Praca Domowa 1 - Raport

Michał Wdowski

15 kwietnia 2019

Spis treści

1 Wprowadzenie

Poniższy raport stanowi dokumentację mojej pracy nad danymi z pewnego serwisu społecznościowego. Zadaniem do wykonania było odtworzenie siedmiu wywołań w języku SQL (wykonanych przy użyciu pakietu `sqldf`) na trzy sposoby - w bazowych funkcjach języka `R` oraz za pomocą pakietów `dplyr` i `data.table`, które nie były wprowadzone na zajęciach, a więc musiały zostać opanowane samodzielnie.

2 Funkcje

2.1 Zadanie 1.

Przyjrzyjmy się poniższemu zapytaniu w języku SQL:

```
df_sql_1 <- function(Users, Posts){
    sqldf ("SELECT
                Users.DisplayName,
                Users.Age,
                Users.Location,
                SUM(Posts.FavoriteCount) AS FavoriteTotal,
                Posts.Title AS MostFavoriteQuestion,
                MAX(Posts.FavoriteCount) AS MostFavoriteQuestionLikes
            FROM Posts JOIN Users ON Users.Id=Posts.OwnerUserId
            WHERE Posts.PostTypeId=1
            GROUP BY OwnerUserId
            ORDER BY FavoriteTotal DESC
            LIMIT 10") -> final
    return(final)
}
```

Potrzebuje ono danych o użytkownikach (tabela `Users`) i o postach (tabela `Posts`). Zapytanie można zinterpretować następująco - "Dla każdego użytkownika pokaż jego nick (`Users.DisplayName`), wiek (`Users.Age`) i lokalizację (`Users.Location`), a także sumaryczną liczbę polubień (`SUM(Posts.FavoriteCount)`) jego pytań (`Posts.PostTypeId==1`), oraz treść (`Posts.Title`) i wynik najbardziej lubianego pytania (`MAX(Posts.FavoriteCount)`). Następnie rekordy posortuj malejąco według sumy polubień (`ORDER BY FavoriteTotal DESC`) i pokaż pierwsze 10 od góry (`LIMIT 10`)".

Rozwiązaniem tego samego problemu w funkcjach bazowych języka `R` może być funkcja:

```
df_base_1 <- function(Users, Posts){
    #Wydobycie potrzebnych danych z ramki danych Users
    Users <- Users[, c("DisplayName", "Age", "Location", "Id")]

    #Wydobycie potrzebnych danych z ramki danych Posts i obrobka - zamiana NA na 0 i wziecie
    #jedynie postow, ktore sa pytaniami
    Posts$FavoriteCount[is.na(Posts$FavoriteCount)] <- 0
    Posts <- Posts[, c("OwnerUserId", "FavoriteCount", "PostTypeId", "Title")]
    Posts <- Posts[Posts$PostTypeId==1, ]

    #Policzenie sumy polubien dla kazdego uzytkownika i posortowanie ich malejaco
    df1 <- aggregate(x=Posts$FavoriteCount, by=Posts["OwnerUserId"], sum)
    rightOrder <- order(-df1$x)
    df1 <- df1[rightOrder, ]

    #Policzenie najbardziej lubianego postu dla kazdego uzytkownika i posortowanie go
    #wg kolejnosci uzytkownikow z ramki df1
    df2 <- do.call(rbind, lapply(split(Posts, Posts$OwnerUserId),
                   function(chunk) chunk[which.max(chunk$FavoriteCount),]))
    df2 <- df2[rightOrder, ]

    #Polaczenie danych
    final <- cbind(df1, df2)

    #Znalezienie danych uzytkonikow po unikatowym kluczu OwnerUserId
    #Uzylem funkcji match, bo wtedy nie znalem jeszcze merge. Mysle, ze dla celow edukacyjnych
    #(i z wrodzonego lenistwa) zachowam to rozwiazanie
    Users <- Users[match(final$OwnerUserId, Users$Id), ]
    final <- cbind(final, Users)

    #Wybranie kolumn, zmiana nazw, dopasowanie typow
    final <- final[, c("DisplayName", "Age", "Location", "x", "Title", "FavoriteCount")]
    colnames(final) <- c("DisplayName", "Age", "Location",
                         "FavoriteTotal", "MostFavoriteQuestion", "MostFavoriteQuestionLikes")
    return(head(final, n=10))
}
```

Zastosowana prowizorycznie funkcja `match` działa w ten sposób, że dopasowuje rekordy po pierwszym znalezionym pasującym wyniku. Ponieważ szukane wartości były unikatowe, to nie pojawiły się błędy przy dopasowaniu, a wynik wywołania funkcji odpowiada poszukiwanej odpowiedzi:

```
dplyr::all_equal(df_sql_1(Users, Posts), df_base_1(Users, Posts), convert=TRUE)

## [1] TRUE
```

W języku funkcji pakietu `dplyr` rozwiązanie może wyglądać nastepująco:

```
df_dplyr_1 <- function(Users, Posts){
    Posts %>% dplyr::do(replace(., is.na(.), 0)) -> Posts
    Posts %>%
        dplyr::filter(PostTypeId==1) %>%
        dplyr::select(OwnerUserId, FavoriteCount) %>%
        dplyr::group_by(OwnerUserId) %>%
        dplyr::summarise(FavoriteTotal=sum(FavoriteCount)) %>%
        dplyr::arrange(desc(FavoriteTotal)) -> Posts1
    Posts %>%
        dplyr::filter(PostTypeId==1) %>%
        dplyr::select(OwnerUserId, FavoriteCount, Title) %>%
        dplyr::group_by(OwnerUserId, Title) %>%
        dplyr::summarise(MostFavoriteQuestionLikes=max(FavoriteCount)) %>%
        dplyr::filter(MostFavoriteQuestionLikes==max(MostFavoriteQuestionLikes)) %>%
        dplyr::inner_join(Posts1, by=c("OwnerUserId" = "OwnerUserId")) -> Posts2
    Users %>%
        dplyr::select(DisplayName, Age, Location, Id) %>%
        dplyr::inner_join(Posts2, by=c("Id" = "OwnerUserId")) %>%
        dplyr::arrange(desc(FavoriteTotal)) %>%
        dplyr::select(DisplayName, Age, Location, FavoriteTotal,
                      MostFavoriteQuestion=Title, MostFavoriteQuestionLikes) %>%
        dplyr::slice(1:10)-> Users
    return(Users)
}
```

