Universidade de Brasília

DEPARTAMENTO DE ESTATÍSTICA

18 agosto 2022

Lista 2

Prof. Guilherme Rodrigues

Computação em Estatística para dados e cálculos massivos

Tópicos especiais em Estatística 1

Aluno: Bruno Gondim Toledo | Matrícula: 15/0167636

Lista 2

Por vezes, mesmo fazendo seleção de colunas e filtragem de linhas, o tamanho final da tabela extrapola o espaço disponível na memória *RAM*. Nesses casos, precisamos realizar as operações de manipulação fora do `R`, em um banco de dados ou em um sistema de armazenamento distribuído. Outras vezes, os dados já estão armazenados em algum servidor/*cluster* e queremos carregar para o `R` parte dele, possivelmente após algumas manipulações.

Nessa lista repetiremos parte do que fizemos na Lista 1. Se desejar, use o gabarito da Lista 1 emsubstituição à sua própria solução dos respectivos itens.

Questão 1: Criando bancos de dados.

a) Crie um banco de dados `SQLite` e adicione as tabelas consideradas no item 2a) da Lista 1.

```
p_load(RSQLite)
SQL <- dbConnect(RSQLite::SQLite(), "dfDB.db")
dbWriteTable(SQL, "df_dataDB", df,overwrite = TRUE)
```

b) Refaça as operações descritas no item 2b) da Lista 1 executando códigos `sql` diretamente no bancode dados criado no item a). Ao final, importe a tabela resultante para o `R`. Não é necessário descrever novamente o que são as regiões de saúde.

Atenção:Pesquise e elabore os comandos `sql` sem usar a ferramenta de tradução de `dplyr` para `sql`.

```
qvrs <- dbGetQuery(SQL,"
                   SELECT regiao_saude,
                   COUNT(*) AS 'quantidade'
                   FROM df_dataDB
                   GROUP BY regiao_saude
                   ORDER BY quantidade DESC
                   ;")

dbWriteTable(SQL, "qvrs", qvrs,overwrite = TRUE)

dbGetQuery(SQL, 'SELECT * FROM qvrs LIMIT 5')

mediana <- dbGetQuery(SQL,"
                      SELECT AVG(quantidade) AS 'mediana'
                      FROM (
                      SELECT quantidade
                      FROM qvrs
                      ORDER BY quantidade
                      LIMIT 2
                      OFFSET (SELECT (COUNT(*) - 1) / 2
                      FROM qvrs))
                      ;")

mediana <- as.numeric(mediana)

dbExecute(SQL,
          'ALTER TABLE qvrs
```

```
          ADD faixa_de_vacinacao varchar AS
          (CASE WHEN quantidade > 2957 THEN "Alto" ELSE "Baixo" END)')

baixo <- dbGetQuery(SQL, 'SELECT * FROM qvrs
           ORDER BY quantidade ASC', n = 5)

alto <- dbGetQuery(SQL, 'SELECT * FROM qvrs
           WHERE faixa_de_vacinacao = "Alto"
           ORDER BY quantidade ASC', n = 5)

tabelasql <- rbind(alto,baixo)
```

Tabela:

regiao_saude	quantidade	faixa_de_vacinacao
29019	2963	Alto
29026	3261	Alto
13006	3288	Alto
29018	3292	Alto
51015	3351	Alto
31050	267	Baixo
31067	315	Baixo
31046	322	Baixo
31076	339	Baixo
31068	345	Baixo

c) Refaça os itens a) e b), agora com um banco de dados MongoDB.

c - a)

```
p_load(mongolite)
mongodf <- mongo(collection = "mongo_df",
            db = "tab",
            url ="mongodb://localhost")
mongodf$insert(df)
```

c - b)

```
# Testes

# mongodf$find()
# mongodf$count('{}')
#mongodf$count('{regiao_saude}:')
#Error: Invalid JSON object: {regiao_saude}
#mongodf$count('{"regiao_saude"}:')
##Error: Invalid JSON object: {"regiao_saude"}
#mongodf$count("regiao_saude")
#Error: Invalid JSON object: regiao_saude
#mongodf$count('{"regiao_saude": ""}')

# mongoqvrs <- mongodf$aggregate('[{"$group": {"_id":"$regiao_saude","n": {"$sum":1}}}]')
```

```
# mongodf$insert(mongoqurs)

# Tentando calcular a mediana
#teste <- mongoqurs$run(command = '[{count = db.coll.count();,db.coll.find().sort( {"a":1} ).skip(co
#teste

#db._id.find().sort( {"n":1} ).skip(db._id.count() / 2).limit(1);
#mongoqurs.find().sort( {"n":1} ).skip(db.teams.count() / 2).limit(1);
#count = db.coll.count();
#mongodf$find().sort( {"n":1} ).skip(count / 2 - 1).limit(1);

#mongodf$find().sort( {"n":1} ).skip(count() / 2).limit(1);

# não consegui!!
```

d) Refaça os itens c), agora usando o Apache Spark.

