USO DO DUCKDB COM R

Thiago de Oliveira Pires

Muitos já tiveram o problema, principalmente relacionado a memória da computador, quando tentam ler e manipular uma base de dados muito grande. Existem várias soluções para lidar com grandes bases de dados, mas uma tem chamado bastante atenção atualmente: `duckdb`. O `duckdb` é um banco de dados simples de operar, por ser um banco de dados embarcado/embutido no estilo SQLite. Contudo, diferentemente do SQLite, o `duckdb` tem suporte para mais de 20 tipos de variáveis. E alguns destes tipos são aninhados como lista, estrutura e *map*. Pode se fazer consultas em arquivos csv e parquet diretamente. Consultas até em arquivos online salvos em um serviço de armazenamento na núvem (e.g. S3). Tem suporte também para consultas em arquivo json. Além de toda sintaxe padrão do SQL, ele tem algumas funções adicionais e suporta até a sintaxe de list comprehention do python. Este capĩtulo irá apresentar alguns recursos do `duckdb` e sua interação com a linguagem R.

Palavras Chave: duckdb; SQL; R; banco de dados.

1 Introdução

O `duckdb` é um sistema de gerenciamento de banco de dados (SGBD) de código aberto (i.e. MIT), escrito em C++ e otimizado para consultas analíticas. Ele foi projetado para fornecer alta velocidade e eficiência em consultas complexas em grandes conjuntos de dados.

Existem duas características principais que distinguem o `duckdb` de outras ferramentas de análise de dados: a arquitetura colunar e o processamento vetorizado. A arquitetura colunar permite armazenar dados em colunas separadas, otimizando o acesso aos dados necessários para consultas específicas. Já o processamento vetorizado realiza operações em lotes, aproveitando as otimizações de hardware e reduzindo a latência de acesso à memória. Com isso se produz consultas mais rápidas e eficiêntes.

Outra vantagem do `duckdb` é a sua capacidade de compressão de dados. Ele utiliza algoritmos de compressão especializados, resultando em economia de espaço em disco e também contribuindo com uma maior velocidade na consulta.

Existem várias *API wrappers* em outras linguagens além de C++ que poderão interagir com o `duckdb`. Além disso, é possível rodar o `duckdb` no próprio *browser* utilizando uma versão compilada em Web Assembly.

Todas estas caractéristicas aliada a facilidade de instalação tornam o `duckdb` uma ferramenta bastante robusta para a análise de grandes bases de dados.

2 Simplicidade

Para usar o `duckdb` no `R` é bastante fácil, tendo uma instação bem simples como vemos à seguir:

```
install.packages("duckdb")
```

Caso necessite de uma versão em desenvolvimento do pacote, a instação poderá ser feita indicando o repositório do `duckdb`:

```
install.packages('duckdb', repos=c('https://duckdb.r-universe.dev',
  'https://cloud.r-project.org'))
```

A versão estável utilizada neste capítulo foi a 0.7.1.1.

3 Velocidade

Nesta seção, vamos investigar a impressionante velocidade do `duckdb` na análise de dados. Para ilustrar esse desempenho, faremos uso de um conjunto de dados que registra a duração de viagens de táxi na cidade de New York. Esses dados foram obtidos a partir da plataforma Kaggle e estão disponíveis no formato `csv` através do link https://www.kaggle.com/competitions/nyc-taxi-trip-duration/data. O conjunto de dados compreende cerca de 1.5 milhões de registros de viagens com o arquivo tendo o tamanho de 191Mb.

Utilizando apenas o `r-base` e o pacote `dplyr` para criar uma variável com os meses, aplicar um `group_by` e um `summarise` para calcular a média, o tempo foi de aproximandamente 22 seg.

```
system.time({
  read.csv("../data/nyc-taxi-trip-duration/train.csv") |>
    dplyr::mutate(month = lubridate::month(dropoff_datetime)) |>
    dplyr::group_by(month) |>
```

```
    dplyr::summarise(`Média (s)` = mean(trip_duration, na.rm = TRUE))
})
```

```
usuário   sistema decorrido
 14.323     6.096    21.711
```

