R Training

Lesson 3

Stefanie Molin

May 15, 2017

I. Data Manipulation

Now, we are going to build on the knowledge from the last session and learn how to use the `dplyr` package to manipulate dataframes quickly and efficiently in conjunction with the packages we covered in the prior lesson.

We are going to continue with the below dataframes as we defined in the last lesson.

```r
# define dataframes
ids <- data.frame(name = c("Alice", "Bob", "Carly", "Dylan"), id = 101:104,
                  stringsAsFactors = FALSE)
ages <- data.frame(age = c(24, 26, 28), id = c(101:102, 105))

# new employees
newEmployees <- data.frame(name = c("Eva", "Frank"), id = 106:107,
                           stringsAsFactors = FALSE)

# revise the ids dataframe to include all employees
ids <- rbind(ids, newEmployees)

# view each dataframe
ids
```

```
##    name  id
## 1 Alice 101
## 2   Bob 102
## 3 Carly 103
## 4 Dylan 104
## 5   Eva 106
## 6 Frank 107
```

```r
ages
```

```
##   age  id
## 1  24 101
## 2  26 102
## 3  28 105
```

```r
newEmployees
```

```
##    name  id
## 1   Eva 106
## 2 Frank 107
```

A. Type conversion and data cleaning

Before we get into the `dplyr` package, let's learn how to clean up our data for analysis in R. We will use [redacted] stats for the last 7 days, handle any `NULL` values that come from our query, and change columns to more appropriate data types. Note that it won't always be necessary to clean this up depending on what you are looking to do in your analysis since you can have summary functions like `sum()`, `mean()`, etc. ignore `NA`'s (the value used when R reads in `NULL`) and you may not need to use the `day` string as a date. We will learn how to do the cleaning using base R; it is also possible in `dplyr`, but I will leave that for you to try on your own.

```
# get date for 7 days ago
startDate <- Sys.Date() - 7

# query for last 30 days of client stats for [redacted]
query <- "SELECT
            *
          FROM
            [redacted]
          WHERE
            day >= '%s'
            AND client_id = 8050"

# query Vertica for data and store in dataframe [redacted]
[redacted] <- QueryVertica(username, sprintf(query, startDate), password)

# look at a summary of the data to check for NA's and wrong data types
summary[redacted]s)
```

```
##      day               client_id      displays        clicks
##  Length:69          Min.   :8050   Min.   :[redacted]   Min.   [redacted]
##  Class :character   1st Qu.:8050   1st Qu.:[redacted]   1st Qu.[redacted]
##  Mode  :character   Median :8050   Median :[redacted]   Median [redacted]
##                     Mean   :8050   Mean   :[redacted]   Mean   [redacted]
##                     3rd Qu.:8050   3rd Qu.:[redacted]   3rd Qu.[redacted]
##                     Max.   :8050   Max.   :[redacted]   Max.   [redacted]
##
##     revenue              tac          post_click_conversions
##  Min.   [redacted]   Min.   :[redacted]   Min.   :[redacted]
##  1st Qu.[redacted]   1st Qu.:[redacted]   1st Qu.:[redacted]
##  Median [redacted]   Median :[redacted]   Median :[redacted]
##  Mean   [redacted]   Mean   :[redacted]   Mean   :[redacted]
##  3rd Qu.[redacted]   3rd Qu.:[redacted]   3rd Qu.:[redacted]
##  Max.   [redacted]   Max.   :[redacted]   Max.   :[redacted]
##                                          NA's   :37
##  post_click_sales post_view_conversions post_view_sales
##  Min.   [redacted]   Min.   [redacted]   Min.   :[redacted]
##  1st Qu.[redacted]   1st Qu.[redacted]   1st Qu.:[redacted]
##  Median [redacted]   Median [redacted]   Median :[redacted]
##  Mean   [redacted]   Mean   [redacted]   Mean   :[redacted]
##  3rd Qu.[redacted]   3rd Qu.[redacted]   3rd Qu.:[redacted]
##  Max.   [redacted]   Max.   [redacted]   Max.   :[redacted]
##  NA's   [redacted]   NA's   :39          NA's   :39
##  post_click_non_deduplicated_conversions post_click_non_deduplicated_sales
##  Min.   :[redacted]                       Min.   [redacted]
##  1st Qu.:[redacted]                       1st Qu.[redacted]
```

```
##   Median   [redacted]                   Median   [redacted]
##   Mean     [redacted]                   Mean     [redacted]
##   3rd Qu.  [redacted]                   3rd Qu.  [redacted]
##   Max.     [redacted]                   Max.     [redacted]
##   NA's    :37                           NA's    :37
##   cumulated_p_ctr cumulated_p_cr zone_currency_id client_country_code
##   Min.    [redacted] Min.    [redacted] Min.    [redacted] Length:69
##   1st Qu.:[redacted] 1st Qu.[redacted] 1st Qu.[redacted] Class :character
##   Median  [redacted] Median  [redacted] Median  [redacted] Mode  :character
##   Mean    [redacted] Mean    [redacted] Mean    [redacted]
##   3rd Qu.:[redacted] 3rd Qu.[redacted] 3rd Qu.[redacted]
##   Max.    [redacted] Max.    [redacted] Max.    [redacted]
##
##   marketplace_revenue_raw marketplace_revenue
##   Min.    [redacted]      Min.    [redacted]
##   1st Qu. [redacted]      1st Qu. [redacted]
##   Median  [redacted]      Median  [redacted]
##   Mean    [redacted]      Mean    [redacted]
##   3rd Qu. [redacted]      3rd Qu. [redacted]
##   Max.    [redacted]      Max.    [redacted]
##
```

