Projeto Laboratórios de Informática III
Grupo 19 - Fase 1

Projeto desenvolvido por
Alex Sá (A104257), Paulo Ferreira (A96268), e Rafael Fernandes (A104271)

Licenciatura em Engenharia Informática

Universidade do Minho
Escola de Engenharia

Departamento de Informática
Universidade do Minho

Contents

1 Abstract

The following report encompasses the progress and decisions made by Group 19 up to the date of the delivery of the first phase of the project developed within the scope of the subject *Laboratórios de Informática III* during the academic year 2023/2024. The main theme of the project is the development of a database stored in memory that will receive four datasets containing flights, reservations, users, and passengers.

Many goals were set for the team to achieve in this first phase. Some of them include the ability to select valid information from the datasets by obeying rules chosen by the professors, the implementation of a batch execution mode, the implementation of optimized algorithms to solve six of ten queries, and the execution of the program without any memory leaks. Documentation and code readability were also among the goals the group set for themselves.

2 Introduction

From the outset, our group's primary aim has been to attain the highest possible grade. The following pages detail our thought process and the challenges we navigated to achieve this overarching goal.

In our pursuit of academic excellence, we deliberately structured the entire project before diving into code. Our key focus was on ensuring maximum stability and prioritizing resilience. This approach allows for straightforward adjustments without causing widespread disruptions. Additionally, the group implemented strategies to minimize memory usage and reduce execution time. This involved utilizing optimized structures to store datasets and exercising caution in algorithm selection, such as avoiding redundant calls to functions like strlen, mallocs, and others that are very resource-expensive.

To kick off the initial phase, which revolved around implementing six queries out of the total ten, the group commenced by conducting a comprehensive analysis of the queries. They meticulously documented the utilized data, selectively removing fields like comments and notes to optimize memory usage. This initial step was a proactive measure to enhance the efficiency of the project. Subsequently, the group delved into a detailed study of each query, striving to identify the optimal organization for all datasets. Once the groundwork was laid, the group strategically adopted a "divide and conquer" approach, initiating with the parsing of the data. While this tactic may appear unconventional due to breaking down a primary task into smaller components, it proved effective. This approach allowed each group member to focus on specific elements, facilitating the optimization of algorithms collectively. This methodical strategy was crucial, considering that data parsing stood as the focal task of the entire project.

Concerted efforts were made to prevent the program from excessively consuming the available memory (capped at 2GB RAM). The introduction of a pagination system surfaced as an initial strategy to tackle this concern. This concept is currently in the contemplative phase, with plans for potential integration in the second phase of development. The objective is to maintain the program's memory usage significantly below the specified limit. This underscores their commitment to proactive memory management. Concurrently, the introduction of threading as an additional measure to enhance overall program performance is also being considered, thereby assigning equal importance to both initiatives.

3 Application Details

3.1 Overview

Providing an overarching view of the project's current status as of November 17, 2023, the application has successfully achieved its intended functionality for the initial phase.

Commencing with the parsing aspect, the primary function, "main," accepts paths to both the CSV files and the "input.txt" file. Subsequently, the "batch" function is invoked, initiating the population of our database by processing each file sequentially. Within the "parse_file" function, various subordinate functions are called in a systematic sequence. These functions collectively tokenize lines, validate their integrity, parse them into the desired structure, write the data into the database, and manage memory. This set of functions constitutes the foundational framework for our parsing operations, serving as the backbone for all other functions in this initial phase.

```c

void parse_file(
    char* filename,
    Tokenizer(tokenizer),
    PreprocessFunction(preprocess),
    VerifyFunction(verifier),
    ParseFunction(parser),
    WriteFunction(writer),
    WriteFunction(discarder),
    DestructFunction(destructor),
    ...
) {
    // Implementation details follow
}
```

Following the parsing of datasets, a function is invoked to process information from "input.txt," encompassing queries and their respective inputs. This phase showcases the efficacy of our data structures, as elucidated in section 3.2. Additionally, the selected algorithm for each query is discussed, providing concise explanations for queries 1, 3, 4, 5, 7, and 9 out of the total of 10 queries.

3.1.1 Query Solutions

The following outlines the strategies employed to solve specific queries:

- **First Query:**
  - **Algorithm:** Hashtable-based approach.
  - **Additional Module:** Statistics for value calculations(total spent by a user, total price for a reservation, and others...).