Contudo, utilizando o `duckdb`, aplicando a mesma análise, o tempo de execução da leitura dados e consulta foi um pouco maior do que 2 seg somente.

```
system.time({
  con <- duckdb::dbConnect(duckdb::duckdb(), "../data/nyc-taxi.duckdb")
  duckdb::duckdb_read_csv(con,
    "nyc-taxi", "../data/nyc-taxi-trip-duration/train.csv")
  dplyr::tbl(con, "nyc-taxi") |>
  dplyr::mutate(month = dplyr::sql("datepart('month',
    strptime(dropoff_datetime, '%Y-%m-%d %H:%M:%S'))")) |>
  dplyr::group_by(month) |>
  dplyr::summarise(`Média (s)` = mean(trip_duration, na.rm = TRUE))
  duckdb::dbDisconnect(con, shutdown = TRUE)
})
```

```
usuário   sistema decorrido
  2.024     0.145     2.331
```

O resultado é a média de duração de viagens por mês:

```
# Source:   SQL [7 x 2]
# Database: DuckDB 0.7.1 [root@Darwin 22.6.0:R 4.3.0/../data/nyc-taxi.duckdb]
  month `Média (s)`
  <dbl>       <dbl>
1     3        958.
2     1        918.
3     4        963.
4     5       1001.
5     6       1012.
6     2        898.
7     7       9484.
```

4 Recursos

4.1 Tipos de dados

No `duckdb` há mais de 20 tipos de dados suportados. A lista completa pode ser consultada neste link https://duckdb.org/docs/sql/data_types/overview. A seguir poderá ser observado alguns dados que foram salvos em alguns tipos no `duckdb` e como os tipos foram preservados ao se fazer a consulta na tabela `dplyr::tbl(con, "examples")`. Uma observação interessante é do tipo `factor` no `R`, e como ele persiste na tabela do `duckdb` como `enum`. Contudo, se esta tabela for lida novamente no `R`, o tipo `factor` permanece.

```
con <-
  duckdb::dbConnect(duckdb::duckdb(), ":memory:")

dplyr::tibble(
  boolean = c(TRUE, TRUE, FALSE, TRUE),
  double = c(-1.2, 5.65, 0.91, 100),
  integer = c(3L, 20L, 0L, -2L),
  timestamp = c("2023-04-01 12:13", "2023-05-30 01:45",
    "2023-06-07 13:01", "2023-09-23 23:02") |>
    lubridate::ymd_hm(),
    varchar = LETTERS[5:8],
    enum = factor(c("Y", "Y", "N", "Y"), levels = c("N", "Y"))
  ) |>
  duckdb::dbWriteTable(con, "examples", value = _, overwrite = TRUE)

dplyr::tbl(con, "examples")
```

```
# Source:   table<examples> [4 x 6]
# Database: DuckDB 0.7.1 [root@Darwin 22.6.0:R 4.3.0/:memory:]
  boolean double integer timestamp           varchar enum
  <lgl>    <dbl>   <int> <dttm>              <chr>   <fct>
1 TRUE     -1.2        3 2023-04-01 12:13:00 E       Y
2 TRUE      5.65      20 2023-05-30 01:45:00 F       Y
3 FALSE     0.91       0 2023-06-07 13:01:00 G       N
4 TRUE    100         -2 2023-09-23 23:02:00 H       Y
```

```
duckdb::dbDisconnect(con, shutdown = TRUE)
```

4.2 Tipos de dados aninhados

Além da extensa diversidade de tipos de dados mencionada anteriormente, o `duckdb` também oferece suporte a tipos de dados aninhados. Esses tipos aninhados viabilizam uma estruturação mais sofisticada dos dados, proporcionando uma estrutura de armazenamento mais complexa.

Os tipos suportados são `list`, `struct` e `map`. É observado que no R existe uma perfeita compatibilidade destes tipos.

Primeiro é criada uma tabela chamada `NEST` com as variáveis:

- `int_list`, com o tipo lista (`[]`) de inteiros (`INT`)
- `varchar_list`, com o tipo lista (`[]`) de caracteres (`VARCHAR`)
- `struct`, do tipo struct (`STRUCT`) com duas variáveis `INT` e `VARCHAR`

Em seguida é feito o `INSERT` de uma observação conforme os tipos que foram definidos na criação da tabela. Foram utilizadas as funções `DBI::dbSendStatement` para pré-definir uma instrução para o banco de dados e `DBI::dbBind` para efetivamente preencher a instrução com os dados que serão inseridos na tabela.