1. Type Conversion

Looking at the summary we can see that `day` was read in as a `character` when it should be a `Date` and `client_id` is not stored in the most optimal way. It is stored as a numeric, however, `client_id` is a categorical value so we should store it as a `factor`. There are a few other columns that make sense to be converted to factors, but we won't show those here. Type conversions can be accomplished by functions of the form `as.*()` where `*` is the class you want to convert to.

2. Data Cleaning (handling `NA`'s)

The second thing we can glean from the summary is which columns have `NA`'s. Depending on the analysis you are doing you may want to remove rows with `NA` values, replace them with new values such as 0, or impute a value. Here we will replace all `NA` values in `post_click_conversions` and `post_click_sales` with 0 and remove all rows that have `NA` in the columns `post_view_conversions` or `post_view_sales`.

```r
# change the day column to a date
[redacted]$day <- as.Date[redacted]$day)

# change the client_id column to a factor
[redacted]$client_id <- as.factor[redacted]$client_id)

# replace NAs with 0 in select columns
[redacted][is.na([redacted]$post_click_conversions), "post_click_conversions"] <- 0
      [is.na([redacted]$post_click_sales), "post_click_sales"] <- 0

# remove select rows
[redacted] <- [redacted][!(is.na([redacted]$post_view_conversions) |
                     is.na([redacted]$post_view_sales)),]

# look at summary of modified columns
summary([redacted][, c("day", "client_id", "post_click_conversions",
                   "post_click_sales", "post_view_conversions", "post_view_sales")])
```

```
##       day               client_id post_click_conversions post_click_sales
##  Min.   :2017-05-08   8050:30   Min.   [redacted]     Min.   [redacted]
##  1st Qu.:2017-05-09             1st Qu.[redacted]     1st Qu.[redacted]
##  Median :2017-05-11             Median [redacted]     Median [redacted]
##  Mean   :2017-05-11             Mean   [redacted]     Mean   [redacted]
##  3rd Qu.:2017-05-13             3rd Qu.[redacted]     3rd Qu.[redacted]
##  Max.   :2017-05-14             Max.   [redacted]     Max.   [redacted]
##  post_view_conversions post_view_sales
##  Min.   [redacted] Min.   [redacted]
##  1st Qu.[redacted] 1st Qu.[redacted]
##  Median [redacted] Median [redacted]
##  Mean   [redacted] Mean   [redacted]
##  3rd Qu.[redacted] 3rd Qu.[redacted]
##  Max.   [redacted] Max.   [redacted]
```

