Universidade do Minho
Relatório Laboratórios de Informática III - fase 2

A100535, Gonçalo Nuno da Silva Loureiro
A100538, Ivan Sérgio Rocha Ribeiro
A100709, Pedro Miguel Meruge Ferreira

Índice

Modularidade e encapsulamento

O projeto ficou dividido nos seguintes módulos:
- “main.h”: executar o programa chamando os outros módulos; tem implementações diferentes para o programa-principal e programa-testes
- “files.h”: abrir os ficheiros de dados (e input para queries no modo batch)
- “query_requests.h”: identificação do modo batch ou modo interativo; funcionamento do modo interativo
- “userdata.h”: parsing e criação de estruturas para users e estatísticas de users
- “driverdata.h”: parsing e criação de estrutura para drivers
- ”ridesdata.h”: parsing e criação de estruturas para rides e estatísticas de rides
- “query_common_funcs.h”: funções de print final, partilhadas pelas Q2,Q7 e Q9
- “common_parsing.h”: funções utilizadas frequentemente para parsing de dados
- “statistics.h”: módulo de construção de estruturas que requerem acesso a mais do que um módulo de parsing dos dados (usersdata,drivedata,ridesdata).

Os módulos interagem entre si com recurso a funções “*get*” para aceder a estruturas, structs e parâmetros criados no seu interior.

O módulo “ridesdata.h” contém muitas estruturas de dados relativas a estatísticas, porque colocá-las noutro ficheiro implicava uma perda significativa na eficiência do módulo. Essas estruturas apenas acedem a dados exclusivos das rides, necessitando de acesso direto aos campos das structs criadas. Foi considerada a hipótese de utilizar “*private header files*”, mas não foi seguida.

Estruturas de dados

As estruturas primárias para os users, drivers e rides mantiveram-se idênticas às da primeira fase. Apenas alteramos os arrays das rides para ser dinâmicos ,usando a Glib, para acomodar um tamanho variável de dados, de grandeza maior. Todas as referências a memória ocupada nesta secção são referentes ao *dataset* grande sem erros, porque tem número de users, drivers e rides fáceis de usar em cálculos.

Users

hashtable da glib. Acesso em tempo constante à informação.

Drivers

array 2D. O array principal é um GPtrArray (dinâmico) que aponta para sub-arrays estáticos, de tamanho definido numa macro “SIZE”. Achamos melhor fragmentar em sub-partes o array. Assim, reallocs apenas afetam o array principal e não toda a informação. Como aspeto negativo ocupa ligeiramente mais espaço (8bytes/pointer, para cada posição do array principal) e perde localidade espacial.

Rides

array 2D. Estrutura idêntica aos drivers, para não ter de alocar (38bytes/ride *10.000.000 rides) 380MB seguidos em memória (para além dos reallocs).

Estatísticas

Criaram-se outras estruturas para obter estatísticas pedidas nas queries. Estruturas das queries são construídas e guardadas na struct *RidesData* do módulo "ridesData.h" e em UserData do módulo "userdata.h".

Q6 e Q4

Constrói-se uma *hashtable* que a cada cidade faz corresponder a struct em baixo.

```
struct CityRides {
    GPtrArray * cityRidesArray;
    partialDriverInfo * driverSumArray;
    int distance[3], total[3];
};
```

O *cityRidesArray* contém pointers para rides, e está ordenado com base em datas. Isto permite procurar eficientemente por rides numa determinada cidade e num dado intervalo de tempo. O mais importante com este array em específico era conseguir obter rides num intervalo de datas para uma certa cidade, o que esta estrutura permite fazer, porém pode ser usada para percorrer todas as cidades.
Os arrays distance[] e total[] acumulam distâncias e números de viagens por *carClass*, numa dada cidade, usados depois na Q4, que apenas tem de executar uma simples conta. A conta podia ser feita e guardada no fim da criação da estrutura, mas é tão rápida e há tão poucas cidades que não haveria diferença de performance.
A estrutura tem mais um pointer para um arra porque foi reaproveitada para a query 7, descrito em baixo.

