Universidade de Brasília

DEPARTAMENTO DE ESTATÍSTICA

02 de agosto de 2022

Lista 1: Computação eficiente (dados em memória)

Computação em Estatística para dados e cálculos massivos

Tópicos especiais em Estatística 1

Prof. Guilherme Rodrigues

César Augusto Fernandes Galvão (aluno colaborador)

Gabriel Jose dos Reis Carvalho (aluno colaborador)

José Vítor Barreto Porfírio 190089971 (**autor das soluções**)

1. **As questões deverão ser respondidas em um único relatório *PDF* ou *html*, produzido usando as funcionalidades do *Rmarkdown* ou outra ferramenta equivalente.**
2. **O aluno poderá consultar materiais relevantes disponíveis na internet, tais como livros, *blogs* e artigos.**
3. **O trabalho é individual. Suspeitas de plágio e compartilhamento de soluções serão tratadas com rigor.**
4. **Os códigos *R* utilizados devem ser disponibilizados na integra, seja no corpo do texto ou como anexo.**
5. **O aluno deverá enviar o trabalho até a data especificada na plataforma Microsoft Teams.**
6. **O trabalho será avaliado considerando o nível de qualidade do relatório, o que inclui a precisão das respostas, a pertinência das soluções encontradas, a formatação adotada, dentre outros aspectos correlatos.**
7. **Escreva seu código com esmero, evitando operações redundantes, visando eficiência computacional, otimizando o uso de memória, comentando os resultados e usando as melhores práticas em programação.**

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. Outas 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 em substituição à sua própria solução dos respectivos itens.

Preparação: Leitura dos bancos

Relembrando a solução do item 2a) da Lista 1, o objetivo era carregar para o ambiente do `R` os dados de vacinação que haviam sido baixados da base de dados do governo. Dessa vez seguiu-se carregando os arquivos pelo `vroom` por ter sido aparentemente mais rápido e poder usar o vetor de arquivos como parâmetro da função.

```
if (!require(pacman)) install.packages("pacman")
pacman::p_load("vroom", "glue", "geobr", "rjson", "dplyr")


dados_path <- ".\\Lista_1\\dados"
arquivos <- glue("{dados_path}\\{list.files(dados_path)}")
vax_all <- vroom(
  arquivos,
  col_select = c(
    "estabelecimento_uf",
    "vacina_descricao_dose",
    "estabelecimento_municipio_codigo"
  ),
  show_col_types = FALSE,
  col_types = list("estabelecimento_municipio_codigo" = col_character()),
  num_threads = 7
)
```

Utilizando as facilidades do `geobr` e carregando o arquivo de códigos também baixado durante a Lista 1, lembrando também que o gabarito da Lista 1 é diferente do que foi solicitado na mesma e ao final dessa Lista consta um aviso sobre como mudar a tabela em uso poderia levar a resultados completamente diferentes, parece adequado juntar de imediato o banco do `geobr` com a correspondência do IBGE já quer são bancos pequenos e podem poupar todas as soluções de um procedimento de *join*

```
reg_saude <- read_health_region()
ibge_geobr <- fromJSON(file = "./Lista_1/mapa_codigos.json") %>% unlist()
codigos <- data.frame(
  "est_mun_codigo" = names(ibge_geobr),
  "code_health_region" = ibge_geobr
) %>%
  merge(reg_saude, by = "code_health_region") %>%
  select(est_mun_codigo, name_health_region)
```

Questão 1: Criando bancos de dados.

a) Crie um banco de dados SQLite e adicione a tabela gerada no item 1e) da Lista 1.

Solução: Dispondo dos pacotes `DBI` e `RSQLite`, para criar o banco de dados basta inicializar a conexão com a `db` (*database*) pela função `dbConnect`, que por sua vez já lida com criar o banco caso ele ainda não exista. A função `dbWriteTable` consegue enviar um `data.frame`, ou similar, do `R` para a `db` como no código a seguir

```r
sql_conn <- dbConnect(RSQLite::SQLite(), "./Lista_2/lista2.sqlite")
dbWriteTable(sql_conn, "vax", vax_all)
dbWriteTable(sql_conn, "codigos", codigos)
```

b) Refaça as operações descritas no item 2b) da Lista 1 executando códigos `sql` diretamente no banco de dados criado no item **a)**. Ao final, importe a tabela resultante para `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.