Samokomentujący się kod w `dplyr` nie wymaga komentarza co do tego, co aktualnie jest wykonywane, jednak warto w tym miejscu zwrócić uwagę na sposób odnalezienia sumy polubień i postu z maksymalną liczbą polubień. Mianowicie, postanowiłem rozdzielić te dwie czynności na oddzielne tabele, które zostały później połączone. Tak czy inaczej, również w tym wypadku wyniki się zgadzają:

```
dplyr::all_equal(df_sql_1(Users, Posts), df_dplyr_1(Users, Posts), convert=TRUE)

## [1] TRUE
```

Ostatnim rozpatrywanym sposobem podejścia do stawiania zapytań jest pakiet `data.table`. Działający kod prezentuje się następująco:

```
df_table_1 <- function(Users, Posts){
    Posts[is.na(Posts)] <- 0
    Posts1 <- Posts[PostTypeId==1, .(OwnerUserId, FavoriteCount)
                    ][, .(FavoriteTotal=sum(FavoriteCount)), by=.(OwnerUserId)]
    Posts2 <- Posts[PostTypeId==1, .(OwnerUserId, FavoriteCount, Title)
                    ][, .(MostFavoriteQuestionLikes=max(FavoriteCount)),
                      by=.(OwnerUserId, Title)
                      ][, .SD[which.max(MostFavoriteQuestionLikes)], by=.(OwnerUserId)]
    setkey(Posts1, OwnerUserId)
    setkey(Posts2, OwnerUserId)
    Posts1 <- Posts1[Posts2, nomatch=0]
    setkey(Users, Id)
    Users <- Users[, .(DisplayName, Age, Location, Id)
                   ][Posts1, nomatch=0][order(-FavoriteTotal),
                                        .(DisplayName, Age, Location, FavoriteTotal,
                                          MostFavoriteQuestion=Title, MostFavoriteQuestionLikes)
                                        ][1:10]
    return(Users)
}
```

Pakiet `data.table` tak naprawdę oferuje nam nowy typ danych, alternatywny to używanych ramek danych i trochę inny w składni, jednakże równie użyteczny. Także i to rozwiązanie okazuje się zwracać poprawny wynik.

```
dplyr::all_equal(df_sql_1(Users, Posts), as.data.frame(df_table_1(dtUsers, dtPosts)),
                 convert=TRUE)

## [1] TRUE
```

Przyjrzyjmy się teraz czasom wykonywania poszczególnych funkcji. Do tego wykorzystamy pakiet `microbenchmark`:

```
microbenchmark::microbenchmark(
    sql <- df_sql_1(Users, Posts),
    base <- df_base_1(Users, Posts),
    dplyr <- df_dplyr_1(Users, Posts),
    dtable <- df_table_1(dtUsers, dtPosts),
    times = 20
)

## Unit: milliseconds
##                                    expr       min        lq      mean
##           sql <- df_sql_1(Users, Posts)  376.7845  381.4263  396.8248
##          base <- df_base_1(Users, Posts) 3267.3323 3365.1537 3535.9160
##         dplyr <- df_dplyr_1(Users, Posts) 1017.4583 1085.1614 1171.6893
##  dtable <- df_table_1(dtUsers, dtPosts) 3037.1615 3171.7765 3306.3000
##     median       uq       max neval
##   389.7294  397.985  484.6857    20
##  3478.0284 3631.941 4108.4429    20
##  1146.2465 1166.360 2044.1008    20
##  3263.7950 3362.998 3930.5961    20
```

2.2 Zadanie 2.

Kod SQL-owy, który stanowi treść zadania, wygląda następująco:

```
df_sql_2 <- function(Posts){
    #Pokazuje ID i tytul postow, ktore maja najwiecej pozytywnie ocenianych odpowiedzi
    sqldf ("SELECT
                Posts.ID,
                Posts.Title,
                Posts2.PositiveAnswerCount
            FROM Posts
            JOIN (
                SELECT
                    Posts.ParentID,
                    COUNT(*) AS PositiveAnswerCount
                FROM Posts
                WHERE Posts.PostTypeID=2 AND Posts.Score>0
                GROUP BY Posts.ParentID
            ) AS Posts2
            ON Posts.ID=Posts2.ParentID
            ORDER BY Posts2.PositiveAnswerCount DESC
            LIMIT 10") -> final
    return(final)
}
```

To zapytanie można interpretować następująco - "Pokaż Id , treść i liczbę pozytywnie ocenionych odpowiedzi każdego posta. Następnie posortuj wynik po liczbie pozytywnie ocenionych odpowiedzi i pokaż pierwsze 10 wyników od góry".