B. `dplyr`

The `dplyr` package is extremely useful when working with dataframes. You can chain together several "verbs" to step-by-step transform your dataframe without having to write a complicated query. You can also take in granular data (either from Vertica or read in from a file) and use `dplyr` to get the information you wanted if Vertica keeps timing out or you aren't sure how to write a query to do what you need. We are going to cover the pipe operator (`%>%`), the 5 verbs and `group_by()`, joins, and set operations.

```
library(dplyr)
```

1. The pipe operator

The pipe operator (`%>%`) uses the object before the pipe as the first argument to the function after the pipe. It is read as "and then". You can use **ctrl + shift + m** to add the pipe operator. Note that you can use this operator for any function; it doesn't have to be a `dplyr` function as long as you load the `dplyr` package. This becomes extremely helpful for readability when you have multiple nested function calls.

```
# find max age from the ages dataframe defined earlier
max(ages$age)
```

```
## [1] 28
```

```
# now using the pipe operator
ages$age %>% max()
```

```
## [1] 28
```

2. The 5 verbs + `group_by()`

There are many functions available in the `dplyr` package, however, the 5 verbs and `group_by()` are the ones I find applicable in most situations. Before we get into what each one does, let's pull in a dataset to work with in the examples. We are going to look at performance stats for the client [redacted]. Note that some of the below can be done in base R, but `dplyr` will make your code easier to read.

```
# get date for 30 days ago
startDate <- Sys.Date() - 30

# query for last 30 days of client stats for [redacted]
query <- "SELECT
            *
          FROM
            [redacted]
          WHERE
            day >= '%s'
            AND client_id = 4624"

# query Vertica for data and store in dataframe macys
[redacted] <- QueryVertica(username, sprintf(query, startDate), password)

# see how many rows we pulled in
nrow([redacted])
```

```
## [1] 312
```

```
# see what columns we pulled in
colnames([redacted])
```

```
##  [1] [redacted]
##  [2]
##  [3]
##  [4]
##  [5]
##  [6]
##  [7]
##  [8]
##  [9]
## [10]
## [11]
## [12]
## [13]
## [14]
## [15]
## [16]
## [17]
## [18]
```

Looks like we got more data than we bargained for! Let's use `dplyr` to clean up the dataframe.

i. `select()`

The `macys` dataframe contains more columns than we need. The `select()` function in `dplyr` works just like a `SELECT` statement in a query. We can use `select()` to only keep the columns relevant to our analysis. To select columns, provide their name or their index; you can drop columns by negating your selection (adding "`-`" in front). We can also do the below in base R with `df[, c("col1", "col2")]`

```
# select columns by name and show first 3 (rename conversions and sales)
[redacted] %>%
  select(day, displays, clicks, revenue, pc_conv = post_click_conversions,
         pc_sales = post_click_sales) %>% head(3)
```

```
##          day displays clicks  revenue pc_conv   pc_sales
## 1 2017-05-01 [redacted]
## 2 2017-05-01
## 3 2017-05-01
```

Note that the below are equivalent ways of performing the same selection (but without the column renaming we performed above).

```
# select columns by index
[redacted] %>%
  select(1:5, 7:8)

# drop columns by index
[redacted] %>%
  select(-c(6, 9:18))
```

`dplyr` also contains several functions that can be used in conjunction with `select()` to pick the columns that match certain criteria; among these are `starts_with()`, `ends_with()`, `contains()`, `matches()`, `one_of()` which you can read up on later.

ii. `filter()`

Great, now we have the columns we needed, but turns out we only needed the last 25 days. Now what? We will use the `filter()` function (like the `WHERE` clause) to filter the data to what we need continuing from the code we started in the `select()` section so you get an idea of how this looks in practice. In base R you can filter using: `df[condition,]` (i.e. `df[as.Date(df$day) <= "2017-05-15",]`).

```
# select columns and filter for last 25 days
x1 <- [redacted] %>%
  select(day, displays, clicks, revenue, pc_conv = post_click_conversions,
         pc_sales = post_click_sales) %>%
  filter(as.Date(day) >= Sys.Date() - 25)

# look at the dates we got back
unique(x1$day)
```

```
##  [1] "2017-05-01" "2017-05-11" "2017-04-28" "2017-04-26" "2017-05-07"
##  [6] "2017-05-06" "2017-04-21" "2017-05-09" "2017-05-12" "2017-05-02"
## [11] "2017-05-13" "2017-04-25" "2017-04-30" "2017-05-05" "2017-05-14"
## [16] "2017-04-20" "2017-05-03" "2017-04-22" "2017-04-29" "2017-04-23"
## [21] "2017-05-04" "2017-04-27" "2017-04-24" "2017-05-08" "2017-05-10"
```

