巨大なデータが SQL サーバーにあるときに、R でどう立ち向かうかマニュアル：dbplyr パッケージを中心として

遠山祐太

Last updated: 2021-05-20

1　はじめに

このノートでは、SQL サーバーに非常に大きなデータ (例えば観測数が 1 億以上) が格納されているときに、RStudio を使ってどのように分析を行うかについて解説していきます。想定している環境は以下のとおりです。

1. 手元の PC：ブラウザで以下の RStudio Server
2. RStudio Server が入っている外部サーバー
3. データが格納されている SQL サーバー

SQL サーバーに格納されているデータが非常に大きく、RStudio Server にデータを落とすことができない、もしくは落としても作業に非常に時間がかかる状況があります。

一つの方法としては、SQL のクエリーを書いて、データを加工・集約した上で、最終的に RStudio Server 上で図表を作成する・回帰分析をするというものになります。この場合、SQL のクエリーを別途学ぶ必要があり、SQL 初学者にはコストが高いものとなります。

本ノートでは、R の `dplyr` パッケージの文法を使って SQL データフレームを加工できる `dbplyr` パッケージの活用方法について紹介していきます。

最初に `dplyr` の使い方について最低限の事項を説明しますが、既にご存知の方はスキップして大丈夫です。その上で、`dbplyr` 及び関連するデータベースパッケージについて説明していきます。最後に、Rstudio Server で作業する際の注意点、特にメモリ管理などについて説明します。

1.1　読む上での注意点

- パッケージ内の関数を読み込む際には、(1) `library()` でパッケージを読み込んだあとにそのパッケージ内の関数を使う、(2) パッケージ名`::`関数名という表記で関数を呼び出す、という 2 つの方法があります。後者の方が冗長になりますが、どのパッケージからその関数を呼んでいるかという対応関係がわかりやすくなります。以下のノートでは 2 通りの表記を混ぜておりますが、ご了承下さい。

- 多分に我流な方法になっていると思いますので、気がついた点・サジェストなどありましたら、お気軽にメールを頂ければと思います。ホームページ：https://yutatoyama.github.io/
- あくまで個人的なノートであり、授業資料ではありません。このノートに従って発生した損害等についてはいかなる責任も負いません。

1.2 変更履歴

- 2020 年 9 月 20 日：初版
- 2021 年 5 月 11 日：SQL サーバー接続時の ID とパスワード入力について加筆修正しました。

2 dplyr によるデータ整形

dplyr パッケージは tidyverse に含まれているデータ加工・整形のためのパッケージで、直感的な作業がしやすく、データ分析においては事実上標準パッケージと読んでも差し支えないでしょう。

```
library("tidyverse")
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.3     v purrr   0.3.4
## v tibble  3.1.1     v dplyr   1.0.5
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

なお、ここでは必要最低限の説明しかしておりません。dplyr の解説はオンラインで多数ありますが、Jaehyun Song さんの https://www.jaysong.net/dplyr_intro/ がオススメです。

2.1 データの変形

ここでは、R に最初から入っているデータ iris を使います。

2.1.1 変数の作成: mutate

変数を作成します。STATA での gen コマンドです。1 つ目の引数にデータフレーム、2 つ目の引数に「新しい変数名＝変数の定義」を書きます。

```
dt <- mutate(iris, lengthsq = Sepal.Length^2)
```

ここでは、新しいデータフレーム dt を作っており、その中には元の変数に加えて、lengthsq が入っています。

同じことをやる別の書き方として

```
mutate(iris, lengthsq = Sepal.Length^2) -> dt
```

があります。

また、元のデータフレーム iris に上書きしたい場合には、iris そのものをアウトプットを格納する変数とします。

```
iris <- mutate(iris, lengthsq = Sepal.Length^2)
```

2.1.2 観測 (行) の選択: filter

ある条件を満たす行のみを取得します。STATA の if 文です。1 つ目の引数にデータフレーム、2 つ目以降に条件文を書きます。

```
iris2 <- filter(iris, Sepal.Length > 5)
```

2.1.3 変数 (列) の選択: select

必要な変数だけを Keep します。Stata の keep コマンドです。1 つ目の引数にデータフレーム、2 つ目以降に Keep する変数名を入れます。