W języku bazowych funkcji `R` rozwiązanie wygląda następująco:

```
df_base_2 <- function(Posts){
    #Filtrowanie zeby wydobyc pozytywnie ocenione odpowiedzi
    Posts2 <- Posts[Posts$PostTypeId==2 & Posts$Score>0, ]

    #Zliczenie pozytynie ocenionych odpowiedzi dla kazdego postu i zmiana nazwy kolumny
    Posts2 <- aggregate(x=Posts2$ParentId, by=Posts2["ParentId"], FUN=length)
    colnames(Posts2)[2] <- "PositiveAnswerCount"

    #Wybranie kolumn i polaczenie z Posts2 po Id postu
    Posts <- Posts[, c("Id", "Title")]
    Posts <- merge(x=Posts, y=Posts2, by.x="Id", by.y="ParentId")

    #Sortowanie malejaco
    Posts <- Posts[order(-Posts2$PositiveAnswerCount), ]
    return(head(Posts, n=10))
}
```

Z kolei w `dplyr` można to zapisać w ten sposób:

```
df_dplyr_2 <- function(Posts){
    Posts %>%
        dplyr::filter(PostTypeId==2, Score>0) %>%
        dplyr::group_by(ParentId) %>%
        dplyr::summarise(PositiveAnswerCount=n()) -> Posts2
    Posts %>%
        dplyr::inner_join(Posts2, by=c("Id" = "ParentId")) %>%
        dplyr::select(Id, Title, PositiveAnswerCount) %>%
        dplyr::arrange(desc(PositiveAnswerCount)) %>%
        dplyr::slice(1:10) -> Posts
    return(Posts)
}
```

Taki napisany a pomocą `data.table` kod także osiąga wymagany wynik:

```
df_table_2 <- function(Posts){
    Posts2 <- Posts[PostTypeId==2 & Score>0, .(PositiveAnswerCount=.N), by=.(ParentId)]
    setkey(Posts, Id)
    setkey(Posts2, ParentId)
    Posts <- Posts[Posts2, nomatch=0
                   ][order(-PositiveAnswerCount), c("Id", "Title", "PositiveAnswerCount")
                     ][1:10]
    return(Posts)
}
```

Wszystkie kody kody spełniają swoją rolę:

```
dplyr::all_equal(df_sql_2(Posts), df_base_2(Posts), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_2(Posts), df_dplyr_2(Posts), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_2(Posts), as.data.frame(df_table_2(dtPosts)),
                 convert=TRUE)

## [1] TRUE
```

Zaś czasy prezentują się następująco:

```
microbenchmark::microbenchmark(
    sql <- df_sql_2(Posts),
    base <- df_base_2(Posts),
    dplyr <- df_dplyr_2(Posts),
    dtable <- df_table_2(dtPosts),
    times = 20
)

## Unit: milliseconds
##                           expr       min        lq      mean    median
##         sql <- df_sql_2(Posts) 274.47041 276.11334 280.24472 278.51056
##        base <- df_base_2(Posts) 303.59035 320.33308 332.46959 324.92184
##      dplyr <- df_dplyr_2(Posts)  67.62535  71.08409  78.18279  73.15219
##  dtable <- df_table_2(dtPosts)  20.89833  21.33547  24.82760  22.60371
##          uq       max neval
##  282.93688 297.36912    20
##  329.29105 471.56165    20
##   87.37211 104.30882    20
##   25.52805  41.70245    20
```

2.3 Zadanie 3.

W zadaniu 3. mamy następujące zapytanie:

```
df_sql_3 <- function(Posts, Votes){
    sqldf ("SELECT
                Posts.Title,
                UpVotesPerYear.Year,
                MAX(UpVotesPerYear.Count) AS Count
            FROM (
                SELECT
                    PostId,
                    COUNT(*) AS Count,
                    STRFTIME('%Y', Votes.CreationDate) AS Year
                FROM Votes
                WHERE VoteTypeId=2
                GROUP BY PostId, Year
            ) AS UpVotesPerYear
            JOIN Posts ON Posts.Id=UpVotesPerYear.PostId
            WHERE Posts.PostTypeId=1
            GROUP BY Year") -> final
    return(final)
}
```

Możemy nadać temu następującą interpretację - "Dla każdego roku wstawiania postów pokaż tytuł pytania, które miało największą liczbę głosów typu "2", oraz pokaż liczbę tych głosów.

W bazowych funkcjach zadziała następujący kod:

```
df_base_3 <- function(Posts, Votes){
    #Tworzenie zestawienia glosow dla kazdego postu w zaleznosci od roku
    UpVotesPerYear <- Votes[Votes$VoteTypeId==2, ]
    UpVotesPerYear[, "CreationDate"] <- substr(UpVotesPerYear[, "CreationDate"], 1, 4)
    UpVotesPerYear <- aggregate(UpVotesPerYear$Id,
                                by=list(UpVotesPerYear$PostId, UpVotesPerYear$CreationDate),
                                FUN=length)
    colnames(UpVotesPerYear) <- c("PostId", "Year", "Count")

    #Polaczenie glosow z postami i wybranie maksymalnego wyniku dla kazdego roku
    Posts <- Posts[Posts$PostTypeId==1, ]
    Posts <- Posts[, c("Id", "Title")]
    Posts <- merge(x=Posts, y=UpVotesPerYear, by.x="Id", by.y="PostId")
    Posts <- do.call(rbind, lapply(split(Posts, Posts$Year),
                                   function(chunk) chunk[which.max(chunk$Count),]))
    Posts <- Posts[, c("Title", "Year", "Count")]
    return(Posts)
}
```

Używam funkcji substr, żeby wyciąć pierwsze 4 znaki z pola oznaczającego datę wstawienia postu. Wydaje się to być najbardziej optymalne i inne funckje nie są potrzebne, zwłaszcza że ta data domyślnie została wprowadzona jako typ tekstowy.