Por último a consulta mostra como são apresentados estes dados aninhados no `R`.

```
con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
DBI::dbExecute(con, "CREATE TABLE NEST (int_list INT[], varchar_list VARCHAR[],
  struct STRUCT(i INT, j VARCHAR))"
)
```

```
[1] 0
```

```
stmt <- DBI::dbSendStatement(con, "INSERT INTO NEST VALUES (?, ?, ?)")
DBI::dbBind(stmt, list("[1, 2]", "['a', 'b']", "{'i': 5, 'j': 'c'}"))

dplyr::tbl(con, "nest")
```

```
# Source:   table<nest> [1 x 3]
# Database: DuckDB 0.7.1 [root@Darwin 22.6.0:R 4.3.0/:memory:]
  int_list  varchar_list struct$i $j
  <list>    <list>          <int> <chr>
1 <int [2]> <chr [2]>           5 c
```

```
duckdb::dbDisconnect(con, shutdown = TRUE)
```

4.3 Leitura e escrita de arquivos externos

4.3.1 `csv` e `parquet`

Com o `duckdb` é possível fazer a leitura e escrita de arquivos em estruturas tabulares tradicionais `csv` e `parquet`.

No exemplo abaixo utiliza-se a função `duckdb::duckdb_read_csv` para a leitura de um arquivo `csv` salvo no diretório `../data/nyc-taxi.csv` e criar uma tabela de nome `nyc-taxi` na base de dados.

```
con <- duckdb::dbConnect(duckdb::duckdb(), "../data/nyc-taxi.duckdb")
duckdb::duckdb_read_csv(con, "nyc-taxi", "../data/nyc-taxi.csv")
```

Em seguida, a partir da tabela `nyc-taxi` na base de dados foi exportado localmente um arquivo no formato `parquet`, utilizando a instrução `COPY`.

```
DBI::dbExecute(con, "COPY 'nyc-taxi' TO '../data/nyc-taxi.parquet'
  (FORMAT PARQUET);")
duckdb::dbDisconnect(con, shutdown = TRUE)
```

Por último, temos a função `read_parquet` para leitura de arquivos locais no formato `parquet`.

```
con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
DBI::dbGetQuery(con, "SELECT * FROM read_parquet('../data/nyc-taxi.parquet')
  LIMIT 2;") |>
  dplyr::as_tibble()

duckdb::dbDisconnect(con, shutdown = TRUE)
```

```
# A tibble: 2 × 11
  id         vendor_id pickup_datetime     dropoff_datetime    passenger_count
  <chr>          <int> <chr>               <chr>                         <int>
1 id2875421          2 2016-03-14 17:24:55 2016-03-14 17:32:30               1
2 id2377394          1 2016-06-12 00:43:35 2016-06-12 00:54:38               1
#   6 more variables: pickup_longitude <dbl>, pickup_latitude <dbl>,
#   dropoff_longitude <dbl>, dropoff_latitude <dbl>, store_and_fwd_flag <chr>,
#   trip_duration <int>
```

4.3.2 `json`

O `duckdb` é um sistema de gerenciamento de banco de dados relacional que não apenas suporta a leitura de dados tabulares convencionais, mas também é capaz de processar arquivos `json`. Com isso, você pode diretamente lidar com dados não estruturados por meio do `duckdb`.

A seguir um exemplo de arquivo no formato `json`.

```
[
  {"Name" : "Mario", "Age" : 32, "Occupation" : "Plumber"},
  {"Name" : "Peach", "Age" : 21, "Occupation" : "Princess"},
  {},
  {"Name" : "Bowser", "Occupation" : "Koopa"}
]
```

Para leitura do arquivo é criada uma conexão, instalada e carregada a extensão para manipução dos arquivos `json`.

```
con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
DBI::dbExecute(con, "INSTALL json;")
DBI::dbExecute(con, "LOAD json;")
```

Incorporando a função `read_json_auto` na consulta através da função `DBI::dbGetQuery`, o resultado da leitura do arquivo é apresentado como um `data.frame`.

```
DBI::dbGetQuery(con, "SELECT * FROM read_json_auto('../data/example.json')")
duckdb::dbDisconnect(con, shutdown = TRUE)
```

É observado no resultado que cada linha do `data.frame` é um documento do arquivo `json` original.

```
    Name Age Occupation
1  Mario  32    Plumber
2  Peach  21   Princess
3   <NA>  NA       <NA>
4 Bowser  NA      Koopa
```

4.4 Funções

Existem inúmeras funções à disposição para uso no `duckdb`. Ao trabalhar com dados em um banco deste sistema, utilizar suas funções internas pode oferecer um desempenho muito superior na consulta.