iii. `arrange()`

Looks like we got the proper dates and columns now, but they are all out of order. We can't tell what we are looking at! Let's use `arrange()` to sort the data by the `day` column; this is equivalent to the `ORDER BY` clause. Note this can be done in base R with `df[order(df$sort_column),]` and you can use `rev()` on `order()` to do decreasing order (i.e. `df[rev(order(df$sort_column),])`.

```
# select columns, filter to last 25 days, and sort by day increasing
x2 <- [redacted] %>%
  select(day, displays, clicks, revenue, pc_conv = post_click_conversions,
         pc_sales = post_click_sales) %>%
  filter(as.Date(day) >= Sys.Date() - 25) %>%
  arrange(day)

# check what order the dates are in now
unique(x2$day)
```

```
##  [1] "2017-04-20" "2017-04-21" "2017-04-22" "2017-04-23" "2017-04-24"
##  [6] "2017-04-25" "2017-04-26" "2017-04-27" "2017-04-28" "2017-04-29"
## [11] "2017-04-30" "2017-05-01" "2017-05-02" "2017-05-03" "2017-05-04"
## [16] "2017-05-05" "2017-05-06" "2017-05-07" "2017-05-08" "2017-05-09"
## [21] "2017-05-10" "2017-05-11" "2017-05-12" "2017-05-13" "2017-05-14"
```

Note you can sort descending by using `arrange(desc(day))`.

iv. `mutate()`

Now that we have data that makes more sense, let's introduce `mutate()` which allows us to add new columns like CTR and CPC. Note in base R you can add columns as `df$new_col <- new_values`.

```
# select columns, filter to last 25 days, add CTR and CPC, sort by day increasing
[redacted] %>%
  select(day, displays, clicks, revenue, pc_conv = post_click_conversions,
         pc_sales = post_click_sales) %>%
  filter(as.Date(day) >= Sys.Date() - 25) %>%
  mutate(ctr = clicks/displays, cpc = revenue/clicks) %>%
  arrange(day) %>%
  head()
```

```
##          day displays clicks  revenue pc_conv  pc_sales         ctr
## 1 2017-04-20 [redacted]
## 2 2017-04-20
## 3 2017-04-20
## 4 2017-04-20
## 5 2017-04-20
## 6 2017-04-20
##          cpc
## 1 [redacted]
## 2
## 3
## 4
## 5
## 6
```

v. `summarize()`

All we have to do now is `summarize()` (aggregate) everything over the time period.

```
# select columns, filter to last 25 days, and sort by day increasing
# summarize the result (we don't need the arrange or mutate here)
[redacted] %>%
  select(day, displays, clicks, revenue, pc_conv = post_click_conversions,
         pc_sales = post_click_sales) %>%
  filter(as.Date(day) >= Sys.Date() - 25) %>%
  summarize(days = n_distinct(day), rows = n(), total_clicks = sum(clicks, na.rm = TRUE),
            total_imps = sum(displays, na.rm = TRUE), spend = sum(revenue, na.rm = TRUE),
            conv = sum(pc_conv, na.rm = TRUE), sales = sum(pc_sales, na.rm = TRUE),
            ctr = total_clicks/total_imps)
```

```
##   days rows total_clicks total_imps    spend   conv    sales         ctr
## 1   25  257 [redacted]
```

Notice how we were able to use `total_clicks` and `total_imps` in an equation in `summarize()` after we defined them; you can also do this with `mutate()`. `dplyr` contains the `n_distinct()` and `n()` functions among others that will be helpful when aggregating. We also included `na.rm = TRUE` in our calls to `sum()`, if you don't R will try to add `NA`'s which are basically NULL's to your data and as a result you won't get anything useful back; this tells R to ignore those in the `sum()` calculation.

