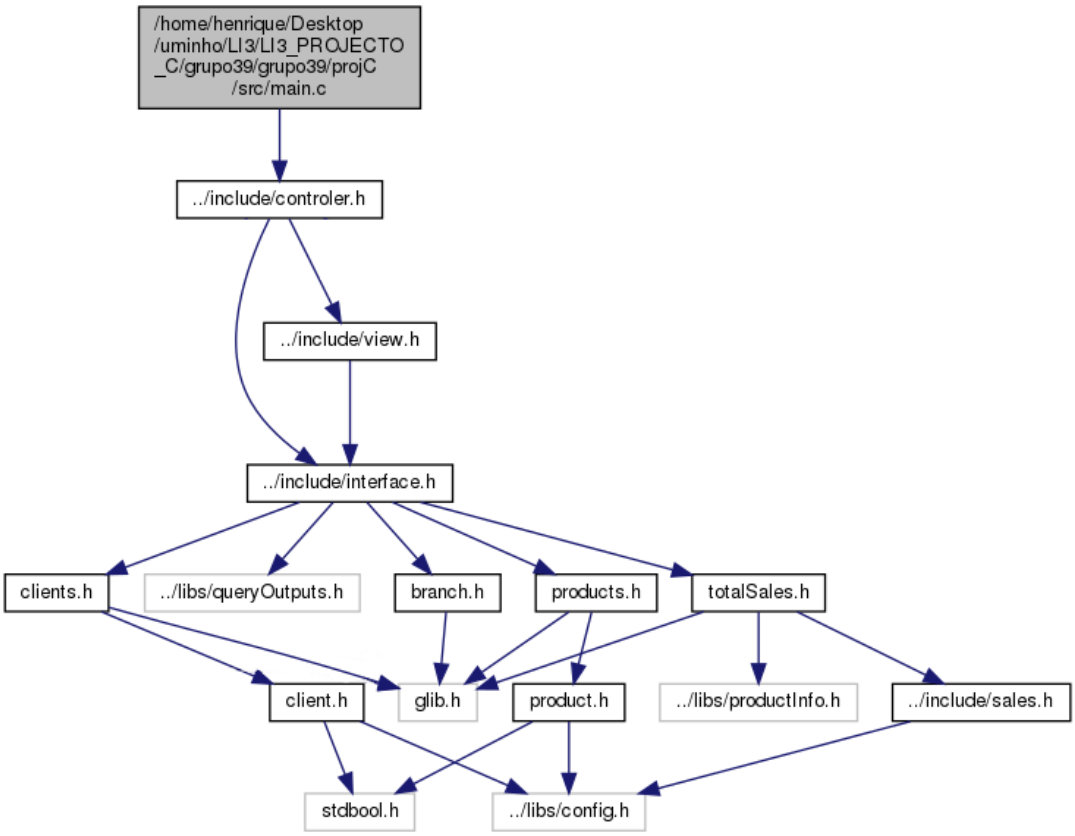

Figure A.1: Dependecy graph of the project.