No exemplo a seguir foi criado uma tabela chamada `functions` com duas variáveis `telefone` e `start_date`. Na consulta à esta tabela é aplicado um regex para extrair somente os números de um texto. Observa-se que a função `regexp_extract` do `duckdb` é chamada dentro da função `dplyr::sql` e dentro de um `dplyr::mutate` ou no caso aqui `dplyr::transmute`. No segundo caso temos uma função que irá contabilizar o número de semanas entre duas datas (`datediff`) e no último caso temos uma função que irá trazer o valor de `pi`.

```
df <- data.frame(telefone = c("Meu telefone é: 21991831234"),
                 start_date = c("1984-10-19") |> as.Date())

con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")

# Registra o df como uma tabela virtual (view)
duckdb::duckdb_register(con, "functions", df)

dplyr::tbl(con, "functions") |>
  dplyr::transmute(tel_extract = dplyr::sql("regexp_extract(telefone, '[0-9]+')"),
                   weeks = dplyr::sql("datediff('week', start_date, today())"),
                   pi = dplyr::sql("pi()"))
```

```
# Source:   SQL [1 x 3]
# Database: DuckDB 0.7.1 [root@Darwin 22.6.0:R 4.3.0/:memory:]
  tel_extract weeks    pi
  <chr>       <dbl> <dbl>
1 21991831234  2029  3.14
```

```
duckdb::dbDisconnect(con, shutdown = TRUE)
```

5 Casos de uso

Nesta seção iremos mostrar alguns exemplos de utilização do `duckdb`.

5.1 Mineração de texto

Neste primeiro exemplo será mostrado como pode ser aplicada várias funções do `duckdb` para manipulação de texto.

No primeiro trecho do código é feita a leitura dos textos que irão ser manipulados e salvo no objeto `bible`.

```
bible <- readr::read_lines(
  url("https://www.o-bible.com/download/kjv.txt"),
  skip = 1
  ) |> dplyr::as_tibble()
```

A seguir é criada uma conexão com uma base de dados e uma tabela com o nome `bible`. Esta tabela criada foi uma tabela virtual, onde os dados não são armazenados fisicamente, ou seja, os dados persistem somente enquanto a conexão com o banco permanece ativa. A criação de tabelas virtuais é feita com a função `duckdb::duckdb_register`.

```
con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
duckdb::duckdb_register(con, "bible", bible)
```

A tabela denominada `bible` pode ser visualizada abaixo:

```
dplyr::tbl(con, "bible")
```

```
# Source:   table<bible> [?? x 1]
# Database: DuckDB 0.7.1 [root@Darwin 22.6.0:R 4.3.0/:memory:]
   value
   <chr>
 1 Ge1:1 In the beginning God created the heaven and the earth.
 2 Ge1:2 And the earth was without form, and void; and darkness was upon the fa~
 3 Ge1:3 And God said, Let there be light: and there was light.
 4 Ge1:4 And God saw the light, that it was good: and God divided the light fro~
 5 Ge1:5 And God called the light Day, and the darkness he called Night. And th~
 6 Ge1:6 And God said, Let there be a firmament in the midst of the waters, and~
 7 Ge1:7 And God made the firmament, and divided the waters which were under th~
 8 Ge1:8 And God called the firmament Heaven. And the evening and the morning w~
 9 Ge1:9 And God said, Let the waters under the heaven be gathered together unt~
10 Ge1:10 And God called the dry land Earth; and the gathering together of the ~
# i more rows
```

Cada linha da tabela é um versículo da bíblia e em cada início do texto há uma **referência** em que as duas primeiras letras representam o livro, o número antes dos ':' representa o capítulo e os últimos números (após os ':') representam o versículo.

No próximo bloco de códigos são aplicadas um conjunto de funções para a manipulação dos textos:

- `regexp_extract` faz a extração do livro (salvo com o nome `book`), ou seja, somente a parte de texto da referência.