Q1 e Q7

Array estático de structs, com tamanho igual ao número de drivers (ver structs abaixo). A cada posição x do array corresponde o ID do driver x, para acessos constantes. A Q1 tem apenas um array associado, com informação global de todos os drivers. A Q7 tem um array para cada cidade com informação exclusiva dessa cidade. Reutiliza-se a hash table criada para as Q4 e Q6, guardando na *struct* de cada cidade um destes arrays.

Vantagens: Um array estático evita *reallocs*; em relação a um pointer array, ocupa menos memória (evita-se memória dos pointers) e pode ser libertado mais rapidamente; beneficia de localidade espacial. O array é pré-alocado com um calloc grande (40bytes/struct * 100.000 drivers) de 40MB no caso da query1, e (24bytes/struct * 100.000drivers) de 24MB para cada uma das 7 cidades, no caso da query 7.Esta solução apresentou-se melhor, de várias alternativas testadas (apresentadas na secção de desempenho)

Desvantagens: ocupar muita memória consecutiva, aumentando linearmente com o número de drivers. No entanto, memória disponível não apresentou uma limitação. No caso de haver uma grande quantidade de drivers sem rides, inválidos ou inativos, ocupa-se memória que não contém informação pertinente de drivers. No entanto, analisando os *datasets*, a proporção desse drivers é reduzida em relação aos drivers com informação (aproximadamente 2.000/100.000 = 2% no dataset large-errors).

```
typedef union {
   unsigned int chart[5];
   double average;
} ratingsUnion;
```

```
struct fullDriverInfo {
  ratingsUnion ratings;
  int driverNumber;
  short int ridesNumber,
        totalTravel;
  double tips;
Date mostRecRideDate;
};
```

```
struct partialDriverInfo {
   ratingsUnion ratings;
   int driverNumber;
};
```

- *"fullDriverInfo"* tem informação para a Q1, mas também um parâmetro com a data mais recente, para reaproveitar esta estrutura para a Q2.
- *"partialDriverInfo"* tem informação para cada cidade da Q7.

Otimizações: Guarda-se num array o número de ratings de cada tipo que o driver recebe. Este espaço é reaproveitado para guardar um double, usando uma *union*. Evita-se, assim, *mallocs* e *frees* desnecessários de um *void **, ou parâmetros extra na struct. Não exploramos aproveitar os bits restantes na union para guardar outros valores da struct, ao mesmo tempo que o double, com *bitshifts*.
Como cada cidade tem de acumular avaliações de drivers para a Q7, não fazia sentido repetir os cálculos para a avaliação global dos drivers na Q1. No fim de completar a leitura das rides, no mesmo loop em que são calculadas as médias de cada cidade, são passados os valores finais de cada driver, em cada cidade, para o array da Q1, reduzindo os cálculos.

Q2
Pointer array estático, com tamanho igual ao número de drivers. A cada pointer deste array corresponde a struct na mesma posição do array da Q1. Este array precisa de ser ordenado segundo os parâmetros da Q2. Daí, usa-se apontadores para a estrutura já criada da Q1 e organiza-se apenas os apontadores.

Vantagens: ocupa-se apenas (8bytes/driver * 100.000 drivers) 8MB e ordena-se pointers que apontam para zonas consecutivas de memória (as structs do array da Q1), beneficiando de localidade espacial no sort. Com esta estratégia, evita-se ocupar memória e tempo a copiar o array da query 1 para um novo array estático, ou a criar um array com parâmetros em comum com a query1, o que envolveria cálculos repetitivos.

Desvantagens: Estrutura pouco robusta, depende dos valores do array da Q1

Q3

Na Q3, pegamos no array com os users e damos-lhe sort em função das suas distâncias percorridas, comparando-as entre eles. Esta informação é guardada nas structs que já está dentro da hashtable. Por fim, pegamos nessa informação e devolvemos os N users com maior distância viajada.