```
iris3 <- select(iris2, Sepal.Length, Petal.Width, Species)
```

2.2 パイプ演算子%>%

パイプ演算子は、パイプの前の要素を、パイプの後の関数の 1 つ目の引数に入れるというものです。これまで紹介してきた関数はすべて 1 つ目の引数としてデータフレームをとります。従って、これらの操作はすべてパイプ演算子でつなぐことができます。

```
iris %>%
    mutate( lengthsq = Sepal.Length^2 ) %>%
    filter( Sepal.Length > 5) %>%
    select( Sepal.Length, Petal.Width, Species) -> iris3

# もしくは、最終的なアウトプットである変数 iris3 を最初に書く

iris3 <- iris %>%
    mutate( lengthsq = Sepal.Length^2 ) %>%
    filter( Sepal.Length > 5) %>%
    select( Sepal.Length, Petal.Width, Species)
```

2.3 データフレームの集約

iris の各種変数を、Species というグループごとに集約しましょう

```
iris %>%
    group_by(Species) %>%
    summarise( mean_length = mean(Sepal.Length),
               mean_width  = mean(Sepal.Width)) %>%
    ungroup() -> mean_iris

print(mean_iris)
```

```
## # A tibble: 3 x 3
##   Species    mean_length mean_width
##   <fct>            <dbl>      <dbl>
## 1 setosa            5.01       3.43
## 2 versicolor        5.94       2.77
## 3 virginica         6.59       2.97
```

ここでは、Species ごとの平均値ということでアウトプットが 3 行となっています。

では、もしデータを集約せずに、あくまで各行ごとに「該当するグループの平均値」という変数を新たに追加する場合はどうしましょう？このときは mutate を使います。

```
iris %>%
    group_by(Species) %>%
    mutate( mean_length = mean(Sepal.Length),
               mean_width  = mean(Sepal.Width)) %>%
    ungroup() -> mutate_iris

print(mutate_iris)
```

```
## # A tibble: 150 x 8
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species lengthsq
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>      <dbl>
##  1          5.1         3.5          1.4         0.2 setosa      26.0
##  2          4.9         3            1.4         0.2 setosa      24.0
##  3          4.7         3.2          1.3         0.2 setosa      22.1
##  4          4.6         3.1          1.5         0.2 setosa      21.2
##  5          5           3.6          1.4         0.2 setosa      25
##  6          5.4         3.9          1.7         0.4 setosa      29.2
```

```
##  7          4.6         3.4          1.4         0.3 setosa      21.2
##  8          5           3.4          1.5         0.2 setosa      25
##  9          4.4         2.9          1.4         0.2 setosa      19.4
## 10          4.9         3.1          1.5         0.1 setosa      24.0
## # ... with 140 more rows, and 2 more variables: mean_length <dbl>,
## #   mean_width <dbl>
```

なお、group_by をして集約した後には、**必ず ungroup()** をしましょう。しないと変なことが起きたりします。

2.4 データフレームのマージ (結合)

2 つのデータフレームを、特定の変数をキーとして結合しましょう。説明は少し力つきたので、細かい説明は以下のページを参照してください。わかりやすいです。https://qiita.com/matsuou1/items/b1bd9778610e3a586e71

個人的なアドバイスとしては、**`left_join()` だけを使うようにする**というものです。4 つもありますが、多くの場合は `left_join()` で事足ります。あと、`left_join()` はパイプ演算にも組み込みやすいです。

3 dbplyr を用いた SQL サーバー上のデータの処理

それでは、`dplyr` パッケージを活用して、SQL サーバー上のデータの処理を

3.1 下準備: SQL データベースへの接続

まず SQL データベースへ接続しましょう。接続には、データベース接続用の `DBI` パッケージを使います。ここでは、具体的に PostgreSQL サーバーを念頭において作業します。以下のパッケージをインストール及びロードします。

```
library('DBI')
library('RPostgreSQL')
library('dbplyr')
```

```
##
## Attaching package: 'dbplyr'
```

```
## The following objects are masked from 'package:dplyr':
##
##     ident, sql
```