Solução: Sabendo da existencia dessa tabela de códigos é possível utilizar uma *query* `SQL` para filtrar apenas as observações para as quais foi aplicada a segunda dose da mesma com a tabela criada no item **a)** com um simples `WHERE`, isto é,

```sql
SELECT * FROM vax
WHERE (vacina_descricao_dose='2ª Dose' OR
       vacina_descricao_dose='2ª Dose Revacinação')
```

Note no código fonte ao final do arquivo que a *query* não foi prontamente executada na base de dados, apenas criou-se uma `string` contendo a *query* para concatenar com as demais e executar a *query* em uma "tacada" só. Para as próximas *querys* vale lembrar que a notação "`{variável}`" é a notação do pacote `glue` para avaliar variáveis e inserir seu valor na *string*, mas ao longo das *querys* apresentadas neste item é apenas para se referir a como a *query* anterior foi alocada.

A implemtenação do `LEFT JOIN` em `SQLite` poderia ser feita para dar sequência à solução, toda via, concatenar os codigos de municipio dessa forma deixou a desejar em performance. Haja vista esse problema, a abordagem utilizada foi selecionar as observações do nome da região de saúde no banco de códigos que corresponde aos códigos dos estabelecimento no banco de vacinações e com isso criar uma coluna de saída

```sql
SELECT (
    SELECT name_health_region
    FROM codigos
    WHERE (codigos.est_mun_codigo =
           seg_dose.estabelecimento_municipio_codigo)
) AS regiao_saude
FROM ({last_query}) AS seg_dose
```

Segue-se do resultado da *query* anterior, e em seguida utilizando a função `COUNT(*)` do `SQL` em conjunto com um `GROUP BY` para contar o número de vacinados por região de saúde.

```sql
SELECT regiao_saude, COUNT(*) AS N
FROM ({last_query})
GROUP BY regiao_saude
```

Dispondo da frequência de cada uma das regiões de saúde no banco de vacinados, vamos parar para salvar a tabela gerada pela query anterior como "qnt_vax", pois por algum motivo esse procedimento se mostrou absurdamente mais rápido que concatenar com a próxima *query* utilizando um `AS` (com o `WITH` e `AS` demorava na ordem de 21 minutos). Então, o procedimento para criar as faixas de vacinação em `SQLite` é através da operação de `CASE WHEN`, que funciona similarmente a um *if statement* do `R`

```sql
SELECT regiao_saude, N,
    CASE WHEN N > (SELECT MEDIAN(N) FROM qnt_vax)
        THEN 'Alto'
        ELSE 'Baixo'
```

```
    END AS Faixa
FROM qnt_vax
```

Note que a tabela filtrada pela faixa de vacinação e organizada pela coluna `N`, apresenta nas 5 primeiras observações as 5 menores observações de cada grupo. Agora basta avaliar essas *querys* e tomar a sua `UNION` como resultado, uma vez que possuem as mesmas colunas (equivalente a um `rbind` do `R`).

```
WITH tabFaixa AS ({last_query})
SELECT * FROM (SELECT * FROM tabFaixa WHERE Faixa='Baixo' ORDER BY N ASC LIMIT 5)
UNION
SELECT * FROM (SELECT * FROM tabFaixa WHERE Faixa='Alto' ORDER BY N ASC LIMIT 5)
ORDER BY N
```

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

Solução: De forma análoga aos itens **a)** e **b)**, o pacote `mongolite` permite a conexão entre o ambiente do `R` e bases de dados `MongoDB`. A função `mongo` abre a conexão com a `db` e cria a coleção (tabela) caso necessário, já o método `$insert` envia dados em formato `JSON` através da conexão, tendo algumas facilidades para enviar diretamente um `data.frame` do `R` sem a necessidade de primeiro converter o objeto para `JSON`.

```
mongo_local <- function(collection) {
  conn <- mongo(
    collection = collection,
    db = "Lista2_db",
    url = "mongodb://localhost"
  )
  return(conn)
}

conn_vax <- mongo_local("vax")
conn_vax$insert(vax_all)

conn_codigos <- mongo_local("codigos")
conn_codigos$insert(codigos)
```

Para realizar uma sequência de operações é conveniente utilizar o método `$aggregate`, método que utiliza uma pipeline escrita em uma extensão de sintaxe para `JSON` para executar vários comandos em sequência, de forma que o resultado do anterior é levado para a execução do próximo método. O método `$lookup` faz o equivalente a um `LEFT JOIN` do `SQL`, o `$project` é semelhante ao `$find` e nesse caso escolhe quais colunas devem ser retornadas (assim como o `select` do `dplyr`), o `$match` atua de forma semelhante ao `filter` do `dplyr`, `$group` é um pouco mais *tricky* e atua como um `group by` juntamente a outros operadores (nesse caso o operador `$sum`, que está contando observações).

```
nt_vax <- conn_vax$aggregate('[
    {
        "$lookup":
        {
            "from": "codigos",
            "localField": "estabelecimento_municipio_codigo",
            "foreignField": "est_mun_codigo",
            "as": "regDocs"
        }
    },
    {
        "$project":{
```

```
            "vacina_descricao_dose": 1,
            "estabelecimento_uf": 1,
            "name_health_region": "$regDocs.name_health_region"
        }
    },
    { "$match": {
        "$or": [
            {"vacina_descricao_dose": "2Âª Dose"},
            {"vacina_descricao_dose": "2Âª Dose RevacinaÃ§Ã£o"}
            ]
        }
    },
    { "$group": {
        "_id": "$name_health_region",
        "N": {"$sum": 1}
        }
    }
]') %>%
  rename("Nome" = `_id`)
```

