

Processamento *Out-of-Core* com **duckdb** e **DBI** no R

ESTAT0109 – Mineração de Dados em Estatística

Prof. Dr. Sadraque E. F. Lucena
sadraquelucena@academico.ufs.br
<http://sadraquelucena.github.io/mineracao>

Objetivo da Aula

- Aprender a manipular grandes bases de dados no R.
- Conhecer os pacotes `duckdb` e `DBI`.
- Fazer consultas usando a linguagem SQL dentro do R.

O Muro da Memória RAM

- O R é, por natureza, uma ferramenta *in-memory*. É comum usarmos o comando: `meus_dados <- read.csv("arquivo_grande.csv")`.
- **Problema:** O que acontece se `arquivo_grande.csv` tem 50 GB e seu notebook tem 16 GB de RAM?
 - O R tenta alocar 50 GB de espaço na RAM.
 - O sistema operacional tenta compensar usando swap (disco), o que torna o processo astronomicamente lento.
 - Na maioria dos casos, a sessão do R simplesmente trava ou é morta pelo sistema.
- **Solução:** Em vez de trazer os dados para o R, nós lemos e processamos os dados diretamente no disco, e trazemos para a RAM apenas o resultado final (que geralmente é pequeno).
 - Isso é chamado de processamento *Out-of-Core* (ou *On-Disk*).
 - Podemos usar os pacotes `duckdb` e `DBI` para fazê-lo.

O Pacote **duckdb**

- Funciona como seu assistente inteligente para dados grandes.
- Imagine que seus dados são uma biblioteca gigante:
 - **Método tradicional:** Trazer todos os livros para sua mesa (RAM).
 - **Com DuckDB:** Pedir ao bibliotecário que consulte os livros nas estantes (disco) e traga apenas a resposta.

O Pacote `duckdb`

- É um sistema de gerenciamento de banco de dados (SGBD) analítico, *in-process* e colunar. Ou seja:
 - **Analítico (OLAP)**: Otimizado para consultas complexas, agregações e filtros (ex: GROUP BY, SUM, AVG).
 - **In-Process**: Não é um servidor (como PostgreSQL ou MySQL). Ele roda dentro da sua sessão R. Não há instalação, configuração ou gerenciamento de servidor. Apenas `install.packages("duckdb")` e pronto.
 - **Colunar**: Esta é a chave. Bancos de dados tradicionais armazenam dados por linha. O `duckdb` armazena por coluna.
- Se sua consulta é `SELECT VARIABEL1, COUNT(*) ...`, ele lê apenas a coluna `VARIABEL1` do disco, ignorando todas as outras (nome, data, etc.). Isso resulta em uma velocidade maior.
- `duckdb` implementa uma versão muito abrangente e moderna do padrão SQL (Structured Query Language).

O Pacote DBI

- **DBI** (*Database Interface*) é um pacote que fornece uma camada de abstração universal para comunicação com bancos de dados no R.
- Ele define um conjunto de funções consistentes:
 - `dbConnect()`: para iniciar a conexão.
 - `dbGetQuery()`: para enviar uma consulta e receber os dados de volta.
 - `dbDisconnect()`: para encerrar a conexão.
- Por que usá-lo?
 - **Consistência:** Você usa as mesmas funções **DBI** para falar com `duckdb`, `RPostgres`, `RMariaDB`, `RSQLite`, etc.
 - **Portabilidade:** Seu código R não muda. Se amanhã você decidir migrar seu processo do `duckdb` (local) para um `PostgreSQL` (servidor), você só precisa alterar a linha do `dbConnect()`.

Como o SQL se encaixa?