```
library('RSQLite')
```

なお、RPostgreSQL を読み込むと DBI も自動に読み込まれるため、library('DBI') はなくても大丈夫です。また、RSQLite はこのノートで「ローカルな SQL サーバー」を用意するためだけに使うので、実際の環境での分析の際には読み込む必要はない (と思います)。

では、まず R を PostgreSQL サーバに接続しましょう。接続には dbConnect() を使います。関数の中身には接続情報を適宜入れます。

```
con <- dbConnect(PostgreSQL(), host="XXXX",
                 port=9999,
                 user= rstudioapi::showPrompt("UserID", "Put your userID"),
                 password=rstudioapi::askForPassword(""),
                 dbname="db_name")
```

ここで、dbConnect 関数における user と password については、ポップアップ画面において入力する形になります。**決してコードにユーザー ID やパスワードを直書きしてはいけません。**ここでできた変数 con には当該サーバーへの接続情報が入ることになります。

さて、本ノートでは便宜上、パソコン内に仮置の「SQL サーバー」を作成し、そこへの接続を用意しましょう。

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
```

今この仮置の「SQL サーバー」(これを in-memory server と呼びます) には何も入っていません。ですので、copy_to() を使って、mtcars と iris という R に最初から入っているデータフレームを入れましょう。

```
dplyr::copy_to(con, mtcars)
dplyr::copy_to(con, iris)
```

では、この接続した SQL サーバーの中身を見てみましょう。

```
dbListTables(con)  # テーブル一覧取得
```

```
## [1] "iris"         "mtcars"       "sqlite_stat1" "sqlite_stat4"
```

ここで、サーバーの中に入っている要素を**テーブル**と呼びます。iris と mtcars がテーブルとして入っているのがわかります。(他の 2 つはひとまず無視しましょう)

3.2 SQL サーバーでのデータの加工

では、この SQL サーバーに入ったデータをどのようにして R で加工・分析していきましょう？一番単純であり、**実践で決してやってはいけない**ことは、このテーブルそのものを SQL サーバーから R へダウンロードすることです。DBI::dbReadTable を使いましょう。

```
cars_db = DBI::dbReadTable(con, "mtcars")
```

こうすることで、SQL サーバー内にあった mtcars というテーブルが、cars_db という変数で R 上で利用可能になっていることがわかります。しかしながら、実践上は SQL サーバー内のテーブルは非常に大きいこと

が多々あり、元のデータを手元にダウンロードすることは**ほぼほぼ不可能です。** (逆にいうと、元データが手元で操作可能なサイズであればわざわざ SQL サーバーにデータが格納されていることはありません。)

そこで、SQL サーバーからデータを呼び出す際には、**クエリ**と呼ばれるデータ抽出・操作のための命令を書くこととなります。クエリ自体は SQL における文法ですが、R においてもクエリを実行するための関数があります。

以下では `dbGetQuery` 関数を使います。1 つ目の引数がサーバーへの接続 `con`、2 つ目が実行したいクエリを文字列で書きます。

```
cars_2 <- dbGetQuery(con, "SELECT * FROM mtcars WHERE cyl = 8")
```

ここでは、mtcars というテーブルから、"cyl = 8" という条件を満たすものだけを抽出するという作業を行っています。抽出したデータは cars_2 という変数で R に保存されます。

ということで、SQL に詳しい方でしたら、説明はここで終了です。しかしながら、R ユーザーの多くは SQL の使い方に慣れているとは限りませんし、また、より複雑な抽出やテーブル間の結合を行う場合、SQL で書くとよりクエリが複雑になっていきます。

そこで `dbplyr()` パッケージの出番です。このパッケージは dplyr と同じ文法で SQL データベース上のテーブルの加工・抽出を行います。より具体的には、ユーザーが dplyr の文法で行いたい加工抽出を書くと、それを SQL のクエリに変換し、SQL サーバー上で実行するという仕組みになっています。

なお、dbplyr で対応できないような加工もありますので、SQL のクエリの文法で一部コードを書く必要も出てくるかもしれません。dbplyr()