Diferente do `SQL`, optou-se por não seguir de forma direta em uma única *query*, mas sim por intrromper o procedimento por aqui para trazer o banco para o ambiente do R e computar a mediana do número de observações, uma vez que não há implementação "base" e nem trivial do cálculo da mediana em MongoDB.

```
conn_qnt_vax <- mongo_local("qnt_vax")
conn_qnt_vax$insert(qnt_vax)
mediana <- mediana(conn_qnt_vax$find()$N)
```

Com o banco contendo a quantidade de vacinados no mongo e dispondo da mediana, seguiu-se tomando cuidado com a notação do pacote `glue` para colocar "{}" reais e {} que chamam variável, isto é,

```
conn_qnt_vax$update(
  query = "{}",
  update = glue('[{{
        "$set": {{
            "Faixa": {{
                "$switch": {{
                    "branches": [
                        {{ "case":
                            {{"$gte": ["$N", {mediana}]}},
                            "then": "Alto" }},
                        {{ "case":
                            {{"$lt": ["$N", {mediana}]}},
                            "then": "Baixo" }}
                    ]
                }}
            }}
        }}
    }}]'),
  multiple = TRUE,
  upsert = TRUE
)
```

Por fim, tomou-se o método `$find` para encontrar as 5 primeiras observações de cada grupo ordenadas por `N` e com ajuda da função `bind_rows` obteve-se um resultado análogo ao encontrado na Lista 1.

```
n <- 5
bot5 <- bind_rows(
  conn_qnt_vax$find(
    limit = n, sort = '{"N": 1}'
  ),
  conn_qnt_vax$find(
    skip = conn_qnt_vax$count() - n,
    limit = n, sort = '{"N": 1}'
  )
)
```

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

Solução: O primeiro passo é abrir a conexão com o `Spark`, que pode ser feita como a seguir pelo pacote `sparklyr`

```
sc <- spark_connect(master = "local", version = "3.0.0")
```

A partir desse ponto há diversas formas de subir os dados para o `Spark`, pode-se usar a função `copy_to` para enviar um `dataframe` do `R`

```
tbl_cod <- copy_to(sc, codigos, name = "codigos", overwrite = TRUE)
```

porém esse método tem limitações para enviar dados muito grandes e após tentar enviar o banco carregado e enfrentar mensagens de erro foi necessária uma mudança de método. É possível enviar os dados pelo `spark_read_csv` em uma chamada do `purrr::map`, uma vez que o `sparklyr` apresentou erro ao tentar passar um vetor com o caminho para os arquivos de uma só vez, uma consequência disso é ter que mandar uma *query* para o `Spark` fazendo a união dos bancos lidos.

```
tbl_vax <- map(
  arquivos,
  ~ spark_read_csv(
    sc = sc, path = .x,
    delimiter = ";"
  )
) %>%
  reduce(dplyr::union_all)
```

Lembrando que o `Spark` aceita *querys* escritas em `SQL`, é intuitivo pensar em utilizar as *querys* do item **b)** com algumas adaptações, isto é, vamos passar o banco gerado do procedimento de `union_all` para a *query* `SQL` que no item **b)** filtrava a tabela "vax", mas qual o nome da tabela que criamos em `Spark` anteriormente? Essa é uma pergunta da qual imagina-se que a reposta não seja previsível, não obstante, a *API* do `Spark SQL` dispõe do padrão ___THIS___ que aponta para a tabela atual

```
SELECT * FROM __THIS__
WHERE (
  vacina_descricao_dose='2ª Dose' OR
  vacina_descricao_dose='2ª Dose Revacinação')
```

No item **b)** havia um problema de desempenho com o `LEFT JOIN`, mas agora a situação é outra. O procedimento de `JOIN` não será feito pelo `SQL` mesmo passando a *query* em `SQL` para o `Spark`, ela será traduzida e então executada, portanto o melhor jeito de informar esse procedimento é com o próprio `LEFT JOIN`

```
SELECT name_health_region as regiao_saude
FROM ({last_query}) as seg_dose
LEFT JOIN codigos
ON seg_dose.estabelecimento_municipio_codigo
  = codigos.est_mun_codigo
```