- **Third Query:**
  - **Algorithm:** Binary search with range determination(made possible by previous sorting).
  - **Catalog Sorting:** By hotel ID, start date, and reservation ID.
- **Fourth Query:**
  - **Algorithm:** Similar to the third query.
  - **Catalog Sorting:** By hotel ID, start date, and reservation ID.
- **Fifth Query:**
  - **Algorithm:** Similar to the third and fourth queries.
  - **Catalog Sorting:** By origin, scheduled departure date, scheduled arrival date, and flight ID.
- **Seventh Query:**
  - **Algorithm:** Ordered Sequence
  - **Additional Module:** Statistic module for median calculation.
- **Ninth Query:**
  - **Algorithm:** Similar to the third query.
  - **Catalog Sorting:** By name and user ID.

As of now, this encapsulates a comprehensive summary of how the application works. The subsequent section will delve into a more detailed examination of the catalogs and data management, offering a deeper insight into the intricacies of our project.

3.2 Managing Data

To optimize the storage process, our group strategically adopted the use of Catalogs, each composed of two integral components: a Hashtable for swift data access with a constant time complexity of O(1) and a GArray for organized and sorted data storage. The implementation harnesses the power of functions from the GLib library[1].

```c
typedef struct catalog {
    GHashTable *hashTable;
    GArray *array;
    int itemCount;
} Catalog;
```

[1]GLib is a core library that underlies projects such as GTK and GNOME.

The decision to incorporate these data structures is grounded in their unique advantages. The Hashtable ensures rapid data access, offering an optimal solution for accessing data through an ID. Simultaneously, the sorted nature of the GArray facilitates streamlined access and searches, achieving a time complexity of $O(\log_2 n)$ thanks to efficient binary search. The utilization of functions from the GLib library further contributes to the overall optimization of the system.

This approach has proven exceptionally effective for our datasets, striking a harmonious balance between constant time access for swift operations and sorted storage for efficient retrieval. This equilibrium addresses the diverse demands posed by the project's objectives.

It's noteworthy that certain queries benefit more from one way of sorting over others, prompting careful consideration of the initial array order based on the most frequent occurrences. An intriguing aspect of our management approach is that, even when adjustments to the order are necessary, they typically involve the second component or beyond. The first component remains relatively consistent, enabling us to leverage binary search for locating a value with that component. Subsequently, selecting the first and last appearances of that value with the specified component enables us to streamline the sorting process exclusively for those values.

3.3 Modularity and Encapsulation

In the development of our project, modularity and encapsulation were not only fundamental principles guiding the organization of our codebase but also the two pillars of evaluation for this project. This section outlines the modular structure we adopted and discusses how encapsulation was implemented to enhance the maintainability and readability of our code.

3.3.1 Modular Code Organization

The project directory structure reflects a modular organization, promoting a clear separation of concerns and facilitating code maintenance. The key components of our modular structure include:

- **Include Directory:** Contains header files (`.h`) defining interfaces for various modules.
- **Source (`src`) Directory:** Houses the implementation files (`.c`) corresponding to each module.
- **Object (`obj`) Directory:** Stores compiled object files (`.o`) generated during the build process.
- **Library (`lib`) and Binary (`out`) Directories:** Intended for libraries and executable binaries, respectively.
- **Resultados Directory:** Contains output files such as the invalid lines and outputs from the queries.

Each module, encapsulated within its own directory, encompasses a specific functionality, promoting ease of navigation and reducing the complexity of individual components.

3.3.2 Catalog Modules

In structuring our catalog modules, we opted for a clear division into two significant components. The first component encompasses functions that operate on generic data, while the second focuses on operations specific to each data structure.

The `CatalogManager` module, represented by `catalogManager.h` and `catalogManager.c`, serves as the central orchestrator, managing interactions among different catalogs seamlessly.

Within the `catalog` directory, each other module encapsulates distinct operations and data structures tailored to flights, passengers, reservations, and users. Each module consists of a header file (`.h`) defining the interface and a corresponding implementation file (`.c`). This modular approach not only promotes a well-organized codebase but also ensures clarity in both generic and data-specific functionalities.

3.3.3 Collections Module

The `collections` directory encapsulates modules that define data structures representing fundamental entities in our system, such as flights, passengers, reservations, and users. Each module (`flight.h`, `passenger.h`, `reservation.h`, `user.h`) defines the structure and operations specific to its entity.

3.3.4 Executers Module

The `executers` module, encompassing the `batch.h` file, orchestrates the execution of batch operations. It acts as a coordinator, invoking relevant functions from other modules to fulfill batch processing requirements.