3.3 `dbplyr()` パッケージ

基本的な文法は dplyr と同じですが、いくつか特殊なポイントがあります。以下の三段階で説明していきます。

1. SQL サーバー上のテーブルへの「リンク」を作成
2. dplyr 文法で抽出・加工を指示する
3. 上の指示を実際に「SQL サーバー上で実行」し、「R にダウンロード」する。

3.3.1 Step 1: テーブルへの参照を作成

まず、SQL サーバー上にあるテーブルへの「参照」を作ります。

```
dt_car = tbl(con, "mtcars")
```

ここで dt_car という変数ができました。さて、ここで注意すべきは dt_car は**データフレームそのもの**ではなく、**SQL サーバー内にあるテーブルへの参照**となっているということです。すなわち、この時点で、mtcars のデータそのものが R に落ちているわけではないのです。

例えば、dt_car のなかの変数 mpg をベクトルとして取得してみましょう。すると、

```
dt_car$mpg
```

```
## NULL
```

となり、NULL が返ってきます。これは、dt_car にはデータの中身自体が入っていないためです。あくまで dt_car は SQL 内のテーブル mtcars へのリンクになっているのです。

3.3.2 Step 2: データの加工・抽出の指示

実際に SQL サーバーから R にどのようにデータを落とすかは後にして、まず dbplyr を用いたデータの加工について説明していきましょう。例えば、cyl=8 のもののみを抽出するという作業は filter を使ってできます。

```
dt_car %>%
    filter(cyl == 8) -> dt_car2
```

ここで filter を適用したあとの結果を dt_car2 として保存しています。この dt_car2 も上と同様、データそのものではありません。実はこの dt_car2 には、**filter をかける作業に該当する SQL クエリ**が含まれているのです。これを見るために、show_query を使いましょう。

```
show_query(dt_car2)
```

```
## <SQL>
## SELECT *
## FROM `mtcars`
## WHERE (`cyl` = 8.0)
```

このクエリ文は、前に dbGetQuery で書いたものとほぼ同じですね。

ということで、dplyr() を使うことで、それと同義のクエリ指示を自動に作成してくれるのが dplyr() の肝となります。より細かい&複雑なクエリ指示も書くことができます。

```
dt_car %>%
    filter(cyl == 8) %>%
    filter(mpg  > 15) %>%
    select(mpg, cyl, disp, hp) %>%
    mutate( mpg_squared = mpg*mpg) -> dt_car3

show_query(dt_car3)
```

```
## <SQL>
## SELECT `mpg`, `cyl`, `disp`, `hp`, `mpg` * `mpg` AS `mpg_squared`
## FROM (SELECT *
## FROM `mtcars`
```

```
## WHERE (`cyl` = 8.0))
## WHERE (`mpg` > 15.0)
```

3.3.3 Step 3: 指示を「SQL サーバーで実行」し、「R にダウンロード」する。

上で書いた dt_car2 と dt_car3 はあくまで SQL クエリであり、実際のデータとはなっていません。そのクエリを実行をどうするかについてここで説明していきます。

まず、「クエリを実行し、実行結果を SQL サーバーに保存する」関数として compute() があります。

```
compute(dt_car3, name = "car_computed")
```

```
## # Source:   lazy query [?? x 5]
## # Database: sqlite 3.35.5 [:memory:]
##     mpg   cyl  disp    hp mpg_squared
##   <dbl> <dbl> <dbl> <dbl>       <dbl>
## 1  18.7     8  360    175        350.
## 2  16.4     8  276.   180        269.
## 3  17.3     8  276.   180        299.
## 4  15.2     8  276.   180        231.
## 5  15.5     8  318    150        240.
## 6  15.2     8  304    150        231.
## 7  19.2     8  400    175        369.
## 8  15.8     8  351    264        250.
```

ここで、SQL サーバーの中身を見てみましょう。

```
dbListTables(con)  # テーブル一覧取得
```

```
## [1] "car_computed" "iris"         "mtcars"       "sqlite_stat1" "sqlite_stat4"
```

ということで、新しいテーブル car_computed ができているのがわかります。ここで留意点としては、ここで作った car_computed は「一時的」なテーブルであり、DBI::dbConnect などで SQL サーバーへ接続し直したりするとテーブルは消えています。