Q8

Na realização da Q8 optamos por dividir o array que tem todas as rides em dois arrays mais pequenos separando-os de forma a que exista um array em que ambos os drivers e users sejam do género masculino e outro em que sejam do género feminino. Decidimos fazer isto, pois percebemos que esta estratégia iria diminuir muito o tamanho do array que continha todas as rides, aumentado assim a performance do nosso trabalho.

Q9

GPtrArray, necessário para ser dinâmico.
Reaproveita-se a hash table criada para a Q4 e Q6 (com rides ordenadas por data, em cada cidade) e percorre-se cada cidade adicionando ao array as rides que entram no critério da query, com recurso a *binary search*.

Vantagens: Todo o espaço ocupado pela estrutura é necessário, só aponta para rides pertinentes para esta query. A query só costuma ser "chamada" num intervalo de datas reduzido, pelo que o tamanho do array é reduzido, logo criar e destruir o array não é tão custoso. Número de cidades é reduzido, pelo que a iteração entre cidades não é muito custosa. O tamanho desta estrutura é reduzido, apenas pointers (8bytes/ride). Reaproveita-se estruturas já criadas.

Desvantagens: Query faz cálculos repetitivos, se for chamada duas vezes para o mesmo intervalo de tempo, pois repete a iteração entre cidades. O GPtrArray é sucessivamente criado e destruído, cada vez que a query é executada. Podia-se criar um GPtrArray global com todas as rides ordenadas por data (fazendo merge das rides já previamente ordenadas de cada cidade, por exemplo), o que escusava a iteração entre cidades para procurar rides. No entanto, não consideramos que a memória ocupada por essa estrutura, (8bytes/rides * 10.000.000 rides) 80MB, justificasse o tempo que era poupado a percorrer um só array, em vez de várias cidades.Grande parte das rides guardadas ness estrutura seriam inutilizadas, porque só são pedidos intervalos de tempo muito reduzidos para a Q9.

Q5

Não foram criadas estruturas novas, mas como era necessário iterar por todas as rides num intervalo de datas, decidimos reaproveitar as estruturas de rides por cidade da Q4 e Q6, de modo semelhante à Q9, devido ao facto de estas estarem ordenadas por data.
Vantagens: evitar um sort "pesado" e uma estrutura nova para guardar milhões de rides.
Desvantagens: Iterar pela hashtable das cidades e fazer binary search em cada cidade para encontrar o intervalo de datas pretendido.

Dificuldades

Entradas inválidas

O desafio principal era parar o "parsing" de uma struct a meio, assim que fosse detetado que era inválida. Para além disso, teríamos de saber exatamente onde o erro ocorreu para poder dar free aos campos que já foram criados. Na estrutura que tínhamos em cada função de parsing dos drivers, users e rides, tal não era possível. Solução: estas funções passam a chamar uma função comum que se encarrega do parsing, detecção de erros e dos frees, passando lhe uma struct que indica o formato desejado, incluindo, para cada campo da struct: a função a usar, se deve ser freed ou não, e o seu offset, visto que a tal função recebe as structs como sendo *void** anónimos (preenche o pointer base + offset com base na função pretendida). Assim, esta tarefa relativamente complexa fica uniformizada com uma única função, que preenche cada campo da struct com a função devida, comunicando através do seu return se a struct lida foi inválida ou não.