3.3.5 Parser Module

The `parser` module, contained in the `parser` directory, encapsulates functionality related to parsing input data and output. The `parser.h` file defines the interface, while `parser.c` implements the parsing logic.

3.3.6 Queries Module

The `queries` module, residing in the `queries` directory, encapsulates operations related to queries. The `queries.h` file defines the query interfaces, and `queries.c` implements the query logic.

3.3.7 Stats Module

The `stats` module, encompassing the `stats.h` file, encapsulates functionality related to statistics. This module provides an interface for calculating and managing statistical data.

3.3.8 Util Module

The `util` module, housed within the `util` directory, encapsulates utility functions used across multiple modules. This includes error handling, input/output operations, and various string and memory manipulation functions.

3.3.9 System Architecture

The following diagram provides a high-level architectural overview of the system, illustrating the interactions between different modules:

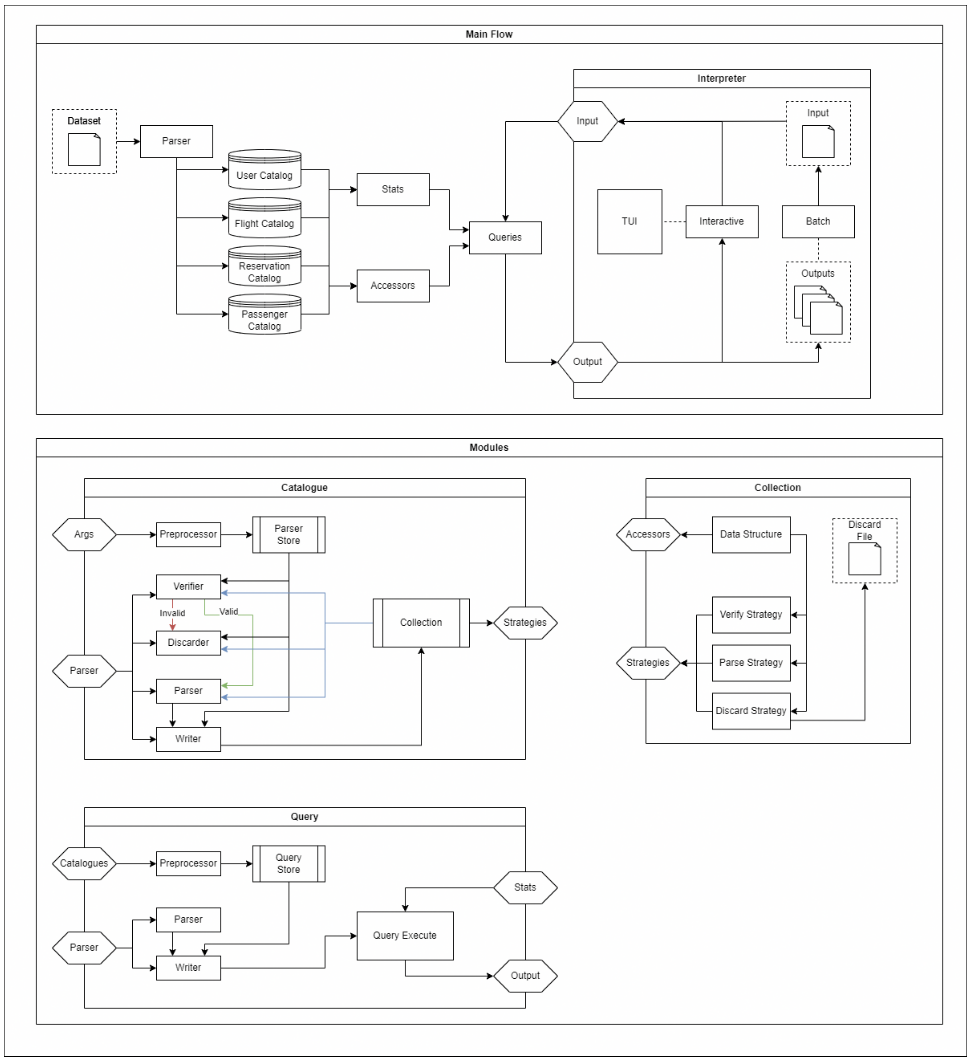

Figure 1: System Architecture

3.4 Performance evaluation

For the comprehensive evaluation of the global performance concerning the implementation of all first-phase requirements, the group conducted a series of 15 tests, calculating an overall average. The specifications of the machines utilized are detailed in the table below:

	Machine 1	Machine 2	Machine 3
CPU	Ryzen 7 5800H	Intel Core i7-1065G7	Intel Core i7-1165G7
Cores/Threads	8/16	4/8	4/8
RAM	16GB DDR4 3200	12GB DDR4 2666	16GB DDR4 3200
Disk	1 TB SSD M.2	512 GB SSD M.2	1 TB SSD M.2
OS	Debian 12	WSL Ubuntu	Pop!_OS Linux
Time Avg.	0.165s	0.215s	0.168s

Upon scrutinizing the analysis, a subtle difference is observed between the tests conducted on Machine 1 and Machine 3, in comparison to Machine 2. Despite all machines sharing the same x86 architecture, the slight variation is likely attributed to the slower RAM, especially in Machine 2. Additionally, the utilization of Windows Subsystem for Linux (WSL) for testing on Machine 2 could contribute to the observed disparity, given the constraints on the filesystem level, which arise from its reliance on a virtual machine that writes to a virtual disk, having limited access to RAM and CPU.

3.5 Experienced Difficulties

Throughout the development process, the group encountered various challenges that demanded thoughtful consideration and problem-solving. These difficulties encompassed several aspects of the project:

- **Choosing Data Structures:** The initial selection of data structures posed a considerable challenge. The group began with considerations for a Hashtable and a tree structure. However, it became apparent that the tree structure consumed significantly more space and was not well-suited for sequential searches, ultimately leading to the adoption of the sorted GArray.
- **Using Generic Data Types:** Incorporating generic data types, while ultimately beneficial, presented a learning curve. Although it proved advantageous in the long run, the group faced initial challenges in adapting to this approach.
- **Making Modularity and Encapsulation:** Establishing modularity and encapsulation was a non-trivial task. Determining the appropriate placement of functionalities and

deciding when to create copies of data were initial hurdles. Overcoming these challenges required iterative refinement of the code organization.

- **Working with Pointers:** The use of pointers, particularly within the context of GLib, where pointers to pointers are commonly employed, presented notable challenges. The obscured visibility of these pointers, deliberately shielded by GLib, made it difficult to discern their targets even with various debugging tools. Consequently, dereferencing these pointers became a task requiring meticulous care to guarantee the accuracy and efficiency of the code.
- **Managing Workload:**The sheer volume of work required for the first phase was demanding, especially considering the limited timeframe available. Striking a balance between project requirements and time constraints became a crucial aspect of the development process.

Addressing these challenges involved collaborative problem-solving and iterative adjustments, ultimately contributing to the group's growth and the refinement of the project.

4 Future Concerns

As we assess our current project, several key areas emerge as focal points for future refinement. First and foremost, our commitment to efficiency drives the need for continuous query optimization. This involves fine-tuning algorithms specific to each query to extract even greater performance from our system.

While our codebase maintains readability and modularity, we acknowledge the critical role of comprehensive documentation. In future developments, a heightened focus on enhancing code documentation will ensure a clear and accessible understanding of our project.

Looking ahead, we consider the implementation of an interactive mode, potentially leveraging tools like ncurses. This move aims to introduce a more dynamic and user-friendly interface, ultimately enhancing the overall user experience with our program, and also completing one of the goals set by the lectures.

Ensuring the reliability and robustness of our system is paramount. Future development efforts will prioritize building a robust testing framework, including the creation of dedicated test modules to systematically validate different aspects of the program.

Additionally, we are actively exploring the implementation of a pagination system to proactively manage memory, particularly with large datasets. This feature, under consideration for integration in upcoming development phases, aligns with our commitment to proactive memory management.

Finally, threading integration is on the horizon as an additional optimization measure. This involves parallelizing specific operations to enhance overall program performance, further contributing to the evolution of our project. These collective considerations outline our roadmap for future development, emphasizing continuous improvement and adherence to best practices.

5 Conclusion

In conclusion, our project has navigated through strategic decisions, modular design, and the implementation of efficient data structures. The adoption of Catalogs, with a combination of Hashtable and GArray, has proven effective in balancing rapid data access and sorted storage. Challenges, such as working with pointers and ensuring modularity, were overcome through meticulous design and careful handling. The future holds exciting prospects, including continuous query optimization, enhanced code documentation, an interactive mode for user-friendly interaction, a robust testing framework, and potential integrations like pagination and threading. Our commitment to efficiency and proactive development remains unwavering, marking a promising trajectory for the evolution of our project.