# LI3 - Relatório da Fase II - Grupo 12

Humberto Gomes (A104348)
José Lopes (A104541)
José Matos (A100612)

janeiro de 2024

**Resumo**

Após a entrega da primeira fase deste trabalho, procurámos implementar as funcionalidades em falta para a conclusão do desenvolvimento desta aplicação: as restantes quatro *queries*, um modo de uso interativo (uma interface TUI), testes funcionais e de desempenho, e suporte para um *dataset* de maior dimensão, mantendo um desempenho aceitável. Adicionalmente, procurámos também corrigir alguns aspetos do encapsulamento e da modularidade no nosso projeto.

## 1 Estrutura do trabalho

Nesta secção, procuramos descrever as diferenças entre a nossa arquitetura atual e a que tínhamos na primeira fase do projeto. Devido ao elevado número de módulos, dividimos a aplicação em subsistemas, que são descritos e esquematizados um a um, apresentados apenas com as relações de dependência mais relevantes. Segue-se a nossa convenção gráfica:

- Um retângulo com cantos arredondados representa uma estrutura de dados (entidade, gestor, alocador, ...);
- Um retângulo sem cantos arredondados representa um módulo cuja tarefa principal é a execução de código. Mesmo assim, alguns destes módulos podem conter estruturas de dados auxiliares (por exemplo, uma gramática definida no módulo de um *parser*);
- $A \to B$ significa que o módulo $A$ depende do módulo $B$.
- Nos subsistemas já existentes na primeira fase, um módulo ou relação de dependência representado(a) a verde indica que o(a) mesmo(a) estava ausente na entrega anterior.

### 1.1 *Parsing*

Nesta fase do projeto, separámos em diferentes módulos as funcionalidades de *input* de dados e *output* de erros num *dataset*, antes concentradas no `dataset_loader`. Agora, este módulo apenas gere duas

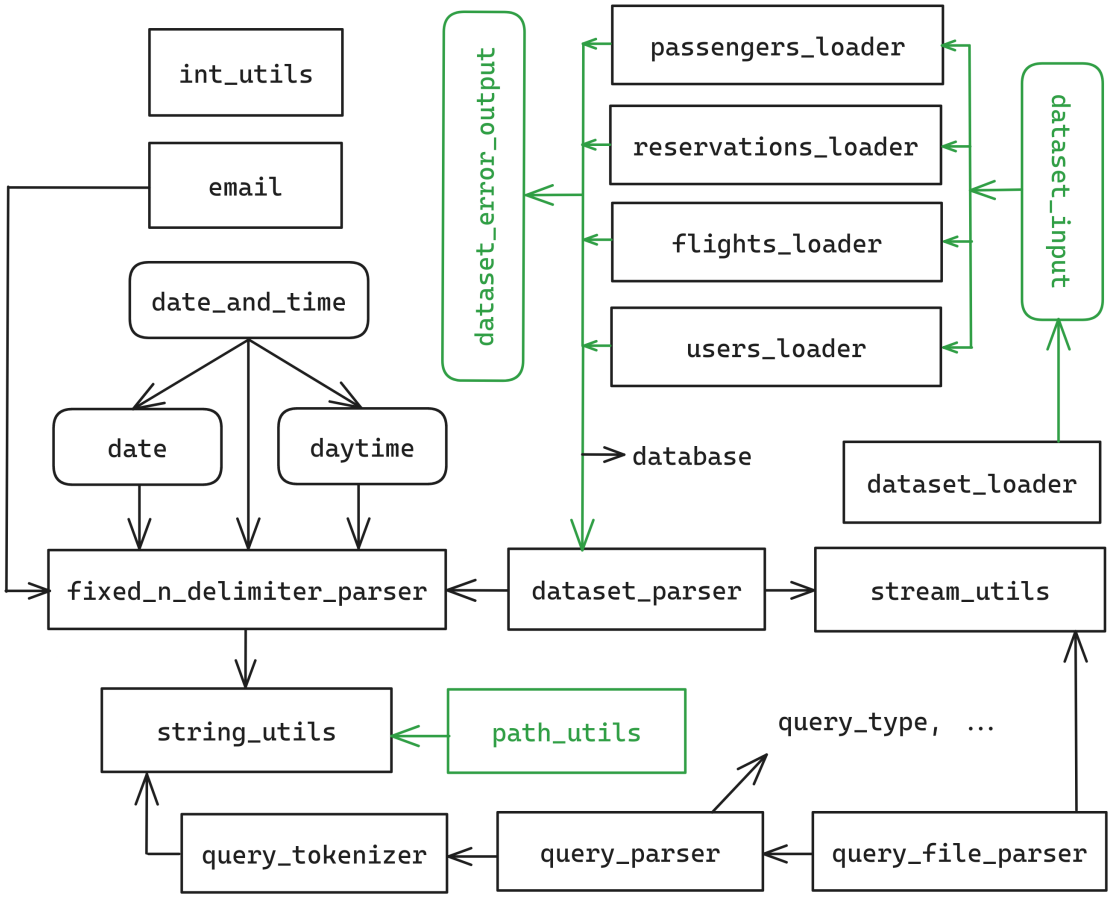


Figura 1: Diagrama de dependências do subsistema de *parsing*

estruturas de dados formadas por *handles* de ficheiros, `dataset_input` e `dataset_error_output`. Além disso, adicionámos o módulo `path_utils`, para manipulação de caminhos para ficheiros.