- O `DBI` permite que o R fale com o `duckdb`, e a língua que eles usam é o **SQL** (Structured Query Language).
- Em vez de usar comandos do pacote `dplyr` (como `filter`, `group_by`, `summarise`) que operam em `data.frames` na memória, nós escrevemos uma *string* de consulta SQL (ex: `SELECT ... FROM ... WHERE ...`)
- Nós passamos essa *string* para o `DBI` (ex: `dbGetQuery(...)`).
- O `DBI` entrega a *string* ao `duckdb`.
- O `duckdb` interpreta o SQL, otimiza a consulta, executa a operação diretamente no arquivo em disco, e retorna apenas o `data.frame` resultante para o R.

Estrutura Geral de Uso

Este é o esquema de 5 passos para qualquer análise *out-of-core* com duckdb:

```
# 1. Carregar as bibliotecas na sessão
library(DBI)
library(duckdb)

# 2. Criar a conexão com o banco de dados (para salvar as consultas)
# Opção A: Em memória (rápido, mas volátil)
#con <- dbConnect(duckdb::duckdb(), dbdir = ":memory:")

# Opção B: Persistente (recomendado)
con <- dbConnect(duckdb::duckdb(),
                  dbdir = "meu_banco_analitico.duckdb")

# 3. Informar ao duckdb onde estão os dados
# Isso NÃO carrega o CSV. Apenas cria um "ponteiro" para ele.
duckdb_register(con, "meus_dados", "arquivo_grande.csv")
```

Estrutura Geral de Uso

Este é o esquema de 5 passos para qualquer análise *out-of-core* com duckdb:

```
# 4. Fazer consultas ao banco usando SQL  
resultado <- dbGetQuery(con, "SELECT COUNT(*) FROM meus_dados")  
  
# 5. Encerrar a conexão e liberar os recursos  
dbDisconnect(con, shutdown = TRUE)
```

- Como as consultas usam SQL, vamos fazer uma breve explicação sobre o uso da linguagem.

Estrutura Geral de uma Consulta SQL

Uma consulta SQL é como uma frase que descreve os dados que você deseja. A ordem de escrita é quase sempre esta:

```
SELECT coluna1, FUNCAO(coluna2) AS novo_nome  
FROM nome_da_tabela  
WHERE condicao_de_filtro (ex: ano = 2023)  
GROUP BY coluna_de_agrupamento (ex: coluna1)  
ORDER BY coluna_de_ordenacao (ex: novo_nome) DESC  
LIMIT 10
```

A instrução **SELECT**

- Especifica as colunas que você quer ver no resultado final.
- Sempre vem acompanhada de **FROM**, que especifica de qual tabela os dados devem ser lidos.

Sintaxe:

```
SELECT coluna1, coluna2, ...
FROM nome_tabela
```

- **coluna1, coluna2, ...** são as colunas das tabelas que você quer selecionar.
- **nome_tabela** representa o nome da tabela que contém os dados.

Exemplo: Suponha que temos uma tabela *clientes* e queremos selecionar todas as colunas.

```
SELECT *
FROM clientes
```

A cláusula WHERE

- Seleciona linhas que atendem a uma condição.

Sintaxe:

```
SELECT coluna1, coluna2, ...
FROM nome_tabela
WHERE condição
```

Exemplos:

```
SELECT *
FROM clientes
WHERE idade=30
```

```
SELECT *
FROM clientes
WHERE idade>24
```

A cláusula WHERE

WHERE aceita alguns operadores:

Operador	Descrição	Operador	Descrição
=	Igual	<> ou !=	Diferente
>	Maior que	<	Menor que
>=	Maior ou igual	<=	Menor ou igual
BETWEEN	Entre um intervalo	LIKE	Busca por um padrão
IN	Para especificar múltiplos valores possíveis para uma coluna		

Exemplo:

```
SELECT *
FROM clientes
WHERE Idade BETWEEN 20 AND 30
```

A instrução GROUP BY

- Agrupa linhas com mesmo valor.
- Geralmente é usada com funções de agregação (`COUNT()`, `MAX()`, `MIN()`, `SUM()`, `AVG()`) para agrupar os resultados de uma ou mais colunas.
- Muito usada em conjunto com `ORDER BY`, para ordenação dos resultados.