d - a)

```
p_load(sparklyr)

config <- spark_config()
config$`sparklyr.shell.driver-memory` <- "4G"
config$`sparklyr.shell.executor-memory` <- "4G"
config$spark.yarn.executor.memoryOverhead <- "1g"

sc <- spark_connect(master = "local", config = config)

sparkdf <- copy_to(sc, df)
```

d - b)

```
sparktable <- sparkdf %>%
  count(regiao_saude) %>%
  show_query()%>%
  collect()

mediana <- median(sparktable$n)
# não consigo colocar todas as transformações em uma unica sequencia de pipes.
# portanto terei que ficar coletando, levando pro spark, fazer a operação e repetir.

# Testando antes rodar no R

tabela <- sparktable %>%
  mutate(faixa_vacinacao = ifelse (sparktable$n >= median(sparktable$n),"alto","baixo"))
tabela<-tabela[order(tabela$n),]
tabela <- rbind(tabela[1:5,],tabela[102:106,])

# Funciona

# Agora mandando para rodar no Spark...

sparktable <- copy_to(sc, sparktable,overwrite = TRUE)
```

```
# teste <- sparktable %>%
#  mutate(faixa_vacinacao = ifelse (sparktable$n >= median(sparktable$n),"alto","baixo"))%>%
#  show_query()%>%
#  collect()

# Não funciona, não mostra a query e não coleta
# Não consegui fazer também!!
```

e) Compare o tempo de processamento das 3 abordagens (`SQLite`, `MongoDB` e `Spark`), desde o enviodo comando `sql` até o recebimento dos resultados no `R`. Comente os resultados incluindo na análise osresultados obtidos no item 2d) da Lista 1.

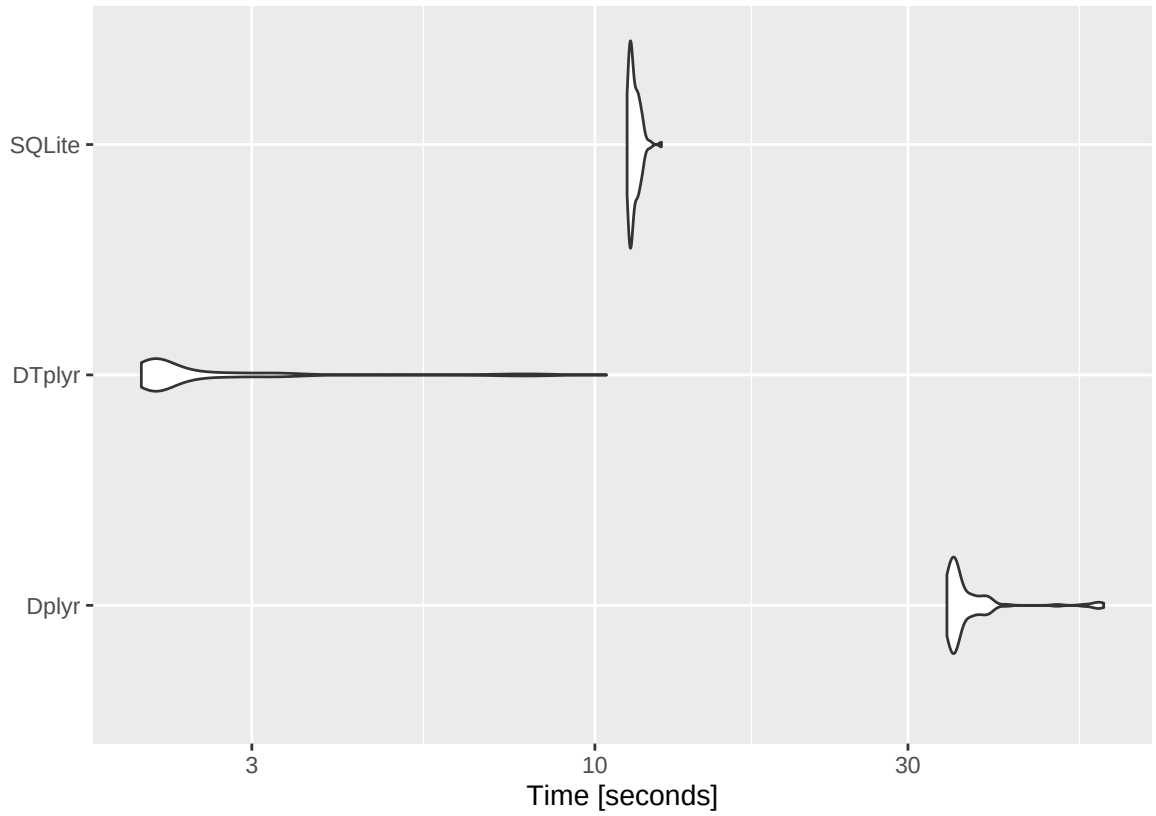
Cuidado: A performance pode ser completamente diferente em outros cenários (com outras operações,diferentes tamanhos de tabelas, entre outros aspectos).

OBS: Como não conseguir fazer os itens **c)** e **d)** por completo, não seria justo falar de comparação entre os itens visto que somente na parte de realizar as operações em `SQL` eu consegui fazer o procedimento por inteiro.

Portanto, o que dá para apresentar é o *microbenchmark* desse item, comparado ao *microbenchmark* do equivalente na lista 1.

Tempo de execução dos códigos usando `SQLite` (Lista 2), `dplyr`, `dtplyr` (Lista 1):

```
p_load(microbenchmark)
mbm <- readRDS('mbml2.rds')
autoplot(mbm)
```



Plotando o `microbenchmark` realizado 100 vezes, podemos observar que o tempo de execução do código em `dtplyr` é consideravelmente mais rápido se comparado ao tempo de execução com o `dplyr`. Inserindo o `SQLite` no gráfico, percebemos que o `SQLite` aparentemente foi mais lento que o `DTplyr` e mais rápido que o `Dplyr`, porém foi o mais consistente nos tempos de execução.

Considerações finais:

Este `PDF` é um relatório formatado a fim de entrega do trabalho de forma direta e elegante. Para acesso ao código completo com comentários sobre os processos, favor observar o arquivo *esbocolista2.r* que segue em anexo na entrega.