## 1.2 Entidades

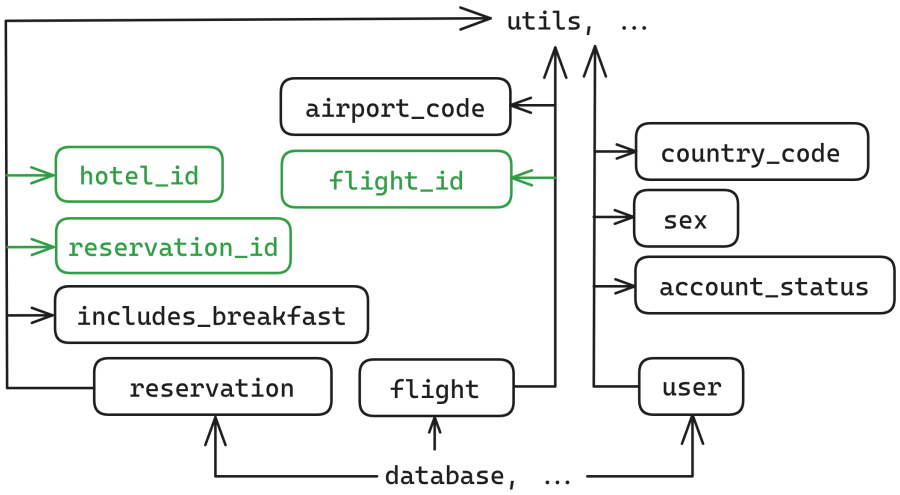


Figura 2: Diagrama de dependências das entidades da aplicação

Nas entidades, criámos módulos para os identificadores de voos, reservas e hotéis, de modo a remover código de *parsing* e *display* previamente duplicado em vários locais da *codebase*. Ademais, passaram a ser os *setters* das entidades a validar se os valores providenciados para os seus campos são válidos. Este antes era o trabalho dos módulos `*_loader`, sendo possível ao programador, usando *setters*, criar uma entidade inválida.

## 1.3 Catálogos

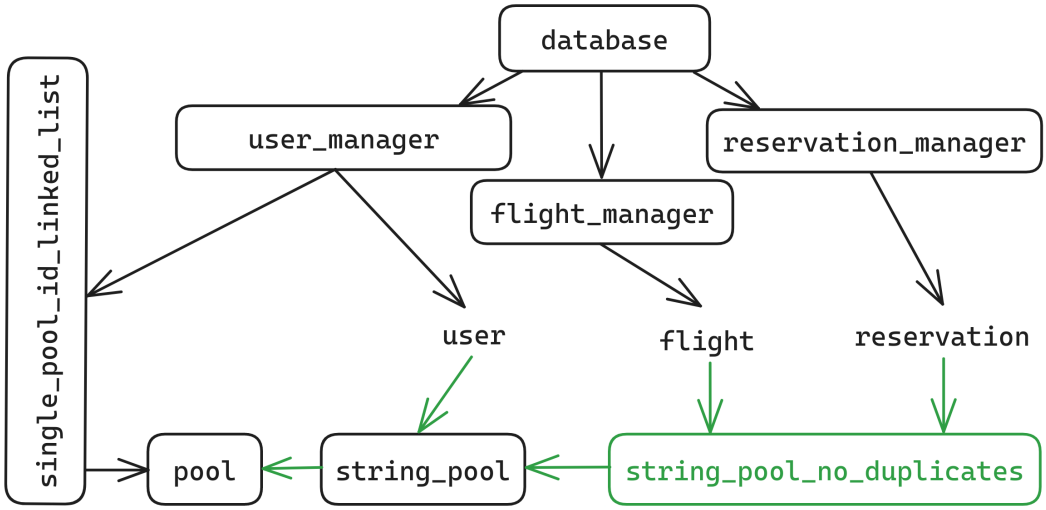


Figura 3: Diagrama de dependências dos catálogos na aplicação

De modo a diminuir o uso de memória da aplicação, criámos o `string_pool_no_duplicates`, um novo alocador que armazena apenas uma vez *strings* sujeitas a repetição, como nomes de hotéis e de companhias aéreas.

Quanto às novas relações de dependência, removemos muito código duplicado ao adaptar a `pool`, para com ela ser possível implementar a `string_pool`. Por último, as entidades passam a depender dos alocadores, por motivos explicados em Conciliação com alocadores.

## 1.4 *Queries*

Como todas as *queries* apresentam a mesma formatação de *output*, adicionámos o `query_writer`, uma estrutura que formata o *output* de uma *query* automaticamente, e o escreve para um ficheiro (modo *batch*) ou para um conjunto de linhas em memória (modo interativo). Os módulos `GConstPtrArray` e `GConstKeyHashTable` são explicados em Conciliação com a `glib`. Segue-se a descrição do funcionamento das *queries* novas e das que sofreram mudanças significativas:

- **Q05** - Listagem dos voos com origem num dado aeroporto entre duas datas. Com uma única iteração pelos voos para todas as *queries* deste tipo, verifica-se se cada voo cumpre as condições para ser parte da resposta a uma *query*. Em caso afirmativo, adiciona-se esse voo a um *array*