Sintaxe:

```
SELECT nomes_das_colunas  
FROM nome_tabela  
WHERE condição  
GROUP BY nomes_das_colunas  
ORDER BY nomes_das_colunas
```

Exemplo:

```
SELECT COUNT(ID_cliente), estado  
FROM clientes  
GROUP BY estado  
ORDER BY COUNT(ID_cliente) DESC
```

Exemplo

Vamos usar o banco de dados sobre mortes ocorridas no Brasil em 2024 disponíveis no Sistema de Informação sobre Mortalidade (SIM), desenvolvido pelo Ministério da Saúde. Os dados estão disponíveis em: <https://opendatasus.saude.gov.br/dataset/sim>.

- Vamos usar o conjunto de dados [D0240OPEN.csv](#) (óbitos ocorridos em 2024) e as variáveis:
 - [CODMUNOCOR](#): Código relativo ao município onde ocorreu o óbito
 - [CAUSABAS](#): Causa básica da declaração de óbito
- Também vamos usar uma tabela do IBGE com os códigos e nomes dos municípios obtida em <https://www.ibge.gov.br/explica/codigos-dos-municípios.php>.
 - Usaremos o arquivo [RELATORIO_DTB_BRASIL_2024_MUNICIPIOS.xls](#).

Exemplo: Preparação do ambiente

```
# 0. Instalando os pacotes (apenas uma vez)
# install.packages(c("DBI", "duckdb"))

# 1. Carregando pacotes.
library(duckdb)
library(DBI)

# 2. Especificando onde serão armazenadas as consultas.
# Isso prepara o "motor" do duckdb para receber comandos
con <- dbConnect(    # 'con' vai armazenar o objeto da conexão
  duckdb::duckdb(),   # Especifica DuckDB como SGBD
  dbdir = ":memory:" # as consultas serão salvas na memória
                      # e depois apagadas ao finalizar
  #dbdir = "sim_obitos.duckdb" # cria uma arquivo p/ armazenar
                            # os resultados da consulta
)

# 3. Definindo o caminho para o arquivo gigante
# para não precisarmos escrever em toda consulta
caminho <- "/home/sadraque/Documentos/UFS/Disciplinas/2025.2/mi
```

Exemplo 1: Contagem total de linhas

Vamos contar quantas linhas há na tabela de dados.

```
# Criando a consulta SQL
# SELECT COUNT(*) : "Selecione a contagem de todas as linhas"
# FROM '%s' será substituído pelo caminho do arquivo
consult <- sprintf("SELECT COUNT(*) AS total
                    FROM '%s'", caminho)

# Enviando a consulta e pegando o resultado.
# O duckdb vai ler o arquivo (sem carregá-lo) e retornar
# apenas o resultado.
total_linhas <- dbGetQuery(con, consult)
total_linhas

      total
1 1426346
```

Exemplo 2: Agregando por Município

Vamos contar o número de óbitos por código do município (`CODMUNOCOR`).

```
# Montar a consulta SQL
consult <- sprintf("SELECT CODMUNOCOR, COUNT(*) AS total_obitos
                     FROM '%s'
                     GROUP BY CODMUNOCOR
                     ORDER BY total_obitos DESC",
                     caminho)

# Enviar a consulta e pegar o resultado
obitos_por_municipio <- dbGetQuery(con, consult)

# Ver as primeiras 4 linhas do resultado
head(obitos_por_municipio, n = 4L)
```

	CODMUNOCOR	total_obitos
1	355030	89208
2	330455	61443
3	310620	23340
4	261160	22146