Appendix B

Memory usage graphs

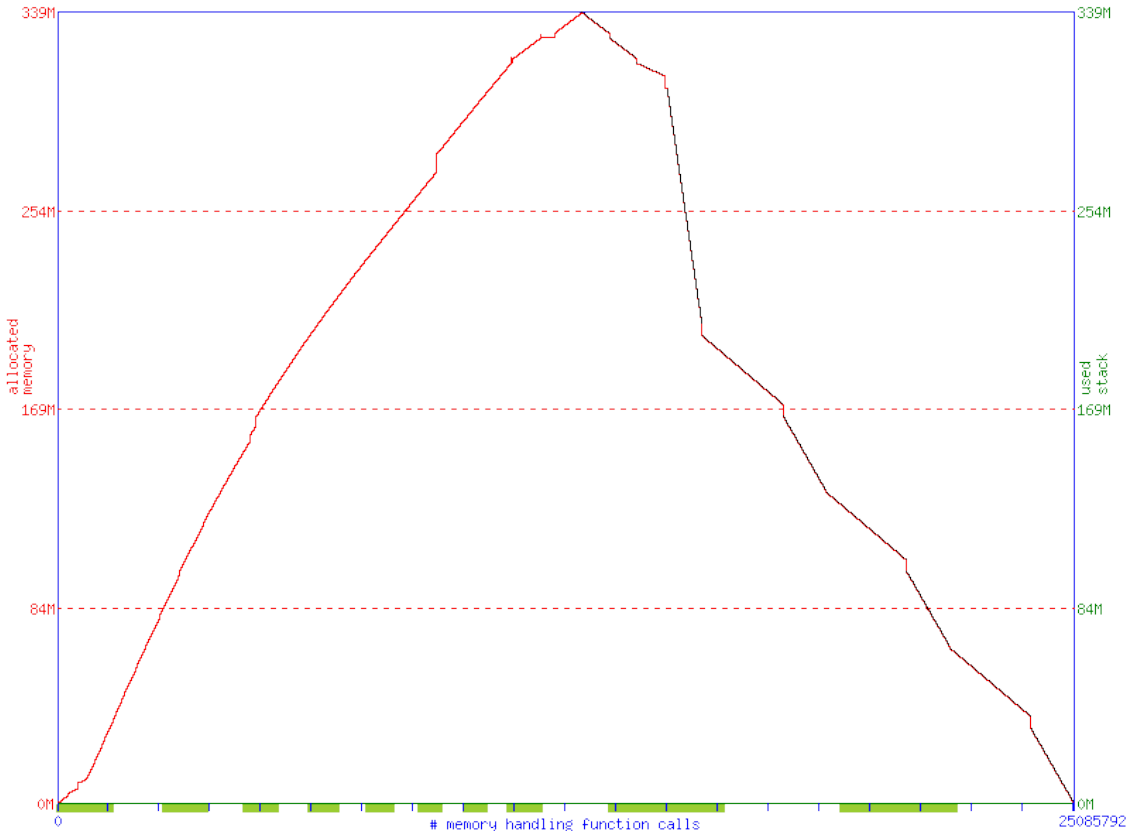

Figure B.1: Graph depicting memory use for 1 million sales.