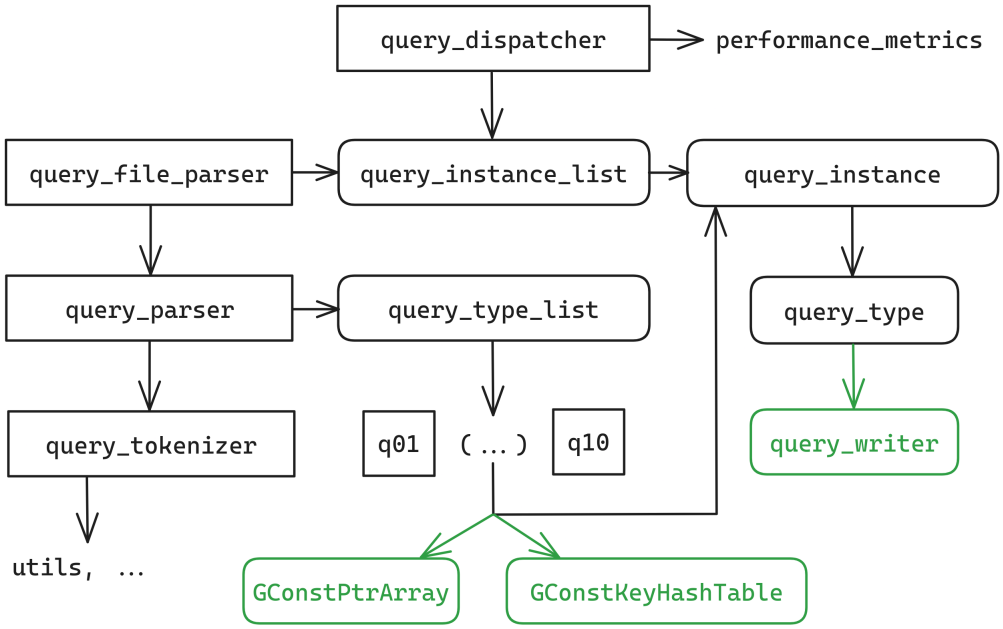


Figura 4: Diagrama de dependências do subsistema de *queries*

associado à *query*. A execução de cada *query* passa apenas pela ordenação e apresentação do seu *array* correspondente.

- **Q07** - Listagem dos aeroportos com maior mediana de atrasos. Com uma única iteração pelos voos para todas as *queries* deste tipo, gera-se uma tabela de *hash* que associa aeroportos a *arrays* de atrasos. Depois, calcula-se a mediana para cada *array*, gerando-se outra *hash table*, entre aeroportos e medianas. Executar cada *query* consiste na consulta direta desta tabela.

- **Q08** - Cálculo da receita de um hotel entre duas datas. Com uma única iteração pelas reservas para todas as *queries* deste tipo, gera-se uma tabela de *hash* que associa *queries* a receitas. Interseta-se, para cada reserva, o intervalo de noites dormidas com o intervalo pedido em cada *query*, para o cálculo do número de noites e da receita total. Executar cada *query* consiste simplesmente na consulta da tabela gerada.

- **Q09** - Listagem de todos os utilizadores cujo nome comece com um dado prefixo. A nova implementação desta *query* é muito semelhante à da **Q05**, mas iterando pelos utilizadores em vez dos voos. Esta solução é significativamente mais veloz que a anterior, que fazia uma iteração pelos utilizadores por *query*.

- **Q10** - Cálculo de várias métricas gerais da aplicação. Com uma única iteração pela totalidade dos catálogos para todas as *queries* deste tipo, verifica-se se cada entidade deve ou não ser contada para a resposta a cada *query*. Como o acesso aos passageiros é feito utilizador a utilizador, torna-se possível, usando uma *bitmask*, facilmente discernir os passageiros únicos. A execução de cada *query* torna-se uma simples pesquisa pelas respostas previamente geradas.

## 1.5 Modo interativo

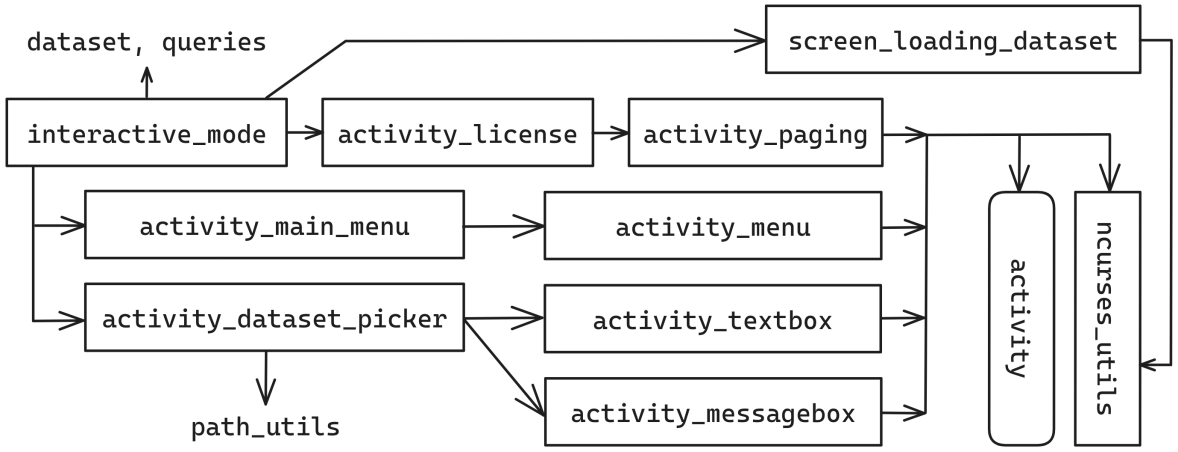


Figura 5: Diagrama de dependências do modo interativo

O modo interativo revolve à volta do conceito de atividade (`activity`), uma estrutura polimórfica que gere o *loop* de eventos de uma interface `curses`. Imagens das atividades implementadas podem ser encontradas em *Screenshots* do modo interativo. `screen_loading_dataset` não responde a eventos, mas desenha no ecrã que a aplicação se encontra a ler um *dataset*. O módulo `interactive_mode` é responsável por controlar a troca entre atividades, formando a aplicação final. Neste subsistema, destaca-se a facilidade de uso (instruções em todas as interfaces, explorador de ficheiros para escolher um *dataset*, ...) e o suporte para Unicode no *layout* das atividades.

