Script 3 Group 60 Report

Luís Pereira (A96681), Rui Oliveira (A95254), Tiago Pereira (A95104)

February 5, 2022

1 Introduction

This report refers to the third and final phase of the project proposed to group 60 for the IT Laboratories III curricular unit in the year of 2021/2022. This script builds on top of the work carried out for the other two scripts, focusing on improving the software in various aspects.

Firstly, the size of the dataset has increased again, meaning it is not garanteed the catalog can be fully loaded into memory. Therefore, a mechanism needs to be developed in order to access the information from file, namely some intermediate layer which acts an index file for the catalog. However, any negative impact this mechanism has on performance should be mitigated as much as possible.

Secondly, an interaction mechanism should also be developed, making it easier for end users to use the application. The original interaction - passing the queries via a text file - must not be removed: the mechanism (we shall refer to it as the Graphical User Interface, or GUI for short) should only be run if the program receives no arguments.

Finally, there is a considerable enfasis on the performance of the application. As a result, tests have to be created in order to assess not only the program's performance, but also its correctness. The objective is for all queries to execute under 5 seconds for the provided dataset.

It should be noted encapsulation and opacity of all the types and data structures are also a very important requisite, requiring special attention when developing new modules, and editing the already existing ones, to make sure those are met.

This report explains all the group's major decisions, providing a high level overview of the application, and a detailed analysis of the testing performed.

2 Technical Solution

2.1 Graphical User Interface

The biggest change in this script to the user experience in this script was the addition of an interaction mechanism in the form of a Graphical User Interface

(GUI for short).

The GUI should be a user-friendly way to input queries into the program and obtain its results in a readable way, including a paging system to split the output into pieces which fit on the screen.

To implement this functionality, the group started by using the ncurses library to help render the content to the screen, but ended up using the ncursesw version of the library instead, as the *regular* ncurses does not support printing non-ASCII characters, which very quickly became a problem given the international scope of the dataset.

The GUI adapts to the size of the terminal, i.e., all the content adapts its position based on the screen size. The number of entries per page is also variable, depending on the height of the terminal. Apart from that, the group also implemented a page which allows the user to explore the catalog without making any queries, as well as a search functionality, meaning the user can search for a string in the catalogs or the output of the executed query.

2.1.1 Pages

The first observation the group made was the need to implement a variety of different pages / menus in order to create an intuitive and easy to use interface. Therefore, there was the need to create a universal representation for these different sections of the GUI. This lead to the creation of the Page type.

The Page type contains all the information needed to display a page to the GUI at any given time. This includes all the components (will be explained later) to render. The module which defines a given page should also provide its state object, which contains all the aditional data needed to display the page, such as input strings, cursor positioning, .etc (in many ways it is similar to a View Model in the MVVM design pattern. This object is stored in the GUI struct as a void$*$. The functions used to get the state, process the user input, update the page based on the state, and to free the state are all stored in the Page object. These objects are managed by a GUI object, which is responsible for containing all the information of the Graphical User Interface, namely which Page object should be rendered, processing any keyboard input - sending the input to the respective input processing function of the active Page -, and determining whether or not the application should be cased / when a change of page is required. It is also responsive for executing any queries a page requests.

2.1.2 Changing pages: Reutilizing the Query object

It is important to note that there will only be a change of pages when there is user input (the only exception to this is the splash screen, however, as the GUI is the one which loads the Catalog, it automatically knows when the splash screen should end).

So to signal to the GUI that there needs to be a page change, the functions which process the user input are forced to return a Query object, similar to those used in the previous script. Should the id of the query be between 1 and 10,the GUI will treat it as "normal" query and will execute it. Should it be a negative integer, the GUI will treat it as a command to change page. All the commands are defined as macros in the page.h header file.

This is yet another example of how flexible the data types implemented in this project can be.

2.1.3 Components

As mentioned previously, the Page object stores information about the components. Components are an opaque and abstract way of separating a GUI object from its actual representation on the screen, meaning that rendering a button is as simple as adding a Button component to the page, and the Button module will take care of printing it properly.

Components are placed on a 2 dimensional grid, where they are displayed. The grid takes up the whole terminal. This makes our GUI responsive, meaning the arrangement of the cells adjust automatically when the terminal is resized.

2.2 Indices

As explained above, the possibility of taking inputs larger than can fit on RAM led inevitably to the need to read the dataset directly from files whenever necessary. Previously, all users, commits and repos were stored as 3 large arrays, and hash tables were created to facilitate lookups for specific properties (users and repos by id, repos by language, commits by repo, etc). This isn't possible anymore, since not even the hash tables are guaranteed to fit in memory. Instead, data structures like these, that serve as indexes to the objects, must also be stored in files.