- `regexp_replace` faz uma limpeza e padronização do texto ao substituir a referência do texto (`\\w+\\d+\\:\\d+`, e.g. Ge1:1) ou (`|`) pontuações (`\\;|\\,|\\.|\\:`) por vazio (`''`). Foi aplicada a função `trim` para tirar os espaços das extremidades do texto e por último foi aplicada a função `lcase` para por o texto em caixa baixa.
- `regexp_split_to_array` transforma o texto em uma lista de palavras, utilizando o espaço (`\\s`) entre as palavras como o ponto de corte.
- `list_filter` filtra a lista de palavras que não correspondem (`regexp_matches`) com *in*, *the* e *and*.

```
(words <- dplyr::tbl(con, "bible") |>
  dplyr::mutate(book = dplyr::sql("regexp_extract(regexp_extract(value,
                '\\w+\\d+\\:\\d+'), '[A-Za-z]+')"),
                text = dplyr::sql("lcase(trim(regexp_replace(value,
                '\\w+\\d+\\:\\d+|\\;|\\,|\\.|\\:', '', 'g')))"),
                word = dplyr::sql("regexp_split_to_array(text, '\\s')"),
                word_clean = dplyr::sql("list_filter(word, x -> NOT
                regexp_matches(x, 'in|the|and'))")) |>
  dplyr::select(book, text, word, word_clean) |> head(1) |> dplyr::as_tibble())
```

```
# A tibble: 1 x 4
  book  text                                                  word   word_clean
  <chr> <chr>                                                 <list> <list>
1 Ge    in the beginning god created the heaven and the earth <chr>  <chr [4]>
```

O resultado da primeira observação pode ser visto com a consulta acima. E abaixo vemos como ficaram estruturadas as variáveis `word` e `word_clean` após a consulta.

```
words$word
```

```
[[1]]
 [1] "in"        "the"       "beginning" "god"       "created"   "the"
 [7] "heaven"    "and"       "the"       "earth"
```

```
words$word_clean
```

```
[[1]]
[1] "god"     "created" "heaven"  "earth"
```

```
duckdb::dbDisconnect(con, shutdown = TRUE)
```

5.2 Dados de COVID-19

Neste próximo exemplo será mostrado como trabalhar com dados de COVID-19.

Os dados contidos no link (`url`) abaixo provêm do repositório da John Hopkins University e abrangem informações relacionadas à pandemia de COVID-19. Esta fonte compila e disponibiliza os dados atualizados de todo o mundo.

Abaixo vemos que os dados contém as informações:

- `Province.State`
- `Country.Region`
- `Lat` e `Long`
- Diversas colunas com o padrão mês, dia e ano (`XMM.DD.AA`) tendo o valor acumulado de casos

```
url <- paste0(
  "https://raw.githubusercontent.com/CSSEGISandData/COVID-19/master/",
  "csse_covid_19_data/csse_covid_19_time_series/",
  "time_series_covid19_confirmed_global.csv"
  )

read.csv(url, stringsAsFactors = FALSE) |>
  dplyr::as_tibble()
```

```
# A tibble: 289 x 1,147
   Province.State         Country.Region   Lat   Long X1.22.20 X1.23.20 X1.24.20
   <chr>                  <chr>          <dbl>  <dbl>    <int>    <int>    <int>
 1 ""                     Afghanistan     33.9  67.7         0        0        0
 2 ""                     Albania         41.2  20.2         0        0        0
 3 ""                     Algeria         28.0   1.66        0        0        0
 4 ""                     Andorra         42.5   1.52        0        0        0
 5 ""                     Angola         -11.2  17.9         0        0        0
 6 ""                     Antarctica     -71.9  23.3         0        0        0
 7 ""                     Antigua and B~  17.1 -61.8         0        0        0
 8 ""                     Argentina      -38.4 -63.6         0        0        0
 9 ""                     Armenia         40.1  45.0         0        0        0
10 "Australian Capital T~ Australia      -35.5 149.          0        0        0
# i 279 more rows
# i 1,140 more variables: X1.25.20 <int>, X1.26.20 <int>, X1.27.20 <int>,
#   X1.28.20 <int>, X1.29.20 <int>, X1.30.20 <int>, X1.31.20 <int>,
#   X2.1.20 <int>, X2.2.20 <int>, X2.3.20 <int>, X2.4.20 <int>, X2.5.20 <int>,
#   X2.6.20 <int>, X2.7.20 <int>, X2.8.20 <int>, X2.9.20 <int>, X2.10.20 <int>,
```

```
#   X2.11.20 <int>, X2.12.20 <int>, X2.13.20 <int>, X2.14.20 <int>,
#   X2.15.20 <int>, X2.16.20 <int>, X2.17.20 <int>, X2.18.20 <int>, ...
```

Para a manipulação destes dados criaremos uma conexão e uma tabela virtual com o nome de `"covid19"`.

```
con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
duckdb::duckdb_register(con, "covid19", read.csv(url, stringsAsFactors = FALSE))
```

Em seguida utilizamos a instrução `PIVOT_LONGER` a fim de mudar a orientação da tabela de *wide* para *long*. Sendo que a principal alteração será agrupar o conteúdo que estão distribuídos em várias colunas separadas para apenas uma coluna com os valores e uma outra coluna com as datas. O argumento `ON` com a função `COLUMNS('X')` declara quais são as colunas que deverão ser transformadas e os argumentos `NAME date` e `VALUE cumulate` declaram a construção das novas colunas que irão receber o novo dado estruturado.