Mais tarde, ao mudar para fread() (explicado em performance), conseguimos encontrar uma forma de manter esta estrutura apesar de outros problemas: era preciso ler uma quantidade fixa de bytes para o buffer, sendo que podia, por exemplo, acabar a meio de uma linha e não num '\n'. Além disso, o buffer teria de ser *refreshed* periodicamente. Conseguimos adaptar a estrutura que já existia, porém para cada call de parsing de uma struct, há *checks* sobre o *buffer* que têm de ser feitos: a função usa um base pointer (bp) e stack pointer (sp) para o buffer, para detetar se está vazio ou se todo ele já levou parse. Ao dar *refresh*, detetamos a *newline* mais próxima do fim, e revertemos o fread desses últimos caracteres. O bp é incrementado por cada função de parse para refletir o número de caracteres lidos. Isto é relativamente pesado para uma função chamada milhões de vezes, mas melhorar iria implicar um *rework* da estrutura do projeto demasiado grande. Mais pormenores descritos em performance.

Tivemos também dificuldade em decidir como marcar uma entrada como inválida sem ocupar mais memória nem prejudicar a estrutura dos gets que teriam de ser alterados, mas acabamos por decidir que, no *lookup*:

Users: pointer devolvido é inválido se for NULL (*lookup* da hashtable devolve NULL).

Rides e Drivers: o primeiro pointer da struct é NULL (assim evitamos ter de fazer memset a 0 da struct inteira, por exemplo).

Datas

Ao transformar as datas num *int* (descrito em performance), embora estivesse claro de que era possível, os resultados não estavam a bater certo, usando unions ou cast de uma struct com {year, month, day} para int. Descobrimos que tal se deve ao *padding* e *aliasing* que o C faz das structs, e que a maneira correta de resolver o problema seria com *bit shifts*.

Performance

Alterações

Usando profilers de performance e o script de testes, identificamos bottlenecks e ineficiências no projeto.

- O parsing inteiro foi alterado, para usar fread() em vez de fgetc(), que melhorou significativamente a performance, embora tenha outros problemas, já descritos em dificuldades. A função de parse mostrou-se pesada tendo em conta a quantidade de vezes que é chamada, mas já estava "enraizada" em todo o projeto e continua a mostrar muito melhor performance que a versão antiga.
- Devido a uma sugestão dos docentes, como o array de cada cidade está ordenado por data, criamos uma binary search modificada que permite encontrar os índices de um intervalo de datas, para evitar um loop por todas as rides de uma cidade. É "modificada" porque as datas podem não existir, e pode haver mais do que uma data igual, o que por si só tornou a função complexa. No final, permitiu evitar loops a usar get…Date() e comparações constantes. Originalmente permitiu evitar *strdup* que é muito lento (mais tarde deixou de ser *strdup*, mas continua a ser um get desnecessário), permitindo que as queries que a usam corram cerca de 1000x mais rápido, pois também não têm de iterar pelas rides todas, mas apenas fazer um loop entre índices '*start*' e '*end*'.
- Alteramos os tamanho dos arrays secundários para um tamanho mais eficiente (testamos tamanho vs performance), bem como do buffer principal.
- Para permitir um lookup por ID mais eficiente, visto que drivers e rides são um array 2D, este tamanho foi alterado novamente para a potência de 2 mais próxima, para permitir fazer cálculo de remainder com bitshift.
- As datas passaram a ser números inteiros, em que os primeiros 16 bits são o ano, os 8 seguintes o mês e os restantes os dias. Isto permite funções de comparação extremamente rápidas (é só subtrair as datas) e cada data ocupa o espaço de um int. Extrair o dia, mês e ano é feito com bit shifts e é pior do que ter acesso aos mesmos diretamente, mas só é usado em prints das queries.
- Eliminamos comentários das rides e cidades dos drivers, visto que não são usados. Nas rides também poderíamos ter apagado, porém visto que a cidade tem de ser parsed para atualizar as estruturas da cidade correspondente, e como o parsing é feito noutro módulo, acabamos por manter as cidades nas suas structs.

Testes de estruturas

Para chegar às estruturas finais da Q1 e Q7 testaram-se diferentes implementações.[1] Testes com o dataset large-errors, com recurso ao script implementado para o "make testes". No caso do macOS, a leitura de memória dá erro no script, por isso usou-se o comando "/usr/bin/temp -v ./prog…", e anotou-se o parâmetro "Maximum resident set size".