The group opted to forego the use of hash tables and instead use sorted arrays and binary search, which are easier to manage in files. This management is done by the Indexer type, which can perform insertions and lookups on its respective index. Each element of the index is a key value pair, and the Indexer defines a comparing function to sort and search through the pairs. Each new pair is inserted unsorted though, and only sorted after all insertions to avoid unnecessary writes to the index file.

One advantage of this method is that, after created, each index is a standalone file, outliving the program and opening the possibility of reloading indexes in future instances of the program. This way, a catalog simply corresponds to the collection of all of these index files, along with the compressed input (since the input files contain a lot of unnecessary information, the program copies them to new binary files in a more concise format, optimizing a lot of the information - integers, for example, are stored as only 4 bytes).

2.2.1 Lazy

To optimize the access to the compressed input files, the group created the Lazy type, which mediates access to the compressed objects stored inside them. Instead of loading the whole object, which even in its compressed format can be quite large, the Lazy only loads the requested members, keeping file accesses to their minimum. The same logic is applied when performing writes to the files.

2.3 Catalog Loading

As before, the group used threads to speed up the catalog loading. The constrains are the same: since the input doesn't fit in RAM, only a handful of objects can be processed at any one time. Now framing the design in terms of Indexers, the goal was to ensure as much of the work load as possible could be performed in parallel. The philosophy is the same that of the previous script of the project: reducing the processing done by each query by sacrificing the catalog loading time. For this very reason, the first 4 queries, which take no parameters, are calculated during this process and stored inside the catalog, and simply printed when requested.

The newCatalog function receives a boolean validate parameter to indicate whether the input should be verified. If false, the users, commits and repos can be read independtly, storing the relevant information in their Indexers and generating the compressed input files. If true, a scheme was designed to guarantee as much parallelism as possible, using 2 buffers to store the existing repo ids and their respective latest commit dates. These are the only data structures in the program with no practical size ceiling. There is, however, a theoretical ceiling: since no two repos can have the same id, and ids are guaranteed to fit inside the c signed integer type, the number of repositories is limited to $2^{31} - 1$, or 2147483647. Since for each repo the program stores 4 bytes, these buffers would take up approximately 8GB of RAM, a feasible number with the current computing technology. Alternatively, the buffers could be stored as files, albeit with a considerable time penalty.

2.3.1 *Phantom Repositories*

A strange consequence of the loading scheme was what the group decided to call *phantom repositories.* The specifications for data validation consist of 4 filters that should be applied in order. Relevant to the *phantom repos* are the following:

- (a) Filter out commits to non-existent repositories;
- (b) Filter out repositories that mention non-existent users (owner_id);

According to these rules, if a repository is filtered out as of rule (b), the commits to that repository won't be eliminated, since the step to check their validity is taken prior, in rule (a). This repository becomes a *phantom*, not registered as a repository since it has been filtered out but existent as the target of its commits. This behaviour only came to the surface when testing test cases generated by the group, since the provided input files did not contain commits to invalid repositories. To maintain consistency with the previous scripts of the project, the group decided to keep this behaviour and ignore any commits to *phantom repositories* when performing the queries.

2.4 Cache

To minimize the impact of file accesses, a Cache was created. This type serves as a mediator for most file accesses in the program, keeping a list of already read file positions and their respective contents. Each time a file is accessed, the Cache then checks whether that file position is already loaded in memory. If not, it is loaded anew, deleting the cache line unused for the longest. The size of each cache line (a chunk that is read at once) was experimentally defined at the optimal value of 1024 bytes (Cache size). Writing to files is also mediated by the Cache, avoiding accessing the actual files at all until the end of the program, when the cache is flushed. Manual flush functions were still created, though, if ever they were needed.

This strategy proved invaluable in small test cases, reducing the time of file accesses by almost ten-fold. For larger test cases though, where the cache can't hold all of the information stored in files, the benefits of the Cache are largely reduced. Check the Performance Analysis section of the report for a detailed analysis.

3 Testing

The final part of the project is testing the application. The group split the testing into two categories with different purposes.

3.1 Correction Tests

The first set of tests was created by hand in order to assess the correction of the software, consisting of small inputs which are verifiable by hand.

Each test case corresponds to a folder. In said folders, there are four files: the users.csv, commits.csv, repos.csv, which store the dataset of the test, and the queries.txt file containing the queries of the test case. In this set of the tests, this file contains 10 queries, one of each type.

In addition, there is a folder named expected, which contains ten files, which are the expected output of each query.