- `replace` irá substituir o valor `'X'` por `''` e a função `strptime` transforma o dado do tipo `VARCHAR` para o tipo `DATE` seguindo o padrão da data `'%m.%d.%y'`.
- É construída uma variável `value` que é o valor acumulado - o valor do dia anterior (`dplyr::lag(cumulate)`), assim criamos uma variável com o valor exato do dia.
- É aplicado um filtro `date > "2020-02-23"`.
- A função `head` equivale a declaração `LIMIT` no SQL.

```
dplyr::tbl(con, dplyr::sql("(PIVOT_LONGER covid19 ON COLUMNS('X')
  INTO NAME date VALUE cumulate)")) |>
  dplyr::select(country = Country.Region, date, cumulate) |>
  dplyr::mutate(date = dplyr::sql("strptime(replace(date, 'X', ''), '%m.%d.%y')"),
                value = cumulate - dplyr::lag(cumulate)) |>
  dplyr::filter(date > "2020-02-23") |> head(3)
```

```
# Source:   SQL [3 x 4]
# Database: DuckDB 0.7.2-dev2706 [root@Darwin 22.4.0:R 4.2.3/:memory:]
  country     date                cumulate value
  <chr>       <dttm>                 <int> <int>
1 Afghanistan 2020-02-24 00:00:00        5     5
2 Afghanistan 2020-02-25 00:00:00        5     0
3 Afghanistan 2020-02-26 00:00:00        5     0
```

Para utilizar a declaração `PIVOT_LONGER` foi necessário instalar a versão `0.8.0` do pacote em desenvolvimento do `duckdb`.

5.3 Lendo dados de um serviço de armazenamento na núvem

Com o `duckdb` é possível ler arquivos remotos armazenados em um recurso na núvem. O exemplo que utilizaremos aqui é do tabela com as viagens de taxi de New York, em um arquivo de formato `parquet` hospedado em um *Cloud Object Storage (COS)* da IBM.

Primeiro é criada uma conexão em memória e nesta conexão é intalado e carregado a extensão `httpfs`.

```
con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
DBI::dbExecute(con, "INSTALL httpfs;")
DBI::dbExecute(con, "LOAD httpfs;")
```

Em seguida são declaradas as informações do serviço onde está hospedado o arquivo no qual se quer consultar.

```
library(glue)

s3_region <- Sys.getenv('S3_REGION')
s3_endpoint <- Sys.getenv('S3_ENDPOINT')
s3_access_key_id <- Sys.getenv('S3_ACCESS_KEY_ID')
s3_secret_access_key <- Sys.getenv('S3_SECRET_ACCESS_KEY')

DBI::dbExecute(con, glue("SET s3_region='{s3_region}';"))
DBI::dbExecute(con, glue("SET s3_endpoint='{s3_endpoint}';"))
DBI::dbExecute(con, glue("SET s3_access_key_id='{s3_access_key_id}';"))
DBI::dbExecute(con, glue("SET s3_secret_access_key='{s3_secret_access_key}';"))
```

As variáveis de ambiente são especificadas da seguinte forma:

```
S3_REGION=us-south
S3_ENDPOINT=s3.us-south.cloud-object-storage.appdomain.cloud
S3_ACCESS_KEY_ID=<s3_access_key_id>
S3_SECRET_ACCESS_KEY=<s3_secret_access_key>
```

Elas podem ser salvas em um arquivo `.Renviron` e lidas com a função `readRenviron()`.