[1] Os valores testados para a estrutura que se escolheu no final não coincidem com os valores obtidos na versão atual do trabalho. Estes testes foram desenvolvidos em fases anteriores da escrita da fase 2.

PC1: Lenovo Y520, i7-7700HQ, Ubuntu 22.04
PC2: MacBook Pro (2020), M1, macOS Monterey versão 12.5.1

Estruturas Cálculos	1. Pointer arrays - Q1 e Q7, sem callocs de structs	2. Array - Q1, pré-"calloced"; pointer arrays - Q7, sem callocs de structs	3. Array - Q1, pré-"*calloced*"; pointer arrays - Q7; pequeno calloc em cada struct	4. Array - Q1, pré-calloced; pointer arrays - Q7, "calloc" em array grande adjacente	5. Array - Q1 e Q7, pré-"calloced"
A. Acumular todas as informações de drivers nas cidades e passar para o array global	**PC 1** Load:10.416s Free:0.764s Memory:1388MB **PC 2** Load:6,914s Free:1,631s Memory:1181MB	**PC 1** Load:10.377s Free:0.759s Memory:1386MB **PC 2** Load:6,919s Free: 1,605s Memory:1178MB	**PC 1** Load: 10.315s Free: 0.669s Memory: 1386MB **PC 2** Load:6,902s Free:1,595s Memory:1173MB	**PC 1** Load: 10,381s Free:0,628s Memory:1375MB **PC 2** Load:6,897s Free:1,576s Memory:1176MB	**PC 1** Load:9.609s Free:0.624s Memory:1369MB **PC 2** Load:6,730s Free:1,570s Memory:1170MB
B. Acumular parte das informações de drivers num array global e em cada cidade	**PC 1** Load: 11.191s Free: 0.783s Memory: 1377MB **PC 2** Load:6,943s Free:1,654s Memory:1184MB	**PC 1** Load:11.103s Free:0.757s Memory:1371MB **PC 2** Load:6,916s Free:1,621s Memory:1177MB	**PC 1** Load: 10.972s Free:0.661s Memory:1371MB **PC 2** Load:6,891s Free:1,601s Memory:1168MB	**PC 1** Load:10.935s Free:0.622s Memory:1360MB **PC 2** Load:6,881s Free:1,538s Memory:1174MB	**PC 1** Load:9.623s Free:0.619s Memory:1355MB **PC 2** Load:6,715s Free:1,528s Memory:1159MB

Conclusões: Alocar arrays de apontadores com todas os apontadores a NULL e inserir structs numa posição x quando havia informação de um driver de ID x não era a solução mais eficiente. Havia necessidade de verificar constantemente se já existia struct numa posição ou não. Analisando o dataset dado, não fazia sentido este modelo, pois a grande maioria das posições nos arrays eram ocupados (como referido na secção estrutura de dados, apenas 2% dos drivers não tinham rides associadas). Implementar arrays estáticos resultou em menos memória ocupada (8 bytes/pointer para struct); menor tempo de execução com a remoção das verificações constantes; mais localidade espacial, porque eram alocados blocos contínuos; "frees" mais rápidos. O "sort" dos arrays da Q7, para cada cidade, também se mostrou significativamente mais rápido num array estático que num array de pointers, devido à localidade espacial, supomos. No fim, implementamos um array global estático na Q1 e arrays estáticos na Q7 para cada cidade, pré-alocados.