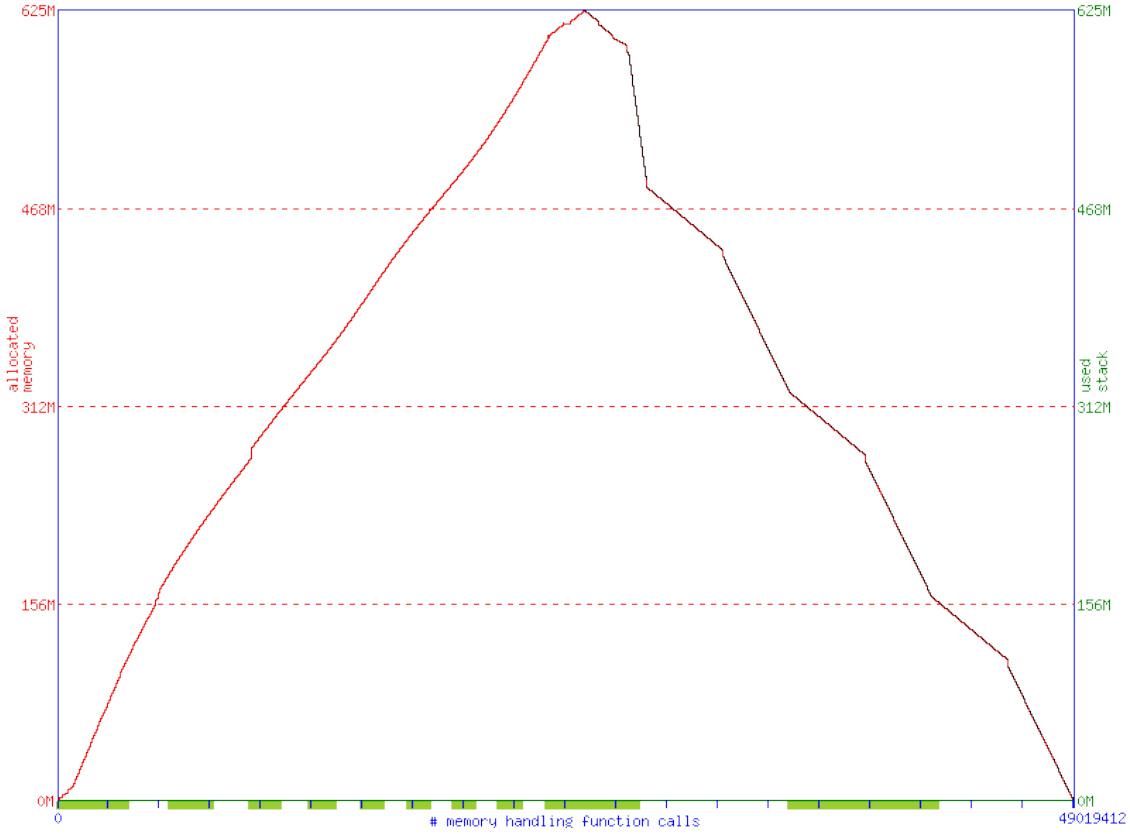

Figure B.2: Graph depicting memory use for 3 million sales.

Figure B.3: Graph depicting memory use for 5 million sales.

Appendix C

Different products module implementations time tables

	1 Million	3 Millions	5 Millions
Load Time	2.66	9.10	15.72
Query 2	4.4×10^{-3}	4.4×10^{-3}	4.4×10^{-3}
Query 4	1×10^{-2}	1.2×10^{-2}	1.2×10^{-2}

Table C.1: Time (in seconds)when the products module is implemented using an Hash table

	1 Million	3 Millions	5 Millions
Load Time	3.94	11.55	21.22
Query 2	1.1×10^{-3}	1.1×10^{-3}	1.1×10^{-3}
Query 4	1.8×10^{-2}	1.8×10^{-2}	1.8×10^{-2}

Table C.2: Time (in seconds)when the products module is implemented using an array of 26 AVL trees

Appendix D

Application's time table

	Input	1 Million	3 Million	5 Million
Load Time	—	2.66	9.10	15.72
Query 2	letter: "I"	2.91×10^{-3}	3.03×10^{-3}	3.04×10^{-3}
Query 3	productID: "AF1184", month: 6	1.10×10^{-5}	1.90×10^{-5}	1.80×10^{-5}
Query 4	branchId:0 (all branches)	9.71×10^{-3}	8.80×10^{-3}	9.28×10^{-3}
Query 5	—	3.62×10^{-3}	3.81×10^{-3}	3.74×10^{-3}
Query 6	—	7.52×10^{-3}	8.95×10^{-3}	6.93×10^{-3}
Query 7	clientID: "Z5000"	6.00×10^{-6}	4.00×10^{-6}	7.00×10^{-6}
Query 8	minMonth: 1, maxMonth: 12	1.00×10^{-6}	1.00×10^{-6}	1.00×10^{-6}
Query 9	productID: "AF1184", branch: 1	1.50×10^{-5}	2.00×10^{-5}	4.90×10^{-5}
Query 10	clientID: "Z5000", month: 7	1.30×10^{-5}	3.00×10^{-5}	3.80×10^{-5}
Query 11	limit: 300000	1.8×10^{-1}	1.8×10^{-1}	1.9×10^{-1}
Query 12	clientID: "Z5000", limit: 300000	6.4×10^{-5}	1.54×10^{-4}	1.69×10^{-4}
DestroySGV	—	6.8×10^{-1}	1.7	2.6

Table D.1: Run time for each query for 1, 3 and 5 million sales