W `dplyr` kod może wyglądać w ten sposób:

```
df_dplyr_3 <- function(Posts, Votes){
    Votes %>%
        dplyr::filter(VoteTypeId==2) %>%
        dplyr::mutate(Year=substr(CreationDate, 1, 4)) %>%
        dplyr::group_by(PostId, Year) %>%
        dplyr::summarise(Count=n()) %>%
        dplyr::select(PostId, Year, Count) -> UpVotesPerYear
    Posts %>%
        dplyr::filter(PostTypeId==1) %>%
        dplyr::select(Id, Title) %>%
        dplyr::inner_join(UpVotesPerYear, by=c("Id" = "PostId")) %>%
        dplyr::group_by(Year) %>%
        dplyr::filter(Count==max(Count)) %>%
        dplyr::select(-Id) -> Posts
    return(Posts)
}
```

Zaś w `data.table`:

```
df_table_3 <- function(Posts, Votes){
    UpVotesPerYear <- Votes[VoteTypeId==2, .(PostId, Year=substr(CreationDate, 1, 4))
                            ][, .(Count=.N), by=.(PostId, Year)]
    setkey(Posts, Id)
    setkey(UpVotesPerYear, PostId)
    Posts <- Posts[PostTypeId==1, .(Id, Title)
                   ][UpVotesPerYear, nomatch=0
                     ][, .SD[which.max(Count)], by=.(Year)
                       ][, .(Title, Year, Count)]
    return(Posts)
} #done
```

Wszystkie wyniki są zgodne ze wzorcem:

```
dplyr::all_equal(df_sql_3(Posts, Votes), df_base_3(Posts, Votes), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_3(Posts, Votes), df_dplyr_3(Posts, Votes), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_3(Posts, Votes), as.data.frame(df_table_3(dtPosts, dtVotes)),
                 convert=TRUE)

## [1] TRUE
```

A czasy prezentują się następująco:

```
microbenchmark::microbenchmark(
    sql <- df_sql_3(Posts, Votes),
    base <- df_base_3(Posts, Votes),
    dplyr <- df_dplyr_3(Posts, Votes),
    dtable <- df_table_3(dtPosts, dtVotes),
    times = 20
)

## Unit: milliseconds
##                                     expr       min        lq      mean
##           sql <- df_sql_3(Posts, Votes) 1399.1456 1409.8533 1421.3514
##          base <- df_base_3(Posts, Votes) 2216.8384 2301.7874 2366.1328
##         dplyr <- df_dplyr_3(Posts, Votes)  398.9323  416.6558  436.6075
##  dtable <- df_table_3(dtPosts, dtVotes)  113.2804  128.0349  170.6364
##     median        uq       max neval
##  1419.9995 1431.7701 1449.4064    20
##  2347.3102 2365.6871 2705.6257    20
##   429.8266  453.4366  483.7131    20
##   145.8998  154.1257  476.3888    20
```

2.4 Zadanie 4.

Zadanie 4. ma następującą tresć w SQL:

```
df_sql_4 <- function(Posts){
    sqldf ("SELECT
                Questions.Id,
                Questions.Title,
                BestAnswers.MaxScore,
                Posts.Score AS AcceptedScore,
                BestAnswers.MaxScore-Posts.Score AS Difference
            FROM (
                    SELECT Id, ParentId, MAX(Score) AS MaxScore
                    FROM Posts
                    WHERE PostTypeId==2
                    GROUP BY ParentId
                ) AS BestAnswers
            JOIN (
                    SELECT * FROM Posts
                    WHERE PostTypeId==1
                ) AS Questions
                ON Questions.Id=BestAnswers.ParentId
            JOIN Posts ON Questions.AcceptedAnswerId=Posts.Id
            WHERE Difference>50
            ORDER BY Difference DESC") -> final
    return(final)
}
```

Można to rozumieć w ten sposób - "Dla każdego posta typu "1" pokaż jego Id, treść, wynik najlepiej ocenionej odpowiedzi i wynik wybranej odpowiedzi, a także różnicę między nimi. Wybierz rekordy, które tę różnicę mają większą niż 50 i posortuj wyniki według tej różnicy".

W bazowym R rozwiązanie wygląda następująco:

```
df_base_4 <- function(Posts){
    #Wydobycie potrzebnych danych do stworzenia tabeli BestAnswers
    BestAnswers <- Posts[, c("Id", "ParentId", "PostTypeId","Score")]

    #Przefiltrowanie tak, by w tabeli byly tylko odpowiedzi
    BestAnswers <- BestAnswers[BestAnswers$PostTypeId==2, ]

    #Znalezienie oceny najwyzej ocenianej odpowiedzi dla kazdego zapytania
    BestAnswers <- aggregate(x=BestAnswers$Score, by=BestAnswers["ParentId"], max)

    #Zmiana nazwy kolumny z wynikami dla kazdego ParentId
    colnames(BestAnswers)[2] <- "MaxScore"

    #Przefiltrowanie (aby zostaly tylko pytania), zmiana nazwy kolumny
    #wydobycie potrzebnych danych
    Questions <- Posts[Posts$PostTypeId==1, ]
    colnames(Questions)[8] <- "MainId"
    Questions <- Questions[, c("MainId", "Title", "AcceptedAnswerId")]

    #Polaczenie pytan z najlepszymi pytaniami i z postami
    Questions <- merge(x=Questions, y=BestAnswers, by.x="MainId", by.y="ParentId")
    Questions <- merge(x=Posts, y=Questions, by.x="Id", by.y="AcceptedAnswerId")

    #Stworzenie kolumny z roznica miedzy wynikiem najlepiej ocenianej odpowiedzi
    #a zaakceptowanej odpowiedzi
    Difference <- Questions$MaxScore - Questions$Score
    Questions <- cbind(Questions, Difference)

    #Zmiana nazw i kolejnosci kolumn
    Questions <- Questions[, c("MainId", "Title.y", "MaxScore", "Score", "Difference")]
    colnames(Questions) <- c("Id", "Title", "MaxScore", "AcceptedScore", "Difference")

    #Przefiltrowanie i sortowanie rekordow tak, zeby pokazane byly roznice wieksze niz 50
    #w porzadku malejacym
    Questions <- Questions[Questions$Difference>50, ]
    Questions <- Questions[order(-Questions$Difference), ]
    return(Questions)
}
```