Exemplo 3: Adicionando filtragem

Vamos consultar o número de óbitos por causas externas.

```
# Criar a consulta SQL
# CODMUNOCOR: Código do município onde ocorreu o óbito.
# CAUSABAS: 'V01' a 'V99' são os códigos da CID-10 para
#           acidentes de transporte.
consult <- sprintf("SELECT CODMUNOCOR,
                      COUNT(CAUSABAS) AS obitos_acidentes
                  FROM '%s'
                  WHERE CAUSABAS BETWEEN 'V01' AND 'V99'
                  GROUP BY CODMUNOCOR
                  ORDER BY obitos_acidentes DESC",
                  caminho)
obitos_acidentes_mun <- dbGetQuery(con, consult)
head(obitos_acidentes_mun, n = 4L) # primeiras 4 linhas
```

	CODMUNOCOR	obitos_acidentes
1	130260	380
2	520870	367
3	261160	347
4	355030	281

Exemplo 3: Adicionando filtragem

- Os códigos do SIM vêm sem os nomes dos municípios.
- Precisamos cruzar com a tabela do IBGE para saber os nomes dos municípios.

```
library(tidyverse)

tab_cod_ibge <- readxl::read_excel(
  "/home/sadraque/Documentos/UFS/Disciplinas/2025.2/mineracao de dados/IBGE_Municípios.xlsx",
  skip = 6, # Pula as 6 primeiras linhas
  col_names = TRUE # Usa a 7ª linha como nome das variáveis (depois da descrição)
) |>
  # Limpar nomes das colunas
  janitor::clean_names()
```

Exemplo 3: Adicionando filtragem

```
tab_cod_ibge <- tab_cod_ibge |>
  # Cria código de 6 dígitos removendo o dígito verificador
  mutate(codigo_6digitos = str_sub(codigo_municipio_completo,
                                    1, -2)) |>
  # Converte para numérico
  mutate(codigo_6digitos = as.numeric(codigo_6digitos)) |>
  # Seleciona e renomeia colunas finais
  select(codigo = codigo_6digitos,
         municipio = nome_municipio,
         UF = nome_uf)
```

Exemplo 3: Adicionando filtragem

```
# juntando o número de acidentes e os nomes dos municípios
obitos_acidentes_nome_municipios <- tab_cod_ibge |>
  left_join(obitos_acidentes_mun,
            by = c("codigo" = "CODMUNOCOR")) |>
  arrange(desc(obitos_acidentes)) # ordem decrescente
head(obitos_acidentes_nome_municipios)
```

A tibble: 6 × 4

	codigo	municipio	UF	obitos_acidentes
	<dbl>	<chr>	<chr>	<dbl>
1	130260	Manaus	Amazonas	380
2	520870	Goiânia	Goiás	367
3	261160	Recife	Pernambuco	347
4	355030	São Paulo	São Paulo	281
5	530010	Brasília	Distrito Federal	271
6	211130	São Luís	Maranhão	269

Encerrando a conexão

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

Exercícios

1. Faça um histograma das idades das pessoas que vieram a óbito em 2024 em todo o país (note que primeiro você precisa fazer uma consulta na base de dados).
2. Faça um gráfico de barras contando os óbitos por sexo para cada estado do país.

Conclusão e Revisão

- **Problema:** A RAM é limitada; dados massivos não cabem nela.
- **Solução:** Processamento *Out-of-Core (On-Disk)*.
- **Ferramentas:**
 - **duckdb:** Motor que faz o trabalho pesado no disco.
 - **DBI:** Interface que o R usa para se comunicar.
 - **SQL:** A linguagem que usamos para descrever o que queremos.
- **Fluxo:** `dbConnect` -> `duckdb_register` -> `dbGetQuery` (com SQL) -> `dbDisconnect`.
- **Verbos SQL:**

• SELECT (o quê)	• GROUP BY (agregar)
• FROM (de onde)	• ORDER BY (ordenar)
• WHERE (filtro de linha)	

- Mais sobre a estrutura de SQL você pode encontrar em [https://www.w3schools.com/sql/.](https://www.w3schools.com/sql/)

Fim