## 1.6 Subsistema de testes

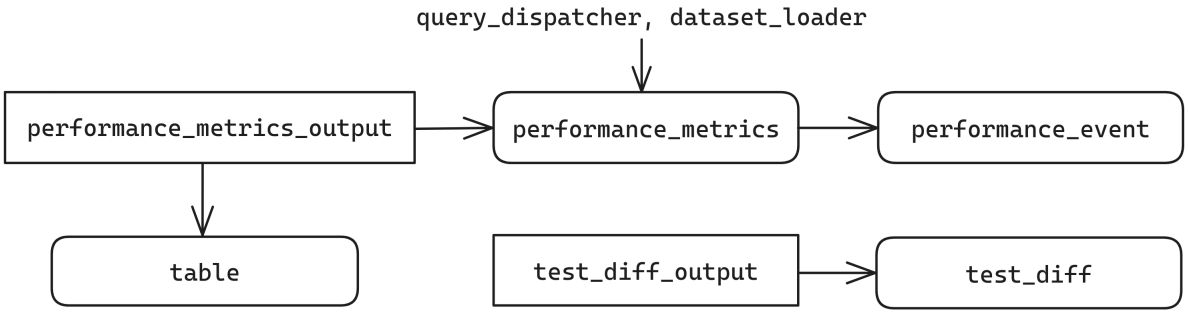


Figura 6: Diagrama de dependências do subsistema de testes

Os testes de desempenho baseiam-se num `performance_event`, uma medição do tempo e da diferença do uso de memória entre dois instantes. `performance_metrics` são um conjunto de eventos de desempenho, produzidos ao longo do decorrer da aplicação, e mostrados ao utilizador no final da sua execução (`performance_metrics_output`). Este módulo apresenta várias tabelas (`table`), relativas ao uso de recursos computacionais na leitura de cada parte do *dataset* e na execução de cada *query*.

Os testes funcionais são implementados por `test_diff`, que calcula as diferenças entre duas diretorias

(resultados esperados *vs* obtidos), que são depois apresentadas ao utilizador por `test_diff_output`. Imagens dos resultados de ambos os tipos de teste encontram-se em *Screenshots* do *output* dos testes.

# 2 Modularidade e encapsulamento

## 2.1 Modularidade

Nesta fase, no âmbito da modularidade, procurámos separar os módulos de I/O e reduzir a duplicação de código relativa aos identificadores das entidades. Ademais, escrevemos um *script* que automaticamente gera um grafo de dependências com todos os módulos. Este permitiu-nos corrigir erros nos diagramas nos relatórios, e encontrar dependências circulares, que prontamente resolvemos.

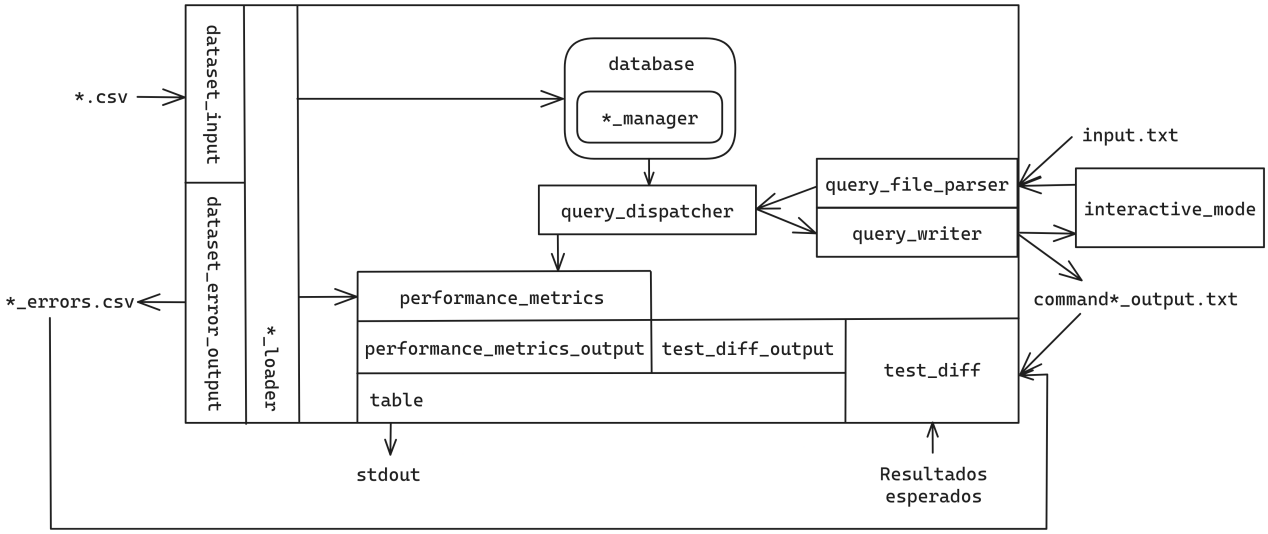


Figura 7: Transferência e armazenamento de dados pelos **principais** módulos da aplicação. Observam-se os novos módulos de I/O.