3.2 Performance Tests

The second set of tests was designed to test the performance of the application, consisting of increasingly larger and larger datasets. More specifically, the datasets - i.e., the .csv files - had a total size of 200MB, 1GB and 10GB

These were generated with an auxiliary program written in C++. The structure of these tests is identical to the first group, except for the lack of the expected folder, as there is no feasible way to assess the correction of the program for such large files, and the possibility of the queries.txt having more than 10 queries.

3.3 Testing suite

To run the developed test cases in a simple and efficient way, the group created a small auxiliary program - tests.c compiled with make test. The program is called with the location of the tests to perform (possibly more than one), and compares the expected output and the actual output, checks for the execution time and outputs the results in a user friendly way.

This is a very flexible setup, that could be used in other projects without major changes.

4 Performance Analysis

4.1 Methodology

The results presented in this section were obtained by running the application 10 times, registering the time and memory usage for each of the ten queries and for loading the catalog. Of the obtained values, the minimum and maximum were excluded, and the average was calculated for those remaining 8.

The machine used to conduct the tests has the following specifications:

- AMD Ryzen 7 2700 CPU at stock clock speed
- 8GB of DDR4-2666MHz RAM + 8GB swap
- Goodram 256GB SSD Sata 6Gb/s
- Arch Linux running KDE Plasma v.5.23.5 and kernel 5.15.13-arch1-1 (64 bit)

The program was compiled in gcc using the optimization flag –O3.

4.2 Cache size

The first tests carried out were to determine what the optimal cache configuration would be in terms of the size of a line. To determine this, the group ran the program with different line sizes against a dataset about 5 times larger than the cache size.

The results obtained are presented in Figure 1.

Based on the results, the group decided to use a line size of 1024 bytes, as the gains from a bigger line are neglegible. The group experimented with other cache size / dataset size ratios, and this pattern remained identical.

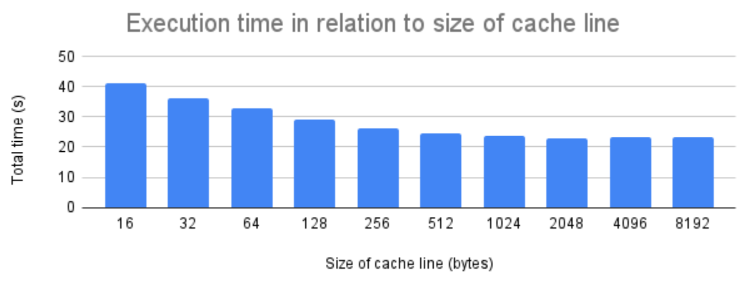

Figure 1: The total execution time (loading catalogs and 10 queries) for different cache line sizes

4.3 Scaling

The following tests were conducted in order to assess the scalability of the project, i.e., how well the application performs when the size of the dataset increases. As explained previously, different datasets of varying were created, and the program was tested against each one.

The results obtained are displayed in the below figures.

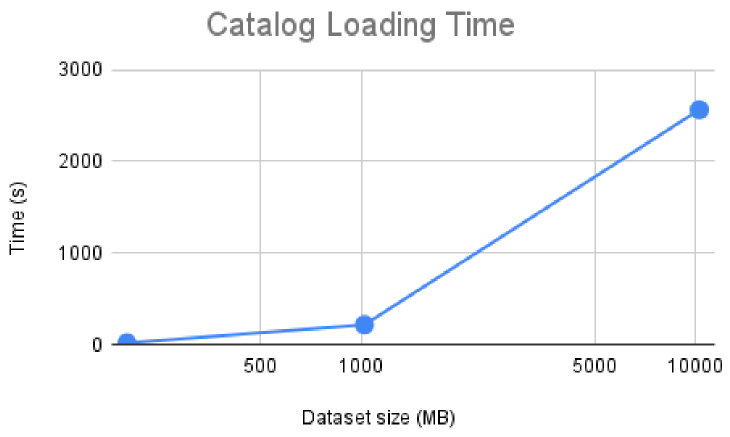

Figure 2: The loading time of the catalogs based on their size (in bytes)

From these values one can conclude that memory usage is essentially constant (see Figure 3) at 1 gigabyte (the size of the cache). It is also shown the ability of the program to process large datasets, considering that the loading time of the catalog is growing linearly to the size of the dataset.[1] . This behaviour is also very similar to what happens with the queries (see Figure 4).