Finalmente a tabela poderá ser consultada diretamente de onde ela está armazenada.

```
dplyr::tbl(con, "s3://duckdb-ser/nyc-taxi.parquet")
```

```
# Source:   table<s3://duckdb-ser/nyc-taxi.parquet> [?? x 11]
# Database: DuckDB 0.7.1 [root@Darwin 22.4.0:R 4.2.3/:memory:]
   id         vendor_id pickup_datetime     dropoff_datetime    passenger_count
   <chr>          <int> <chr>               <chr>                         <int>
 1 id2875421          2 2016-03-14 17:24:55 2016-03-14 17:32:30               1
 2 id2377394          1 2016-06-12 00:43:35 2016-06-12 00:54:38               1
 3 id3858529          2 2016-01-19 11:35:24 2016-01-19 12:10:48               1
 4 id3504673          2 2016-04-06 19:32:31 2016-04-06 19:39:40               1
 5 id2181028          2 2016-03-26 13:30:55 2016-03-26 13:38:10               1
 6 id0801584          2 2016-01-30 22:01:40 2016-01-30 22:09:03               6
 7 id1813257          1 2016-06-17 22:34:59 2016-06-17 22:40:40               4
 8 id1324603          2 2016-05-21 07:54:58 2016-05-21 08:20:49               1
 9 id1301050          1 2016-05-27 23:12:23 2016-05-27 23:16:38               1
10 id0012891          2 2016-03-10 21:45:01 2016-03-10 22:05:26               1
#   more rows
#   6 more variables: pickup_longitude <dbl>, pickup_latitude <dbl>,
#   dropoff_longitude <dbl>, dropoff_latitude <dbl>, store_and_fwd_flag <chr>,
#   trip_duration <int>
```

5.4 Análise de dados espaciais

Com o `duckdb` também é possível analisar dados espaciais através da extensão `spatial`.

Primeiro é criada uma conexão em memória e nesta conexão é intalado e carregado a extensão `spatial`.

```
con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
DBI::dbExecute(con, "INSTALL httpfs;")
DBI::dbExecute(con, "LOAD httpfs;")
```

Será lido novamente os dados do COS como mostrado na seção anterior e em seguida na consulta será aplicado um conjunto de funções para análise espacial.

```
nyc_taxi_spatial <- dplyr::tbl(con, "s3://duckdb-ser/nyc-taxi.parquet") |>
  dplyr::mutate(
    pickup_point = dplyr::sql("ST_Transform(ST_Point(pickup_latitude,
      pickup_longitude), 'EPSG:4326', 'ESRI:102718')"),
    dropoff_point = dplyr::sql("ST_Transform(ST_Point(dropoff_latitude,
      dropoff_longitude), 'EPSG:4326', 'ESRI:102718')"),
    aerial_distance = dplyr::sql("ST_Distance(pickup_point, dropoff_point)/3280.84")
  ) |> dplyr::as_tibble()
```

- `ST_Point` recebe como input as geolocalizações
- `ST_Trasform` faz a conversão das coordenadas geográficas de latitude e longitude, que utilizam o sistema "WGS84" (`EPSG:4326`), em coordenadas de projeção específicas para a região de Nova York, denominadas "NAD83 / New York Long Island ftUS" (`ESRI:102718`). Essa transformação é útil ao lidar com dados geoespaciais relacionados à área de Nova York e é necessária para assegurar uma representação precisa das coordenadas nessa região, minimizando a distorção
- `ST_Distance` calcula a distância em linha reta entre o ponto de embarque e o ponto de desembarque. A razão entre a distância e o valor 3280.84 é um ajuste para apresentar o valor em Km.

Abaixo vemos apenas um ajuste na apresentação dos resultados e a tabela com as informações extraídas e calculada.

```
nyc_taxi_spatial |>
  dplyr::select(pickup_longitude, pickup_latitude,
                dropoff_longitude, dropoff_latitude,
                aerial_distance) |>
  dplyr::slice(1) |>
  tidyr::pivot_longer(tidyr::everything()) |>
  dplyr::mutate(value = tibble::num(value, digits = 5))
```

```
# A tibble: 5 × 2
  name                  value
  <chr>             <num:.5!>
1 pickup_longitude  -73.98215
2 pickup_latitude    40.76794
3 dropoff_longitude -73.96463
4 dropoff_latitude   40.76560
5 aerial_distance     1.50216
```

A seguir vemos o mapa com os pontos de embarque e desembarque, ligados por uma linha reta representando a distância que foi calculada.

5.5 Um banco embarcado em uma API

O `duckdb` é um banco de dados embarcado, o que significa que ele é projetado para ser incorporado diretamente em aplicativos ou sistemas, em vez de ser executado como um serviço separado em um servidor por exemplo.

Nesse exemplo será mostrado como incorporar o `duckdb` em uma API construída com `plumber`.