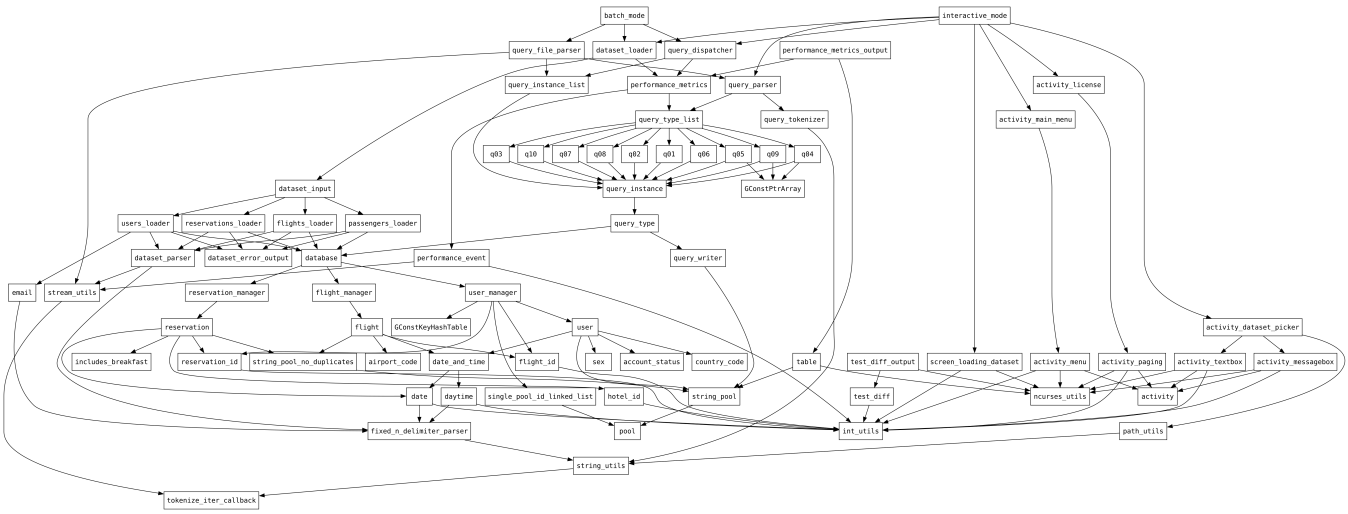


Figura 8: Grafo de dependências de toda a aplicação. Foi gerado automaticamente e não há distinção entre tipos de módulos.

## 2.2 Encapsulamento

### 2.2.1 *The great refactoring*

Apesar de, na primeira fase deste trabalho, termos definido *getters* e *setters* para as estruturas de dados, estávamos cientes da ausência de encapsulamento, pois vários *getters* devolviam apontadores mutáveis para dados no interior das estruturas. Como nunca escrevíamos para estes apontadores, torná-los constantes provou-se a melhor solução para o encapsulamento.

Este processo de aplicar a *keyword* `const` aos argumentos dos métodos e aos seus valores devolvidos provou-se moroso: a cada `const` adicionado, vários módulos deixavam de compilar devido a erros de tipos, causando uma reação em cadeia de mudanças necessárias. No processo de encapsulamento, adicionámos também métodos de cópia profunda (semelhantes aos *copy constructors* de C++) à maioria das estruturas de dados da aplicação. [1]

Apercebemo-nos da importância de uma interface modular bem pensada que dispensa mudanças, já que estas se provaram muito trabalhosas. No modo interativo e no subsistema de testes, como pensámos nas interfaces dos módulos com encapsulamento em mente, não precisámos de gastar tempo na sua correção.

### 2.2.2 Conciliação com alocadores

As únicas estruturas de dados que expõem memória do seu interior são os alocadores `pool` e `string_pool`. Não encaramos isto como um problema para o encapsulamento, pois é algo que qualquer alocador tem de fazer (por exemplo, o método `malloc` expõe memória da arena com que é implementado).

Previamente, durante uma inserção na base de dados, as entidades e os seus campos eram alocados pelos *managers* nas *pools*, e não pelas entidades em si. Resolvemos este problema usando uma estratégia semelhante à utilizada na linguagem Zig: se um método precisa de alocar memória, então leva um alocador como argumento:

```
fn validate(allocator:  Allocator, s:  []const u8) Allocator.Error!bool
```

Este método de Zig valida texto JSON, levando também como argumento um alocador. Na nossa aplicação C, é dada a possibilidade de se usar uma *pool*, ou de se passar o valor `NULL` ao argumento do alocador, para o uso de `malloc`:

```
user_t *user_clone( pool_t         *allocator,
                    string_pool_t  *string_allocator,
                    const user_t   *user);
```

[1] Com exceção de alocadores (*pools*) e estruturas que contêm `FILE *`.

### 2.2.3 Conciliação com a `glib`

O uso de `const` trata-se de um contrato de que um método não irá modificar um objeto. A documentação das estruturas genéricas da `glib` assegura isto.[2] No entanto, estas utilizam `void *` sem o *qualifier* `const`, dado que os programadores podem querer usá-las para armazenar dados mutáveis. A documentação da `glib` incentiva os programadores a usarem apontadores constantes nas suas estruturas, desde que usem o tipo correto quando os reivindiquem.

No entanto, isto é altamente suscetível a erros, pelo que criámos *wrappers* das estruturas da `glib` para apontadores constantes, que lhes retiram o *qualifier* nas inserções, e o reinserem nas consultas. Isto reduz o potencial de erro do programa todo para pequenos métodos facilmente validáveis, de um modo semelhante a conceito de `unsafe` em Rust.

# 3 Desempenho

## 3.1 Melhorias

Quando os *datasets* de maior dimensão foram adicionados à plataforma de testes, já a nossa aplicação apresentava um desempenho excelente. Mesmo assim, procurámos continuar a otimizar o nosso programa.

Para reduzir ainda mais o uso de memória, introduzimos a `string_pool_no_duplicates`, que evita armazenar *strings* repetidas. Ademais, reordenamos os campos nas estruturas de dados das entidades, de modo a reduzir o *padding* e o número de *bytes* ocupados por cada uma.