Para calcular a mediana teve-se outra mudança: a função `MEDIAN` não é corretamente traduzida para o `Spark`, o que geraria um erro. O problema pode ser resolvido utilizando uma função nativa do `Spark`, nesse caso a função `percentile_approx` que calcula os percentis solicitados

```
SELECT regiao_saude, COUNT(*) AS N
FROM ({last_query})
GROUP BY regiao_saude

WITH qntVax AS ({last_query})
SELECT regiao_saude, N,
  CASE WHEN N > (
    SELECT percentile_approx(N, 0.5)
    FROM qntVax)
  THEN 'Alto'
  ELSE 'Baixo'
  END AS Faixa
FROM qntVax
```

Também há funções mais adequadas do `Spark` para permitir encontrar os casos em que houveram menos vacinados por faixa de vacinação, nesse caso a função `dense_rank` em conjunto com a *window function* do `SQL`, como sugerido em *https://stackoverflow.com/questions/36660625/spark-sql-top-n-per-group*.

```
WITH tabFaixa AS ({last_query})
SELECT regiao_saude, N, Faixa
FROM (
  SELECT *, dense_rank()
  OVER (PARTITION BY Faixa ORDER BY N DESC) as posicao
  FROM tabFaixa)
WHERE posicao <= 5
```

Agora é questão de enviar essas *querys* para o `Spark` através da função `ft_sql_transformer` e executar um `collect` para trazer os resultados de volta ao `R`

```
##################### Querys adapatadas para o Spark
seg_dose_query <- glue("SELECT * FROM __THIS__
                   WHERE (vacina_descricao_dose='2ª Dose'
                   OR vacina_descricao_dose='2ª Dose Revacinação')")

fast_join <- glue("SELECT name_health_region as regiao_saude
                   FROM ({seg_dose_query}) as seg_dose
                   LEFT JOIN codigos
                   ON seg_dose.estabelecimento_municipio_codigo
                       = codigos.est_mun_codigo")

qnt_vax_query <- glue("SELECT regiao_saude, COUNT(*) AS N
                       FROM ({fast_join})
                       GROUP BY regiao_saude")

faixa_query <- glue("WITH qntVax AS ({qnt_vax_query})
```

```
                    SELECT regiao_saude, N,
                        CASE WHEN N > (
                            SELECT percentile_approx(N, 0.5)
                            FROM qntVax)
                            THEN 'Alto'
                            ELSE 'Baixo'
                            END AS Faixa
                    FROM qntVax")

bot5_query <- glue("WITH tabFaixa AS ({faixa_query})
                    SELECT regiao_saude, N, Faixa FROM
                    (
                        SELECT *, dense_rank()
                        OVER (PARTITION BY Faixa ORDER BY N DESC) as posicao
                        FROM tabFaixa
                    )
                    WHERE posicao <= 5")

##################### Resultados em Memória
tbl_vax %>%
  ft_sql_transformer(bot5_query) %>%
  collect()
```

e) Compare o tempo de processamento das 3 abordagens (SQLite, MongoDB e Spark), desde o envio do comando sql até o recebimento dos resultados no `R`. Comente os resultados incluindo na análise os resultados 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).

Solução:

Tabela 1: Benchmark dos procedimentos utilizados nos itens **b)**, **c)** e **d)**.

Solução	Mínimo (s)	Média (s)	Máximo (s)
SQLite	338.1	339.9	343
Mongo	68	78.3	83.7
Spark	19.7	19.9	20.5

Tabela 2: Benchmark do tempo de execução das funções da Lista 1 para lidar com o banco de vacinados e regiões de saúde.

Pacote	Mínimo (s)	Média (s)	Máximo (s)
data.table	92	93	94
dtplyr	100	103	106
dplyr	161	164	168

Comparando então com a minha solução da Lista 1 é possível observar que todos os métodos apresentados na Lista 2 não se saíram mal em tempo computacional, o que acontece é que todos tiveram de passar por um trabalho há mais para concluir o *join*, possivelmente essa foi a diferença entre o `Spark` desempenhar melhor que o `data.table` e pior que o `dplyr`.

Comparando os métodos da Lista 2 apenas entre eles, observa-se que o `Spark` teve o melhor desempenho, mesmo tendo que incluir no benchmark a concatenação de cada um dos arquivos `csv`.

Repositório Contendo o Código Fonte

https://github.com/Voz-bonita/Big-Data-Compute