[1]It should be noted that in Figure 2 the x-axis has a logarithmic scale

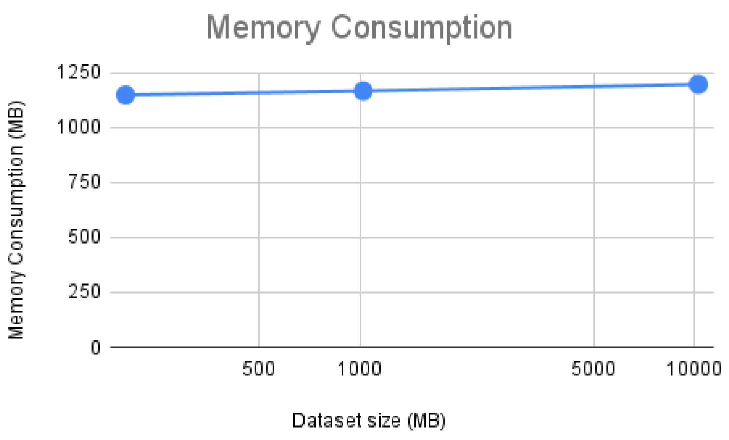

Figure 3: The memory usage of the program (in bytes)

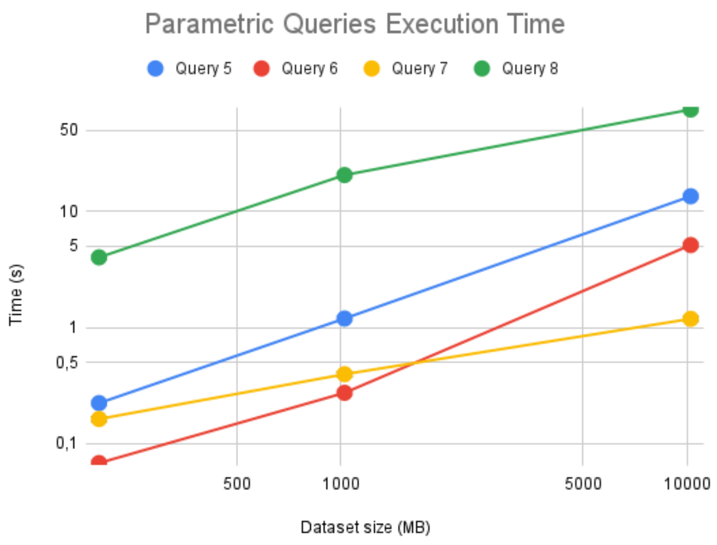

Figure 4: Execution time of the queries for different dataset sizes)

Again this is a testament to the efficiency of the application, which has always been a priority for the group since the beginning of the project.

4.4 Comparison between machines

The final part of the analysis is comparing performance between different machines, hoping to learn more about the hardware necessities of the application. The group tested against a 1GB dataset, as its size means the application will not need a long time to finish, but it is also long enough to highlight the differences between the machines.

The specifications of the used machines are presented in the Table 1.

	Machine 1	Machine 2	Machine 3
CPU	Ryzen 7 2700	Intel i7-8750H	Ryzen 5 4560U
Core/Threads	8/16	6/12	6/12
RAM	8GB DDR4	8GB DDR4	8GB DDR4
Disk	256GB SSD	1TB NVME	2TB NVME
OS	Arch Linux	Manjaro	Ubuntu 20.04

Table 1: Specifications of the machines used for this test

The obtained results are presented in Table 2

	Machine 1	Machine 2	Machine 3
Catalog Loading	215.506001	334.315600	217.607612
Query 5	13.550254	10.325092	16.788716
Query 6	0.516211	0.551799	0.530260
Query 7	0.555208	0.522233	0.468602
Query 8	43.884924	37.248766	36.22105
Query 9	10.504265	8.541580	7.994096
Query 10	57.200414	72.784081	54.546938

Table 2: Execution time (in seconds) of the program for the different machines

The only significant differences in execution times lie in the loading of the catalog and in the last query. These are most likely the result of different processor architectures, given the biggest difference between machine 2 and the rest is the Intel CPU. One takeaway from these results is also the fact that the disk access is as big of a limiting factor as expected, which proves the efficiency of the cache.

5 Conclusion

Overall, the group considers this project to have been a success. All the requirements were met and the group went further with the development of the application than strictly required, namely in the interface.

The performance of the project is really good, as demonstrated in the previous section, namely due to the group's commitment to having a very efficient program since the start of the project.

There are several ways to improve the application even further, which were not implemented due to time constraints. To name but a few, splitting the GUI into a client/server which would communicate via pipes would have been a really interesting design choice, improving the grid system to be more responsive - similarly to some CSS libraries like Bootstrap - and creating a garbage collector so that the GUI code can be more readable, as there are a lot of free calls in other to avoid memory leaks.

In addition, a more sofisticated indexing structure, which would allow the storage of other data structures such as hash tables would be beneficial in terms of performance, as well as to allow multiple instances of the program running at once. Finally, creating a more advanced search module who improve the performance of the search funcionality of the GUI a lot.

In conclusion, the group believes the project was a success, and that all the requirements were not only met but also exceeded, making this application a solid solution to the given problem.