では、手元の RStudio にダウンロードして変数として扱うにはどうすれば良いでしょうか？一つは、compute して SQL サーバーに作成したテーブルを dbReadTable で落とすという方法です。もう一つの方法が、compute とダウンロードを同時に行う collect() というものです。

```
dt_car3 %>%
     collect() -> computed_car

#他の書き方として
# computed_car <- collect(dt_car3)
```

すると、computed_car という新しいデータフレームが R に落ちているのが確認できます。

3.4 細かい Tips

以上が dbplyr() を使った作業の大枠の Step です。基本的には、dbplyr() を使って加工・抽出・集約を繰り返し、十分テーブルが小さくなった時点で、collect() で手元に落とすというのが肝です。

大原則は、collect() する前に、SQL で極限まで加工することです。同じ加工でも、SQL 上でやった方が、Rstudio でやるよりも早いです。

(以下個人的な経験。あくまで参考。) 観測数が 1 億・変数（列数）が 4 のデータフレームで、RStudio 上でおおよそ 1GB になりました。このフレームを RStudio 上で加工することも可能でしたが、そこそこ時間がかかる、そしてサーバー落ちにつながるなどもしばしばありました。なお、メモリは 32GB です。

以下では、私が作業していて気がついたコツのようなものを説明していきます。

3.4.1 Tips 1: compute/collect する前のデータの様子を見る方法

中身の先頭をみる方法として head() があります。

```
head(dt_car3, 3)
```

```
## # Source:   lazy query [?? x 5]
## # Database: sqlite 3.35.5 [:memory:]
##     mpg   cyl  disp    hp mpg_squared
##   <dbl> <dbl> <dbl> <dbl>       <dbl>
## 1  18.7     8  360    175        350.
## 2  16.4     8  276.   180        269.
## 3  17.3     8  276.   180        299.
```

なお、ここでは実際に SQL サーバー上で dt_car3 のクエリを実行し、実行した結果の先頭 3 行のみを返しています。クエリを SQL で実行しているので時間が多少かかるかもしれませんが、R の方に落としているのはわずか 3 行分なのでダウンロードにはほとんど時間がかかっていません。（この点は Tips 2 でも触れます。）

また、変数の長さを見たい場合には count 関数を使います。

```
count(dt_car3)
```

```
## # Source:   lazy query [?? x 1]
## # Database: sqlite 3.35.5 [:memory:]
##       n
##   <int>
## 1     8
```

3.4.2 Tips 2: どこの部分で計算に時間がかかるか？

少し細かい点ですが、計算や分析をスムーズに行うために重要な点です。

dbplyr() での作業は

1. dplyr の文法で抽出加工集約の指示を出す
2. その指示を、SQL サーバー上で実行する
3. 実行した結果できるテーブルを Rstudio に落とす

という 3 ステップです。

1 つめのステップはコードを書くだけなのでほとんど時間はかかりません。たとえば、dt_car3 はデータフレームそのものではなく、あくまで SQL クエリが入っているだけです。

2 つ目のステップでは、実際にクエリを実行します。データのサイズやクエリの複雑さにもよりますが、そこそこ時間がかかります。これが compute() に対応する部分です。

そして 3 つ目のステップが肝です。ここで落とすテーブルが大きい場合、実行に長時間かかります。同時に、メモリの消費も非常に大きくなります。ですので、ここで落とすファイルを以下に小さくするかが、作業において重要となってきます。

(個人的経験) 例えば同じクエリを compute() して SQL サーバーにテーブルで保管する場合と、collect() をして R にダウンロードする場合、それぞれを比較した際、前者が 120 秒程度で終わったのに対し、後者では 600 秒くらいかかったこともありました。

3.4.3 Tips 3: アウトプットは極力変数に格納しよう。

具体例を出して説明します。

```
dt_car %>%
    filter(cyl == 8) %>%
    filter(mpg  > 15) %>%
    select(mpg, cyl, disp, hp) %>%
    mutate( mpg_squared = mpg*mpg) -> dt_car3

# 別の書き方
dt_car3 <- dt_car %>%
    filter(cyl == 8) %>%
    filter(mpg  > 15) %>%
    select(mpg, cyl, disp, hp) %>%
    mutate( mpg_squared = mpg*mpg)
```