A maioria do tempo de execução é gasto no *parsing* do *dataset*, onde apenas conseguimos alcançar micro-otimizações nos métodos mais chamados (`strsep` e `int_utils_parse_positive`), determinados com recurso ao `callgrind`. As *queries* são praticamente instantâneas para um utilizador do modo interativo, e responsáveis por apenas uma pequena fração do tempo de execução da aplicação. Logo, procurámos simplificar as suas implementações, em vez de nos focarmos no seu desempenho.

> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.
>
> Knuth, D. E. (1974). Computer programming as an art.
> *Association for Computing Machinery (ACM), 17*(12), 671.
> `https://doi.org/10.1145/361604.361612`

Várias vezes se verificou um melhor desempenho utilizando estruturas de dados mais simples, com

[2] Há algumas exceções (como uma função `GDestroyNotify` ser providenciada), mas garantimos que a nossa solução para este problema não permite nenhuma destas.

pior complexidade assintótica, mas melhor localidade de memória. Foi importante garantir que os catálogos, onde o número de entidades é muito elevado, usavam as estruturas com melhor comportamento assintótico, mas tal não se provou tão relevante para dados auxiliares durante a execução das *queries*.

## 3.2 *Benchmarking*

Para testar o desempenho do programa, utilizámos a versão mais recente da aplicação nos seguintes sistemas:

| | Desktop | Laptop | Mac | Tablet |
|---|---|---|---|---|
| CPU | Intel i3-7100 | Ryzen 7 5800H | Apple M1 Pro | Mediatek Helio P22T |
| Frequência[3] | 3.9 GHz | 4.4 GHz | 3.2 GHz | 2.3 GHz |
| Memória | DDR4 2400 MT/s | DDR4 3200 MT/s | LPDDR5 6400 MT/s | LPDDR4X 4266 MT/s |
| Linux | 6.6.11 (chroot) | 5.15.91 | 5.15.73 (VM) | 4.9.190 |
| Compilador | GCC 12.2.0 | GCC 9.4.0 | GCC 11.4.0 | Clang 17.0.6 |

O número de variáveis entre sistemas é elevadíssimo, pelo que apenas estamos interessados em procurar padrões no desempenho do nosso programa comuns a todos os computadores, e não faremos comparações entre eles.

A aplicação foi executada 10 vezes, com 500 *queries* no *dataset* grande com erros. As execuções mais rápida e a mais lenta foram removidas. A partir das 8 restantes, calculam-se as seguintes médias de tempo e uso de memória:

| | Desktop | Laptop | Mac | Tablet |
|---|---|---|---|---|
| Utilizadores (s) | 0.81 | 0.63 | 0.89 | 5.07 |
| Voos (s) | 0.17 | 0.13 | 0.19 | 0.98 |
| Reservas (s) | 5.41 | 5.51 | 7.11 | 46.1 |
| Passageiros (s) | 4.98 | 4.55 | 6.08 | 36.9 |
| *Queries* (s) | 3.74 | 2.73 | 3.41 | 28.2 |
| Tempo total (s) | 15.3 | 13.6 | 17.6 | 117.8 |
| Pico de memória (MiB) | 935.1 | 935.7 | 935.0 | 945.2 |

## 3.3 Possíveis melhorias

Como mencionado acima, não vemos as *queries* como um potencial local onde procurar otimizações. A parte mais lenta do nosso programa é a leitura de um *dataset*, onde já otimizámos a localidade de memória e a complexidade algorítmica de inserção o mais que fomos capazes. Sem sacrifício da

[3]Em sistemas heterogéneos, a frequência do *cluster* de maior desempenho, dado que o nosso programa apenas usa um núcleo.

modularidade e do encapsulamento, não vemos mais otimizações viáveis que não a paralelização da leitura do *dataset*:

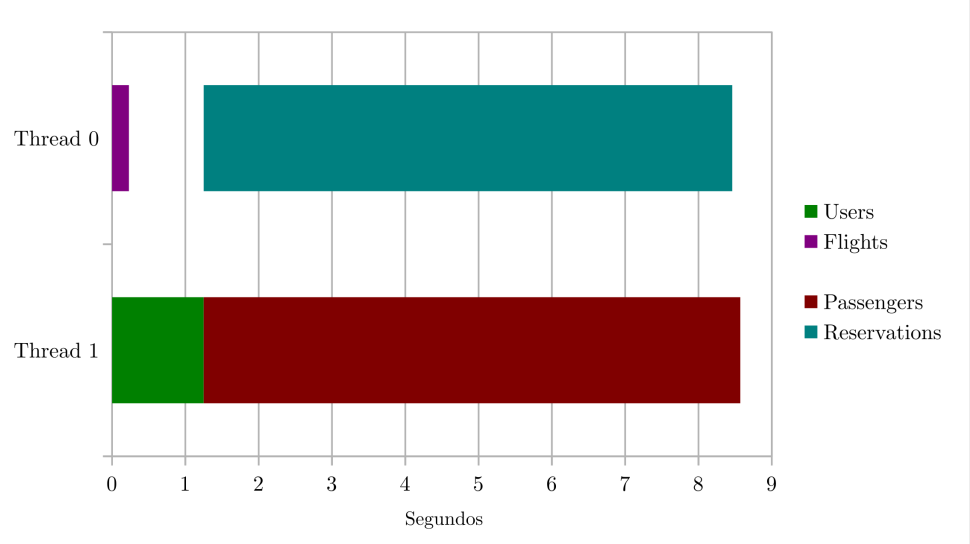


Figura 9: Possível distribuição de tarefas por *threads*, com os tempos da Figura 18

A distribuição de tarefas acima cumpre as condições necessárias para assegurar as relações de dependência de dados (reservas depois dos utilizadores, passageiros depois dos utilizadores e dos voos) e consegue reduzir o tempo de *parsing* de 16 s para um mínimo teórico de 8.6 s! A única dependência de dados que encontrámos, um alocador, é facilmente resolúvel com o uso de dois alocadores distintos para as relações utilizador – voo / reserva.

