R E DUCKDB

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.

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 Nova 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` 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` o tempo de execução da leitura dados e consulta foi um pouco maior do que 2 seg.

```
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 pode 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`.

```
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 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.

```r
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`.

```r
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`.

```r
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

5 Casos de uso

5.1 Text mining

5.2 Dados de COVID 19

5.3 Lendo dados do S3

5.4 Análise de dados espaciais

5.5 Um banco embarcado em uma API