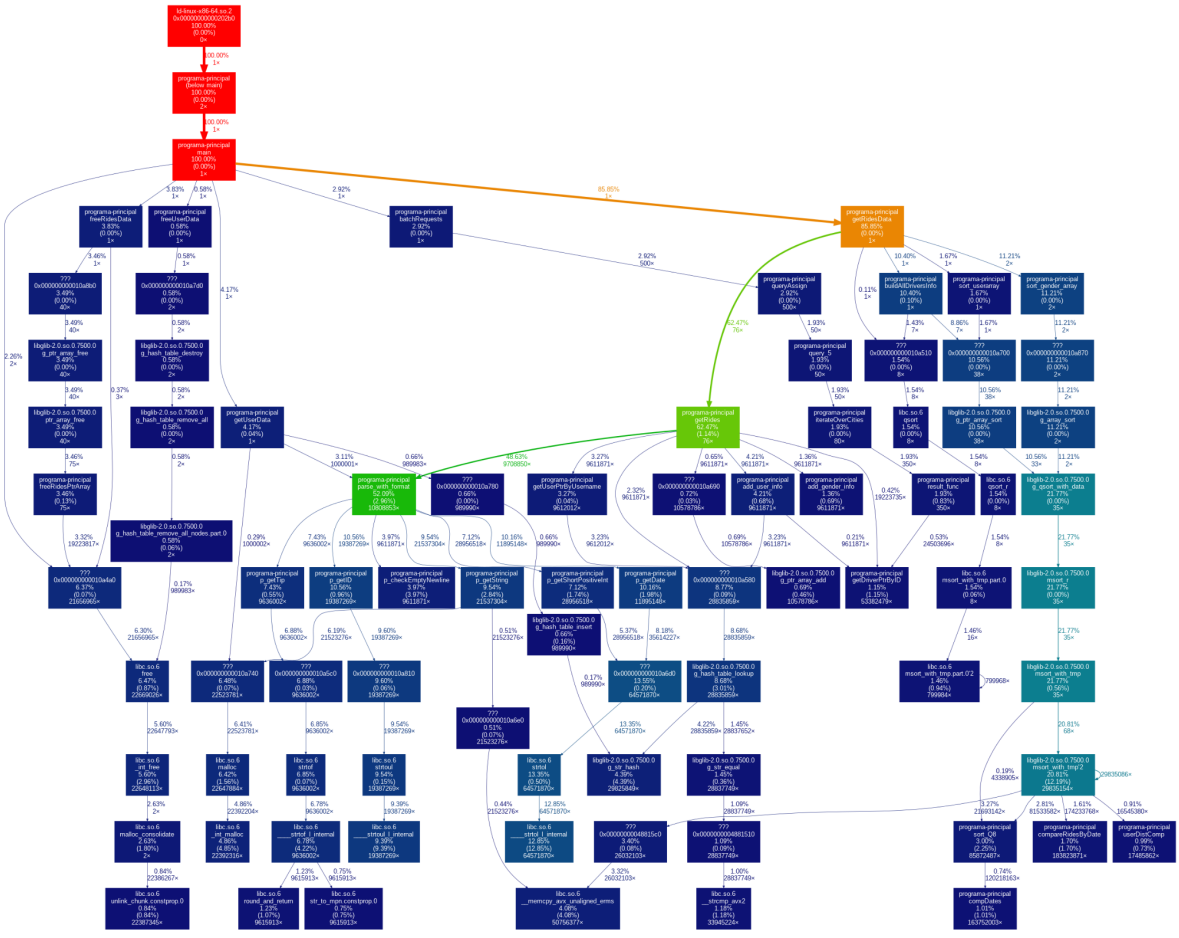
Módulos de testes

Os módulos de testes estão localizados em “src/testes/” e “include/testes/”. Executando “make testes” é gerado um executável com os módulos base + módulos de testes. Porém, qualquer módulo com o mesmo nome é substituído pelo de testes. Assim, por exemplo, os testes usam um ficheiro ‘main’ diferente do normal (usam o main contido em src/testes/main.c).