W `dplyr` zadziała następujący kod:

```
df_dplyr_4 <- function(Posts){
    Posts %>%
        dplyr::select(Id, ParentId, PostTypeId, Score) %>%
        dplyr::filter(PostTypeId==2) %>%
        dplyr::group_by(ParentId) %>%
        dplyr::summarise(MaxScore=max(Score)) -> BestAnswers
    Posts %>%
        dplyr::filter(PostTypeId==1) %>%
        dplyr::select(MainId=Id, Title, AcceptedAnswerId) %>%
        dplyr::inner_join(BestAnswers, by=c("MainId" = "ParentId")) %>%
        dplyr::inner_join(Posts, by=c("AcceptedAnswerId" = "Id")) %>%
        dplyr::mutate(Difference = MaxScore - Score) %>%
        dplyr::select(Id=MainId, Title=Title.x, MaxScore, AcceptedScore=Score, Difference) %>%
        dplyr::filter(Difference>50) %>%
        dplyr::arrange(desc(Difference)) -> Questions
    return(Questions)
}
```

W `data.table` możemy zastosować takie rozwiązanie:

```
df_table_4 <- function(Posts){
    BestAnswers <- Posts[PostTypeId==2, c("Id", "ParentId" , "Score")
                         ][, .(MaxScore=max(Score)), by=.(ParentId)]
    Questions <- Posts[PostTypeId==1, .(MainId=Id, Title, AcceptedAnswerId)]

    setkey(Questions, MainId)
    setkey(BestAnswers, ParentId)
    Questions <- Questions[BestAnswers]

    setkey(Questions, AcceptedAnswerId)
    setkey(Posts, Id)
    Questions <- Questions[Posts
                           ][, .(Id=MainId, Title, MaxScore,
                                 AcceptedScore=Score, Difference=(MaxScore-Score))
                             ][Difference>50
                               ][order(-Difference)]
    return(Questions)
} #done
```

Wszystkie wyniki są zgodne ze wzorcem:

```
dplyr::all_equal(df_sql_4(Posts), df_base_4(Posts), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_4(Posts), df_dplyr_4(Posts), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_4(Posts), as.data.frame(df_table_4(dtPosts)),
                 convert=TRUE)

## [1] TRUE
```

A czasy prezentują się następująco:

```
microbenchmark::microbenchmark(
    sql <- df_sql_4(Posts),
    base <- df_base_4(Posts),
    dplyr <- df_dplyr_4(Posts),
    dtable <- df_table_4(dtPosts),
    times = 20
)

## Unit: milliseconds
##                           expr       min        lq      mean    median
##         sql <- df_sql_4(Posts) 343.41994 347.12261 352.39526 349.15787
##        base <- df_base_4(Posts) 330.15668 332.74762 347.59772 344.57334
##       dplyr <- df_dplyr_4(Posts)  96.70906 102.52757 106.18572 104.10745
##  dtable <- df_table_4(dtPosts)  46.03767  46.45867  53.51576  49.70574
##          uq       max neval
##  353.79246 375.46145    20
##  358.17582 397.08379    20
##  106.55603 123.94313    20
##   54.42434  73.71196    20
```

2.5 Zadanie 5.

Zadanie 5. ma następującą tresć w SQL:

```
df_sql_5 <- function(Posts, Comments){
    sqldf ("SELECT
            Posts.Title,
            CmtTotScr.CommentsTotalScore
            FROM (
                SELECT
                PostID,
                UserID,
                SUM(Score) AS CommentsTotalScore
                FROM Comments
                GROUP BY PostID, UserID
            ) AS CmtTotScr
            JOIN Posts ON Posts.ID=CmtTotScr.PostID AND Posts.OwnerUserId=CmtTotScr.UserID
            WHERE Posts.PostTypeId=1
            ORDER BY CmtTotScr.CommentsTotalScore DESC
            LIMIT 10") -> final
    return(final)
}
```

Można to rozumieć w ten sposób - "Pokaż tytuły tych postów typu "1", których autor zdobył największą sumę ocen w komentarzach pod nim, oraz pokaż tę sumę. Następnie posortuj je według tej sumy i pokaż pierwsze 10 wyników od góry".

W bazowym `R` rozwiązanie wygląda następująco:

```
df_base_5 <- function(Posts, Comments){
    #Wybranie potrzebnych kolumn z Comments
    CmtTotScr <- Comments[, c("PostId", "UserId", "Score")]

    #Zliczenie sumy ocen komentarzy dla kazdego posta i uzytkownika, potem zmiana nazwy kolumny
    CmtTotScr <- aggregate(CmtTotScr$Score,
                           list(PostId=CmtTotScr$PostId, UserId=CmtTotScr$UserId), sum)
    colnames(CmtTotScr)[3] <- "CommentsTotalScore"

    #Wybranie tylko postow typu 1 oraz wybranie potrzebnych wierszy
    Posts <- Posts[Posts$PostTypeId==1, ]
    Posts <- Posts[, c("Id", "OwnerUserId", "Title")]

    #Polaczenie Posts i CmtTotScr po uzytkowniku, ktory jest wlascicielem posta
    #oraz po id posta
    Posts <- merge(x=Posts, y=CmtTotScr,
                   by.x=c("OwnerUserId", "Id"), by.y=c("UserId", "PostId"))

    #Wybranie potrzebnych kolumn i sortowanie
    Posts <- Posts[, c("Title", "CommentsTotalScore")]
    Posts <- Posts[order(-Posts$CommentsTotalScore), ]
    return(head(Posts, n=10))
}
```