これは、加工した結果を dt_car3 に保存しています。では、最後の`-> dt_car3` がなかった場合どうなるでし

ょうか？

```
dt_car %>%
    filter(cyl == 8) %>%
    filter(mpg  > 15) %>%
    select(mpg, cyl, disp, hp) %>%
    mutate( mpg_squared = mpg*mpg)
```

```
## # Source:   lazy query [?? x 5]
## # Database: sqlite 3.35.5 [:memory:]
##     mpg   cyl  disp    hp mpg_squared
##   <dbl> <dbl> <dbl> <dbl>       <dbl>
## 1  18.7     8  360    175        350.
## 2  16.4     8  276.   180        269.
## 3  17.3     8  276.   180        299.
## 4  15.2     8  276.   180        231.
## 5  15.5     8  318    150        240.
## 6  15.2     8  304    150        231.
## 7  19.2     8  400    175        369.
## 8  15.8     8  351    264        250.
```

これは内部的には、**SQL サーバー上で「クエリを実行」**していることになっています。このノートの例では非常に小さいデータセットなので問題ありませんが、大きいデータの場合にはクエリの実行にも時間がかかります。

これは collect() をする際により重要となります。例えば、

```
dt_car3 %>%
     collect() -> computed_car
```

は collect() したものを computed_car に保存していますが、

```
dt_car3 %>%
     collect()
```

```
## # A tibble: 8 x 5
##     mpg   cyl  disp    hp mpg_squared
##   <dbl> <dbl> <dbl> <dbl>       <dbl>
## 1  18.7     8  360    175        350.
## 2  16.4     8  276.   180        269.
## 3  17.3     8  276.   180        299.
## 4  15.2     8  276.   180        231.
## 5  15.5     8  318    150        240.
```

```
## 6  15.2     8  304    150        231.
## 7  19.2     8  400    175        369.
## 8  15.8     8  351    264        250.
```

の場合は、collect したテーブルをそのままコンソールに流していることになります。こうすると、せっかく時間をかけてダウンロードしたものが無駄になってしまいます。必ずアウトプットを変数に格納するようにしましょう。

3.4.4 Tips 4: よく使う集計方法

あるグループに該当する観測数が何個あるかを示すものです。

```
dt_car %>%
    group_by(cyl) %>%
    tally() %>%
    collect() -> cyl_group
print(cyl_group)
```

```
## # A tibble: 3 x 2
##     cyl     n
##   <dbl> <int>
## 1     4    11
## 2     6     7
## 3     8    14
```

3.4.5 Tips 5: compute() の使い所及びマージの際の注意点

SQL サーバー上のテーブルから加工した新たなテーブル同士を結合（マージ）するときなどには compute して SQL サーバーに一旦テーブルを置いておくと良いでしょう。

なお、dbplyr を使ってテーブル同士をマージする際には、そのテーブルは**両方とも SQL サーバーにないといけません**。つまり、SQL サーバーにあるテーブルと、RStudio 上にあるテーブルを結合はできません。もし Rstudio で作ったテーブルを、SQL 上にあるものとマージしたい場合には、Rstudio のものを `copy_to` 関数で SQL サーバーに移す必要があります。

4 RStudio Server の作業時の注意点。特にメモリ管理について。

1. SQL から Rstudio に落としたデータが大きい場合、メモリ消費量が非常に大きくなります。Rstudio server 環境では、サーバー落ちなどのトラブルに繋がる可能性が高くなります。
2. 大きいファイルを落とした場合（目安としてはサンプルサイズ数千万で変数が５－６個以上）には、メモリの消費量を常に気にするようにしましょう。
3. メモリ消費量は”Terminal” というパネルで `top` とタイプすると、現在稼働しているユーザーのメモリ

消費量を見ることができます。

4. メモリ消費量が多い場合（例えば 50% 以上）には、大きなデータフレームを変数から消すようにしましょう。ただし、`rm()` で変数を消しただけでは、メモリが開かない場合があります。その時は、`gc()` を 2 回やりましょう。

```
gc(reset = TRUE)
gc(reset = TRUE)
```

5. gc をやってもメモリ消費量が減らない場合があります。そのときには、一旦 Rstudio を再起動することを勧めます。