# 4 Conclusão

Estamos bastante satisfeitos com o resultado final deste trabalho, onde pudemos aplicar conhecimentos de várias UCs. Pensamos que a maior conclusão a tirar deste projeto seja a importância de se pensar bastante antes de se escrever qualquer programa. Esta filosofia aplica-se ao desenvolvimento de aplicações modulares e encapsuladas, à escolha da *schema* de uma base de dados para o melhor desempenho, à arquitetura global de um programa para este ser facilmente extensível, *etc.* O desenvolvimento destes 75 módulos permitiu-nos aprofundar o nosso conhecimento em C, o que por sua vez nos capacitou para implementar conceitos de outras linguagens e paradigmas, para suprir certas lacunas de C: polimorfismo, tipos genéricos, alocadores customizados, *etc.*

# 5 Anexos

## 5.1 *Screenshots* do modo interativo


Figura 10: Menu (`activity_menu`), em particular, o menu principal (`activity_main_menu`)

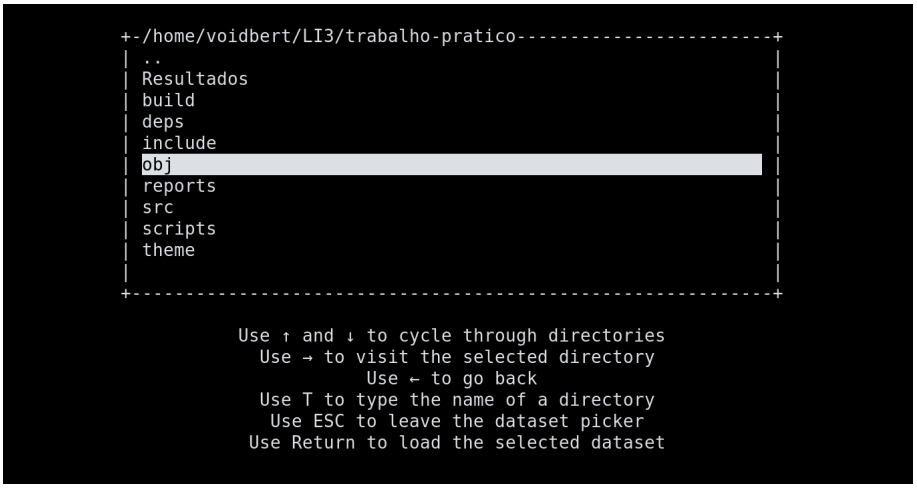


Figura 11: Navegador de ficheiros para a escolha de um *dataset* (`activity_dataset_picker`)


Figura 12: Ecrã durante a leitura de um *dataset* (`screen_loading_dataset`)

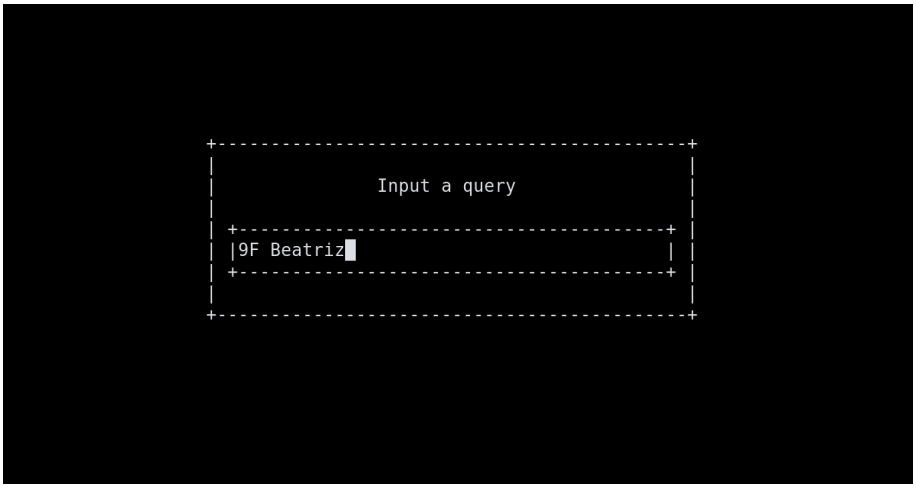


Figura 13: Caixa de texto para a entrada de uma *query* (`activity_textbox`)


Figura 14: Output paginado de uma *query* (`activity_paging`)


Figura 15: Licença da aplicação (`activity_license`)

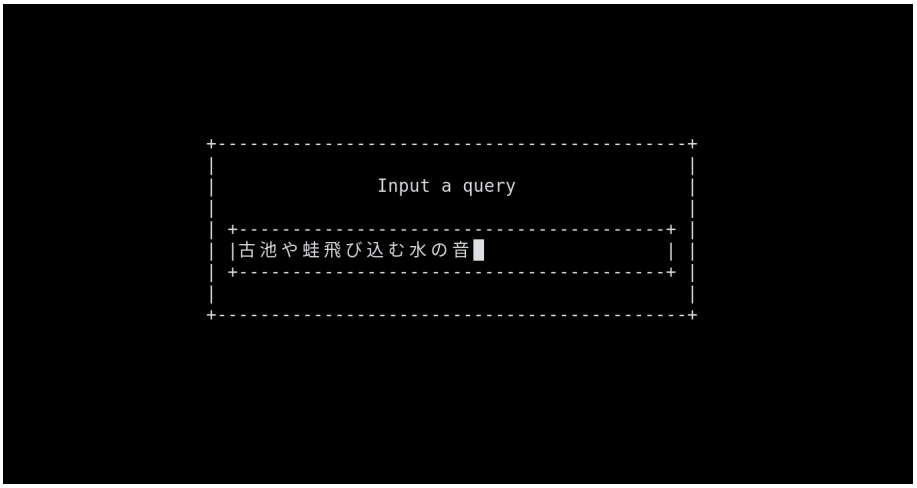


Figura 16: Suporte para caracteres de largura dupla numa `activity_textbox`. O *scrolling* mantém-se funcional.