W `dplyr` zadziała następujący kod:

```
df_dplyr_5 <- function(Posts, Comments){
    Comments %>%
        dplyr::select(PostId, UserId, Score) %>%
        dplyr::group_by(PostId, UserId) %>%
        dplyr::summarise(CommentsTotalScore=sum(Score)) -> CmtTotScr
    Posts %>%
        dplyr::filter(PostTypeId==1) %>%
        dplyr::select(Id, OwnerUserId, Title) %>%
        dplyr::inner_join(CmtTotScr, by=c("Id" = "PostId", "OwnerUserId" = "UserId")) %>%
        dplyr::select(Title, CommentsTotalScore) %>%
        dplyr::arrange(desc(CommentsTotalScore)) %>%
        dplyr::slice(1:10) -> Posts
    return(Posts)
}
```

W `data.table` możemy zastosować takie rozwiązanie:

```
df_table_5 <- function(Posts, Comments){
    CmtTotScr <- Comments[, .(CommentsTotalScore=sum(Score)), by=.(PostId, UserId)]
    setkey(Posts, Id, OwnerUserId)
    setkey(CmtTotScr, PostId, UserId)
    Posts <- Posts[PostTypeId==1, c("Id", "OwnerUserId", "Title")
                   ][CmtTotScr, nomatch=0
                     ][order(-CommentsTotalScore), c("Title", "CommentsTotalScore")
                       ][1:10]
    return(Posts)
}
```

Wszystkie wyniki są zgodne ze wzorcem:

```
dplyr::all_equal(df_sql_5(Posts, Comments), df_base_5(Posts, Comments), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_5(Posts, Comments), df_dplyr_5(Posts, Comments), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_5(Posts, Comments), as.data.frame(df_table_5(dtPosts, dtComments)),
                 convert=TRUE)

## [1] TRUE
```

A czasy prezentują się następująco:

```
microbenchmark::microbenchmark(
    sql <- df_sql_5(Posts, Comments),
    base <- df_base_5(Posts, Comments),
    dplyr <- df_dplyr_5(Posts, Comments),
    dtable <- df_table_5(dtPosts, dtComments),
    times = 20
)

## Unit: milliseconds
##                                       expr        min         lq       mean
##           sql <- df_sql_5(Posts, Comments)  630.29903  632.90648  649.7615
##          base <- df_base_5(Posts, Comments) 2622.21240 2660.25848 2738.1732
##         dplyr <- df_dplyr_5(Posts, Comments)  327.52171  344.92718  370.4864
##  dtable <- df_table_5(dtPosts, dtComments)   37.20456   37.53168   43.4695
##      median         uq       max neval
##   639.76045  656.09932  742.2674    20
##  2702.93178 2790.60134 2973.9698    20
##   369.89733  379.82715  521.4134    20
##    40.16846   42.60631  109.0292    20
```

2.6 Zadanie 6.

Zadanie 6. ma następującą tresć w SQL:

```
df_sql_6 <- function(Users, Badges){
    sqldf ("SELECT DISTINCT
                Users.Id,
                Users.DisplayName,
                Users.Reputation,
                Users.Age,
                Users.Location
            FROM (
                    SELECT
                        Name, UserID
                    FROM Badges
                    WHERE Name IN (
                        SELECT
                            Name
                        FROM Badges
                        WHERE Class=1
                        GROUP BY Name
                        HAVING COUNT(*) BETWEEN 2 AND 10
                    )
                    AND Class=1
                ) AS ValuableBadges
            JOIN Users ON ValuableBadges.UserId=Users.Id") -> final
    return(final)
}
```

Można to rozumieć w ten sposób - "Pokaż Id, nick, reputację, wiek i lokalizację tych użytkowników, którzy zdobyli medale klasy 1, które zostaly zdobyte od 2 do 10 razy".

W bazowym R rozwiązanie wygląda następująco:

```
df_base_6 <- function(Users, Badges){
    #Wydobycie tylko medale klasy 1
    FirstClassBadges <- Badges[Badges$Class==1, ]

    #Tworzenie zestawienia zdobyc medalu w zaleznowsi od nazwy, potem ustawienie nazw kolumn
    FirstClassBadges <- as.data.frame(table(FirstClassBadges$Name), stringsAsFactors=FALSE)
    colnames(FirstClassBadges) <- c("Name", "Freq")

    #Wydobycie tylko medali zdobytych 2-10 razy
    FirstClassBadges <- FirstClassBadges[FirstClassBadges$Freq>=2 & FirstClassBadges$Freq<=10, ]
    FirstClassBadges <- FirstClassBadges[, "Name"]

    #Wydobycie danych o zdobytych medalach klasy 1
    ValuableBadges <- Badges[, c("Name", "UserId", "Class")]
    ValuableBadges <- ValuableBadges[ValuableBadges$Class==1, ]

    #Znalezienie medali, ktore sa zdefiniowane w FirstClassBadges
    ValuableBadges <- ValuableBadges[ValuableBadges$Name %in% FirstClassBadges, ]

    #Wydobycie danych o uzytkownikach
    Users <- Users[, c("Id", "DisplayName", "Reputation", "Age", "Location")]

    #Polaczenie uzytkownikow z medalami z ValuableBadges i pokazanie unikatowych rekordow
    Users <- merge(x=Users, y=ValuableBadges, by.x="Id", by.y="UserId")
    Users <- unique(Users[, c("Id", "DisplayName", "Reputation", "Age", "Location")])
    return(Users)
}
```