Utilizando ./programa-testes <args> com 1 argumento (path do Dataset), é testado o tempo de load e free de dados. Com 3 argumentos, respetivamente path para Dataset, path para ficheiro de input, e path para ficheiros “command<x>_output.txt”, para poderem ser comparados, são corridos testes para cada query: verificamos se o seu output está correto, e corremos cada input 10x. No fim, adicionamos o tempo médio dessas 10x a um acumulador. No final de todas as queries, é calculada a média ‘global’ de cada query, ou seja, média das médias, usando este acumulador.

Vários dados, tanto do modo com 1 argumento como com 3, são passados para ficheiros de output para poderem ser utilizados por um script ou vistos com mais detalhe. O script usado é ./Scripts/test.sh, que mostra: tempo (média das médias) de cada query, tempo médio de load e free (testado 5x), e tempo e memória utilizados globalmente (testados 1x).

Os bottlenecks foram testados usando “callgrind” e um gerador de gráficos a partir do seu output. Decisões para melhorar a performance foram baseadas nos gráficos e no script de testes.



PC1: Lenovo Y520, i7-7700HQ, Ubuntu 22.04
PC2: MacBook Pro (2020), M1, macOS Monterey versão 12.5.1
PC3: MackBook Air (2017), i5 1,8GHz, macOS Monterey versão 12.6.1

Dataset: **Large-errors** Load in (s) Free in (s) Memory in (MB) Queries in (s)	**PC1** Load:11.6623 Free:0.633 Memory: 1558 Q1: 0.000001 Q2: 0.000438 Q3: 0.000461 Q4: 0.000001 Q5: 0.063494 Q6: 0.001914 Q7: 0.000270 Q8: 0.006778 Q9: 0.000579	**PC2** Load: 7.99786 Free:1.57007 Memory: 1581 Q1: 0.000001 Q2: 0.000368 Q3: 0.000307 Q4: 0.000001 Q5: 0.009466 Q6: 0.000946 Q7: 0.000233 Q8: 0.003979 Q9: 0.000387	**PC3** Load:14,1503 Free:2,13456 Memory: 1631 Q1:0,000001 Q2:0.000459 Q3:0.000472 Q4:0.000001 Q5:0.071134 Q6:0.002103 Q7:0.000290 Q8:0.007013 Q9:0.000598

Modo interativo

Utilizamos a biblioteca <ncurses.h>, segundo a indicação dos docentes. O caminho dos dados de inputs é feito ainda no terminal. Se o tamanho do terminal for menor que o tamanho padrão em terminais Unix (80,24), aparece uma mensagem de erro e o programa termina. Caso contrário, a tela inicial do modo interativo aparece.
Não há deteção de input do rato, só do teclado. Usa-se o mesmo método de paginação e interação com as páginas tanto para a secção de queries, como para o painel de ajuda.
Na secção das queries, cada linha de output de uma query é separada em colunas. O espaçamento entre colunas é proporcional ao tamanho do terminal.
Na secção de ajuda, há 3 linhas de informações para cada query. Se a informação relativa a um query não couber toda na página atual, é passada na totalidade para a página seguinte, para melhor legibilidade.

Implementações sugeridas na fase 1

- Na Q1 existiam duas funções que calculavam datas, agora existe apenas uma que faz os cálculos tanto dos *users* como dos drivers.
- Na Q1 a maioria dos cálculos para users que eram feitos dentro do ficheiro da própria, passaram a ser feitos no ficheiro *"userData.c"*.
- Queries recebiam sempre um número fixo de 3 *char** de input, ainda que não os utilizassem todos. Agora recebem um array de *char**, utilizando só as strings que precisam.
- Sort da Q2 passou a ser executado apenas uma vez no ficheiro "*RidesData.c*" (tal como todas as outras estruturas com sort implementadas nesta fase).

- Procura de datas, passou a ser feita com BSearch(modificada) em vez de loop do início ao fim do array, como descrito em performance.
- Array principal das rides passou a ser dinâmico, com recurso à glib, para acomodar número variável de dados em escala de grandeza maior (arrays dos users e drivers já eram dinâmicos).