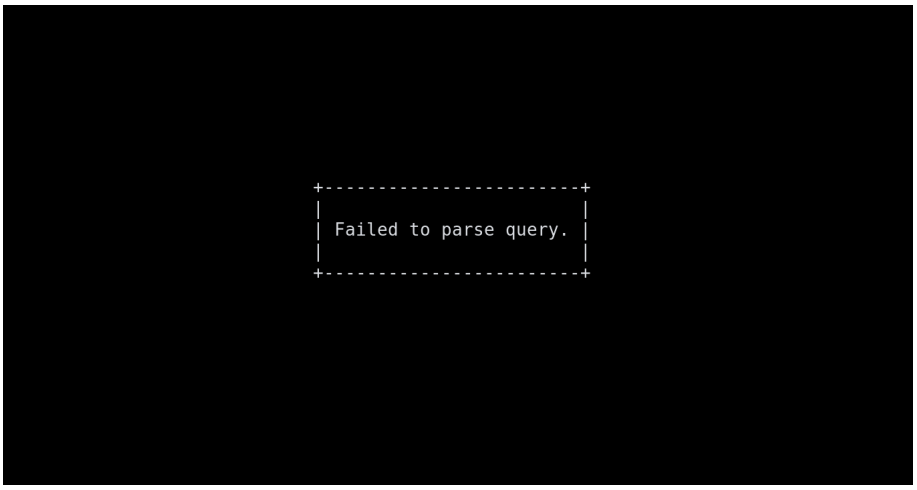


Figura 17: Aviso de erro numa `activity_messagebox`

## 5.2 *Screenshots* do *output* dos testes

DATASET LOADING

| | Time (s) | Memory (MiB) |
|---|---|---|
| Users | 1.25 | 140.80 |
| Flights | 0.23 | 14.63 |
| Passengers | 7.32 | 228.34 |
| Reservations | 7.21 | 539.40 |

Figura 18: Tabela de tempos de leitura de cada parte de um *dataset*

QUERY STATISTICAL DATA GENERATION

| | Time (ms) | Memory (KiB) |
|---|---|---|
| Query 1 | - | - |
| Query 2 | - | - |
| Query 3 | 130.41 | 4.00 |
| Query 4 | 159.04 | 3360.00 |
| Query 5 | 12.60 | 0.00 |
| Query 6 | 16.43 | 0.00 |
| Query 7 | 21.62 | 792.00 |
| Query 8 | 256.59 | 0.00 |
| Query 9 | 154.80 | 0.00 |
| Query 10 | 5755.82 | 0.00 |

Figura 19: Tabela de tempos de geração de dados estatísticos para cada *query*

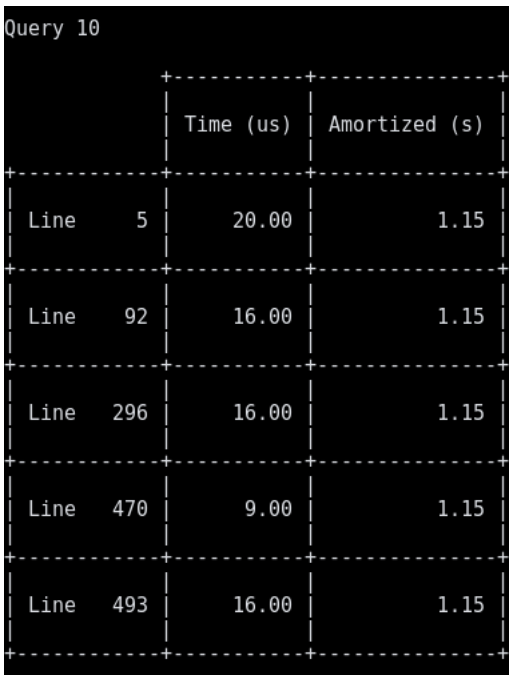


Figura 20: Tabela de tempos de execução de cada *query*. É visível como são automaticamente escolhidas duas unidades diferentes para as duas colunas.

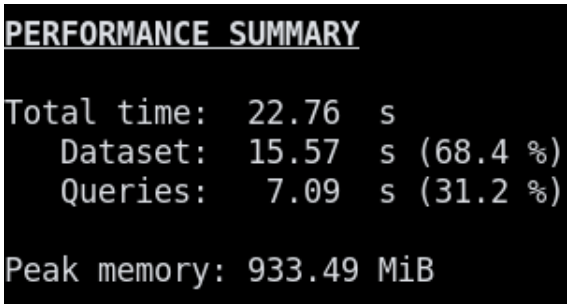


Figura 21: Sumário do *benchmarking* da aplicação

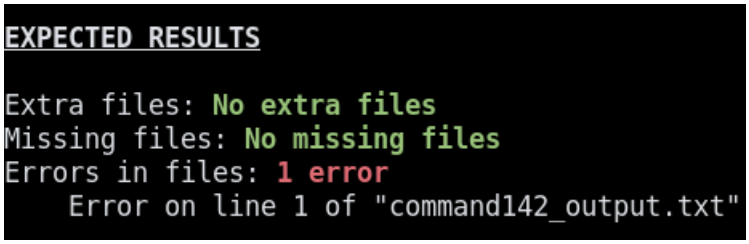


Figura 22: Exemplo de resultado de testes funcionais com um erro