W `dplyr` zadziała następujący kod:

```
df_dplyr_6 <- function(Users, Badges){
    Badges %>%
        dplyr::filter(Class==1) %>%
        dplyr::group_by(Name) %>%
        dplyr::summarise(n=n()) %>%
        dplyr::filter(n>=2 & n<=10) -> FCB
    Badges %>%
        dplyr::select(Name, UserId, Class) %>%
        dplyr::filter(Class==1 & Name %in% FCB$Name) -> VB
    Users %>%
        dplyr::inner_join(VB, by=c("Id" = "UserId")) %>%
        dplyr::distinct(Id, .keep_all=TRUE) %>%
        dplyr::select(Id, DisplayName, Reputation, Age, Location) -> Users
    return(Users)
} #done
```

W `data.table` możemy zastosować takie rozwiązanie:

```
df_table_6 <- function(Users, Badges){
    FCB <- Badges[Class==1] [, .(n=.N), by=.(Name)] [n>=2 & n<=10]
    VB <- Badges[Class==1 & Name %in% FCB[, Name], .(Name, UserId, Class)]

    setkey(Users, Id)
    setkey(VB, UserId)
    Users <- Users[VB][, .(Id, DisplayName, Reputation, Age, Location),
                        by=.(Id, DisplayName, Reputation, Age, Location)]
    return(Users)
} #done
```

Wszystkie wyniki są zgodne ze wzorcem:

```
dplyr::all_equal(df_sql_6(Users, Badges), df_base_6(Users, Badges), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_6(Users, Badges), df_dplyr_6(Users, Badges), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_6(Users, Badges), as.data.frame(df_table_6(dtUsers, dtBadges)),
                 convert=TRUE)

## [1] TRUE
```

A czasy prezentują się następująco:

```
microbenchmark::microbenchmark(
    sql <- df_sql_6(Users, Badges),
    base <- df_base_6(Users, Badges),
    dplyr <- df_dplyr_6(Users, Badges),
    dtable <- df_table_6(dtUsers, dtBadges),
    times = 20
)

## Unit: milliseconds
##                                     expr        min         lq      mean
##           sql <- df_sql_6(Users, Badges) 298.660926 301.735775 308.68712
##         base <- df_base_6(Users, Badges)   7.765909   7.960808  11.95148
##       dplyr <- df_dplyr_6(Users, Badges)  15.172258  16.344727  22.03356
##  dtable <- df_table_6(dtUsers, dtBadges)  10.892864  11.031872  19.04890
##      median         uq       max neval
##  304.894263 308.567543 361.67857    20
##    8.311833   8.824499  38.34588    20
##   16.662847  17.624972 103.11141    20
##   11.362535  15.544363 113.19086    20
```

2.7 Zadanie 7.

Zadanie 7. ma następującą tresć w SQL:

```
df_sql_7 <- function(Posts, Votes){
    sqldf ("SELECT
             Posts.Title,
             VotesByAge2.OldVotes
            FROM Posts
            JOIN (
             SELECT
              PostId,
              MAX(CASE WHEN VoteDate = 'new' THEN Total ELSE 0 END) NewVotes,
              MAX(CASE WHEN VoteDate = 'old' THEN Total ELSE 0 END) OldVotes,
              SUM(Total) AS Votes
             FROM (
              SELECT
               PostId,
               CASE STRFTIME('%Y', CreationDate)
                WHEN '2017' THEN 'new'
                WHEN '2016' THEN 'new'
                ELSE 'old'
                END VoteDate,
               COUNT(*) AS Total
              FROM Votes
              WHERE VoteTypeId=2
              GROUP BY PostId, VoteDate
             ) AS VotesByAge
             GROUP BY VotesByAge.PostId
             HAVING NewVotes=0
            ) AS VotesByAge2 ON VotesByAge2.PostId=Posts.ID
            WHERE Posts.PostTypeId=1
            ORDER BY VotesByAge2.OldVotes DESC
            LIMIT 10") -> final
    return(final)
}
```

Można to rozumieć w ten sposób - "Pokaż tytuły postów typu "1", które mają najwięcej głosów typu "2" sprzed 2016 i nie mają głosów typu "2" w/po 2016".

Żeby ułatwić sobie zadanie, wykorzystałem funkcję pomocniczą "age", która określa, czy data jest dawna czy nie:

```
age <- function(x){
    #funkcja pomocnicza do zadania 7 - zwraca informacje, czy cos jest "stare" czy "nowe".
    #Ze względu na to, że lata są z zakresu 2011-2017, wystarczy porownanie leksykograficzne
    x[x>=2016] <- "new"
    x[x<2016] <- "old"
    return(x)
}
```

W bazowym R rozwiązanie wygląda następująco:

```
df_base_7 <- function(Posts, Votes){
    #Wybranie glosow typu 2
    VotesByAge <- Votes[Votes$VoteTypeId==2, ]
    #Obrobka daty i posegregowanie na stare i nowe
    VotesByAge["CreationDate"] <- age(substr(VotesByAge$CreationDate, 1, 4))
    colnames(VotesByAge)[2] <- "VoteDate"
    #Liczenie liczby starych i nowych glosow dla kazdego posta
    VotesByAge <- aggregate(x=VotesByAge$Id, by=list(VotesByAge$PostId, VotesByAge$VoteDate),
                            FUN=length)
    colnames(VotesByAge) <- c("PostId", "VoteDate", "Total")

    #Utworzenie dwoch tabel - na nowe i na stare glosy
    VotesByAgeNew <- VotesByAge[VotesByAge$VoteDate=="new", ]
    VotesByAgeOld <- VotesByAge[VotesByAge$VoteDate=="old", ]

    #Liczenie max dla kazdego postu wsrod starych i nowych glosow
    VotesByAgeNew <- aggregate(x=VotesByAgeNew$Total, by=VotesByAgeNew["PostId"], FUN=max)
    VotesByAgeOld <- aggregate(x=VotesByAgeOld$Total, by=VotesByAgeOld["PostId"], FUN=max)

    #Polaczenie tablic ale z pokazaniem niepasujacych wartosci, tam bedzie zero zamiast NA
    VotesByAge2 <- merge(x=VotesByAgeNew, y=VotesByAgeOld, by.x="PostId", by.y="PostId",
                         all=TRUE)
    colnames(VotesByAge2) <- c("PostId", "NewVotes", "OldVotes")
    VotesByAge2$NewVotes[is.na(VotesByAge2$NewVotes)] <- 0
    VotesByAge2$OldVotes[is.na(VotesByAge2$OldVotes)] <- 0

    #Segregacja, tak zeby byly posty, ktore mają tylko stare głosy
    VotesByAge2 <- VotesByAge2[VotesByAge2$NewVotes==0, ]

    #Polaczenie z postami typu 1, segregacja, sortowanie
    Posts <- Posts[Posts$PostTypeId==1, ]
    Posts <- Posts[, c("Title", "Id")]
    Posts <- merge(x=Posts, y=VotesByAge2, by.x="Id", by.y="PostId")
    Posts <- Posts[, c("Title", "OldVotes")]
    Posts <- Posts[order(-Posts$OldVotes), ]
    return(head(Posts, n=10))
}
```