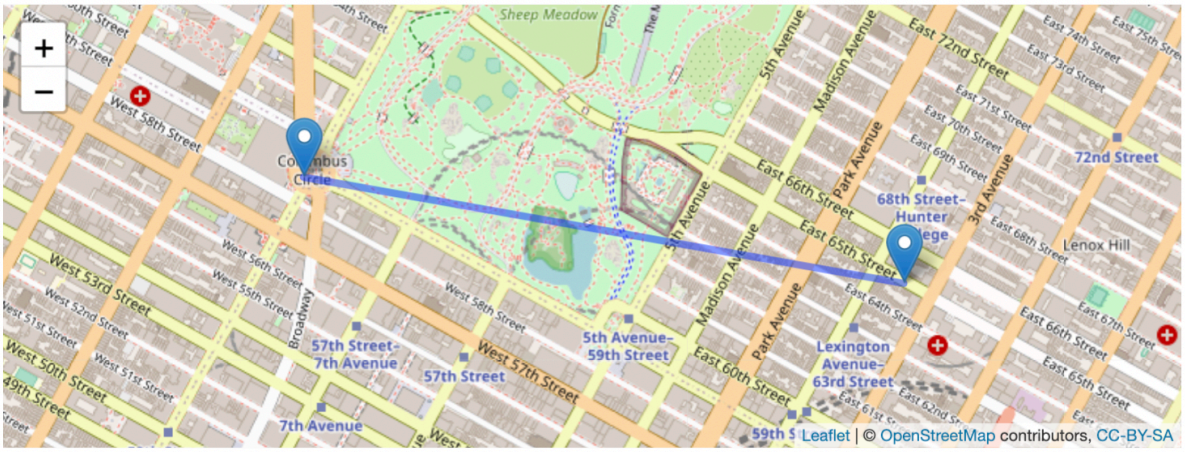

Figura 1: Pontos de embarque e desembarque

A API é estruturada em uma função no arquivo `api.R`. Esta função irá consultar uma tabela no formato `parquet` que está hospedada no COS. A função tem um argumento `input` e receberá um `id` para filtrar as informações de uma viagem de taxi.

```
#* @apiTitle Mostrar informações segundo ID

#* @param input
#* @get /info
function(input) {
  # Ler variáveis de ambiente
  readRenviron(".Renviron")

  s3_region <- Sys.getenv("S3_REGION")
  s3_endpoint <- Sys.getenv("S3_ENDPOINT")
  s3_access_key_id <- Sys.getenv("S3_ACCESS_KEY_ID")
  s3_secret_access_key <- Sys.getenv("S3_SECRET_ACCESS_KEY")

  # Criar conexão com o banco
  con <- duckdb::dbConnect(duckdb::duckdb(), ":memory:")
  invisible(DBI::dbExecute(con, "INSTALL httpfs;"))
  invisible(DBI::dbExecute(con, "LOAD httpfs;"))
  invisible(DBI::dbExecute(
    con,
    glue::glue("SET s3_region='{s3_region}';")
  ))
  invisible(DBI::dbExecute(
```

```
    con,
    glue::glue("SET s3_endpoint='{s3_endpoint}';")
  ))
  invisible(DBI::dbExecute(
    con,
    glue::glue("SET s3_access_key_id='{s3_access_key_id}';")
  ))
  invisible(DBI::dbExecute(
    con,
    glue::glue("SET s3_secret_access_key='{s3_secret_access_key}';")
  ))

  # Consulta
  resposta <- dplyr::tbl(con, "s3://duckdb-ser/nyc-taxi.parquet") |>
    dplyr::filter(id == input) |>
    dplyr::as_tibble() |>
    as.data.frame()

  duckdb::dbDisconnect(con, shutdown = TRUE)

  # Resultado
  return(resposta)
}
```

Para iniciar a API deve ser executado o comando abaixo, onde é informado o nome do script e a porta em que a API estará disponível:

```
plumber::pr("api.R") |>
  plumber::pr_run(port=8010)
```

Para fazer a requisição na API podemos usar o pacote `httr`. No exemplo a API roda localmente `http://127.0.0.1` na porta `8010` e foi requisitada as informações da viagem com o `id` de `id2875421`.

```
httr::GET("http://127.0.0.1:8010/info?input=id2875421") |>
  httr::content() |>
  jsonlite::toJSON(auto_unbox = TRUE, pretty = TRUE)
```

No resultado observa-se as informações da viagem segundo o `id` informado:

```
[
  {
```

```
    "id": "id2875421",
    "vendor_id": 2,
    "pickup_datetime": "2016-03-14 17:24:55",
    "dropoff_datetime": "2016-03-14 17:32:30",
    "passenger_count": 1,
    "pickup_longitude": -73.9822,
    "pickup_latitude": 40.7679,
    "dropoff_longitude": -73.9646,
    "dropoff_latitude": 40.7656,
    "store_and_fwd_flag": "N",
    "trip_duration": 455
  }
]
```