W `dplyr` zadziała następujący kod:

```
df_dplyr_7 <- function(Posts, Votes){
    Votes %>%
        dplyr::filter(VoteTypeId==2) %>%
        dplyr::mutate(VoteDate=age(CreationDate)) %>%
        dplyr::group_by(PostId, VoteDate) %>%
        dplyr::summarise(Total=n()) -> VotesByAge
    VotesByAge %>%
        dplyr::filter(VoteDate=="new") %>%
        dplyr::group_by(PostId) %>%
        dplyr::summarise(NewVotes=max(Total)) -> VotesByAgeNew
    VotesByAge %>%
        dplyr::filter(VoteDate=="old") %>%
        dplyr::group_by(PostId) %>%
        dplyr::summarise(OldVotes=max(Total)) -> VotesByAgeOld
    Posts %>%
        dplyr::filter(PostTypeId==1) -> Posts
    VotesByAgeOld %>%
        dplyr::full_join(VotesByAgeNew, by=c("PostId" = "PostId")) %>%
        dplyr::filter(is.na(NewVotes)) %>%
        dplyr::inner_join(Posts, by=c("PostId" = "Id")) %>%
        dplyr::select(Title, OldVotes) %>%
        dplyr::arrange(desc(OldVotes)) %>%
        dplyr::slice(1:10) -> Final
    return(Final)
}
```

W `data.table` możemy zastosować takie rozwiązanie:

```
df_table_7 <- function(Posts, Votes){
    VotesByAge <- Votes[VoteTypeId==2, .(Id, PostId, VoteDate=age(CreationDate))
                        ][, .(Total=.N), by=.(PostId, VoteDate)]
    VotesByAgeNew <- VotesByAge[VoteDate=="new", .(NewVotes=max(Total)), by=.(PostId)]
    VotesByAgeOld <- VotesByAge[VoteDate=="old", .(OldVotes=max(Total)), by=.(PostId)]

    Posts <- Posts[PostTypeId==1]

    setkey(Posts, Id)
    setkey(VotesByAgeNew, PostId)
    setkey(VotesByAgeOld, PostId)
    Final <- VotesByAgeNew[VotesByAgeOld
                           ][is.na(NewVotes)
                             ][Posts
                               ][order(-OldVotes), .(Title, OldVotes)
                                 ][1:10]
    return(Final)
} #done
```

Wszystkie wyniki są zgodne ze wzorcem:

```
dplyr::all_equal(df_sql_7(Posts, Votes), df_base_7(Posts, Votes), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_7(Posts, Votes), df_dplyr_7(Posts, Votes), convert=TRUE)

## [1] TRUE

dplyr::all_equal(df_sql_7(Posts, Votes), as.data.frame(df_table_7(dtPosts, dtVotes)),
                 convert=TRUE)

## [1] TRUE
```

A czasy prezentują się następująco:

```
microbenchmark::microbenchmark(
    sql <- df_sql_7(Posts, Votes),
    base <- df_base_7(Posts, Votes),
    dplyr <- df_dplyr_7(Posts, Votes),
    dtable <- df_table_7(dtPosts, dtVotes),
    times = 20
)

## Unit: milliseconds
##                                    expr       min        lq      mean
##           sql <- df_sql_7(Posts, Votes) 1357.0092 1365.6112 1410.4049
##          base <- df_base_7(Posts, Votes) 2596.6975 2679.5060 2851.7742
##         dplyr <- df_dplyr_7(Posts, Votes) 7906.2730 8309.3233 8756.8167
##  dtable <- df_table_7(dtPosts, dtVotes)  259.9777  286.1569  311.8903
##     median        uq        max neval
##  1379.1324 1394.3237  1636.4661    20
##  2760.0247 2876.4885  3608.2957    20
##  8584.1678 8897.3338 11653.3980    20
##   299.1606  306.6079   519.6511    20
```

3 Omówienie

Patrząc na czasy wykonania poszczególnych kwerend, jak i mając na uwadze doświadczenie związane z pisaniem kodu w funkcjach bazowych `R`, `dplyr` i data.table, można wydać ocenę na temat tych sposobów. Z reguły `data.table` radzi sobie najszybciej, trochę za nim jest `dplyr`. Mimo że kod w jest dłuższy niż w `data.table`, to jednak moje doświadczenia z nim są lepsze - sprawia wrażenie duzo bardziej przejrzystego. Funkcje bazowe i `sqldf` zazwyczaj są wolniejsze o rząd wielkości, i o ile `sqldf` jest idealny dla osób zaznajomionych z SQL, to funkcje bazowe `R` w moim osobistym rankingu plasują się na ostatnim miejscu.