vi. `group_by()`

That is helpful, but it wasn't exactly what we were looking for. We want all these metrics by day, but the data isn't aggregated like that in Vertica and `summarize()` can't handle it on its own. We need to `GROUP BY` just like in a query. We can use `dplyr`'s aptly named `group_by()` to do this for us and complete our analysis.

```
# select columns, filter to last 25 days, group by day to summarize
# add ctr and cpc and sort by day
[redacted]_pivot <- [redacted] %>%
  select(day, displays, clicks, revenue, pc_conv = post_click_conversions,
         pc_sales = post_click_sales) %>%
  filter(as.Date(day) >= Sys.Date() - 25) %>%
  group_by(day) %>%
  summarize(total_clicks = sum(clicks, na.rm = TRUE),
            total_imps = sum(displays, na.rm = TRUE), spend = sum(revenue, na.rm = TRUE),
            conv = sum(pc_conv, na.rm = TRUE)) %>%
  mutate(ctr = total_clicks/total_imps, cpc = spend/total_clicks) %>%
  arrange(day)

# view first few rows
head([redacted]_pivot, 3)
```

```
## # A tibble: 3 × 7
##          day total_clicks total_imps    spend  conv        ctr       cpc
##        <chr>        <dbl>      <dbl>    <dbl> <dbl>      <dbl>     <dbl>
## 1 2017-04-20 [redacted]
## 2 2017-04-21 [redacted]
## 3 2017-04-22 [redacted]
```

*Note that order matters with `group_by()` and `summarize()`. If you want to summarize by groups you **must** first `group_by()` and then `summarize()`.*

It is important to note that after using this `dplyr` function, your dataframe gets turned into a tibble. Don't worry though, you can still use all your dataframe code on these objects; they are just enhanced versions of dataframes.

```
class([redacted]_pivot)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

3. Join operations

`dplyr` also provides various functions for joining dataframes with clearer syntax than base R's `merge()`. There are 2 main categories: mutating joins and filtering joins. As with base R, `dplyr`'s version will join on all common columns, but you can specify them in the `by` argument. `dplyr` will tell you which columns it joined on in the output.

i. Mutating joins

Mutating joins add columns to the resulting dataframe. For example, if you are left joining table `y` to table `x`, you are adding the columns of `y` that `x` didn't have and joining on the columns in common.

- `left_join()`
- `right_join()`
- `inner_join()`
- `full_join()`

Let's redo the joins we did using `merge()` from base R in the last lesson.

```
# inner join
inner_join(ids, ages)
```

```
## Joining, by = "id"
```

```
##    name  id age
## 1 Alice 101  24
## 2   Bob 102  26
```

```
# left outer join
left_join(ids, ages)
```

```
## Joining, by = "id"
```

```
##    name  id age
## 1 Alice 101  24
## 2   Bob 102  26
## 3 Carly 103  NA
## 4 Dylan 104  NA
## 5   Eva 106  NA
## 6 Frank 107  NA
```

```
# right outer join
right_join(ids, ages)
```

```
## Joining, by = "id"
```

```
##    name  id age
## 1 Alice 101  24
## 2   Bob 102  26
## 3  <NA> 105  28
```

```r
# full outer join
full_join(ids, ages)
```

```
## Joining, by = "id"
```

```
##    name  id age
## 1 Alice 101  24
## 2   Bob 102  26
## 3 Carly 103  NA
## 4 Dylan 104  NA
## 5   Eva 106  NA
## 6 Frank 107  NA
## 7  <NA> 105  28
```

ii. Filtering joins

Filtering joins return the result of a join without adding columns from the second dataframe. Here you are joining `x` and `y`, but you don't want any columns from `y`, you are just using it to **filter** `x`.

- `semi_join()`: filters results of primary table to those with matches in the secondary table (preview of rows kept from left table after inner join)
- `anti_join()`: filters results of primary table to those which do not have a match in the secondary table (opposite of `semi-join()`). You can also think of this as which columns will I lose if I do an inner join.

Let's take a look at how these work with the same data from the mutating joins section.

```r
# semi-join (which rows in ids have matches in ages?)
semi_join(ids, ages)
```

```
## Joining, by = "id"
```

```
##    name  id
## 1 Alice 101
## 2   Bob 102
```

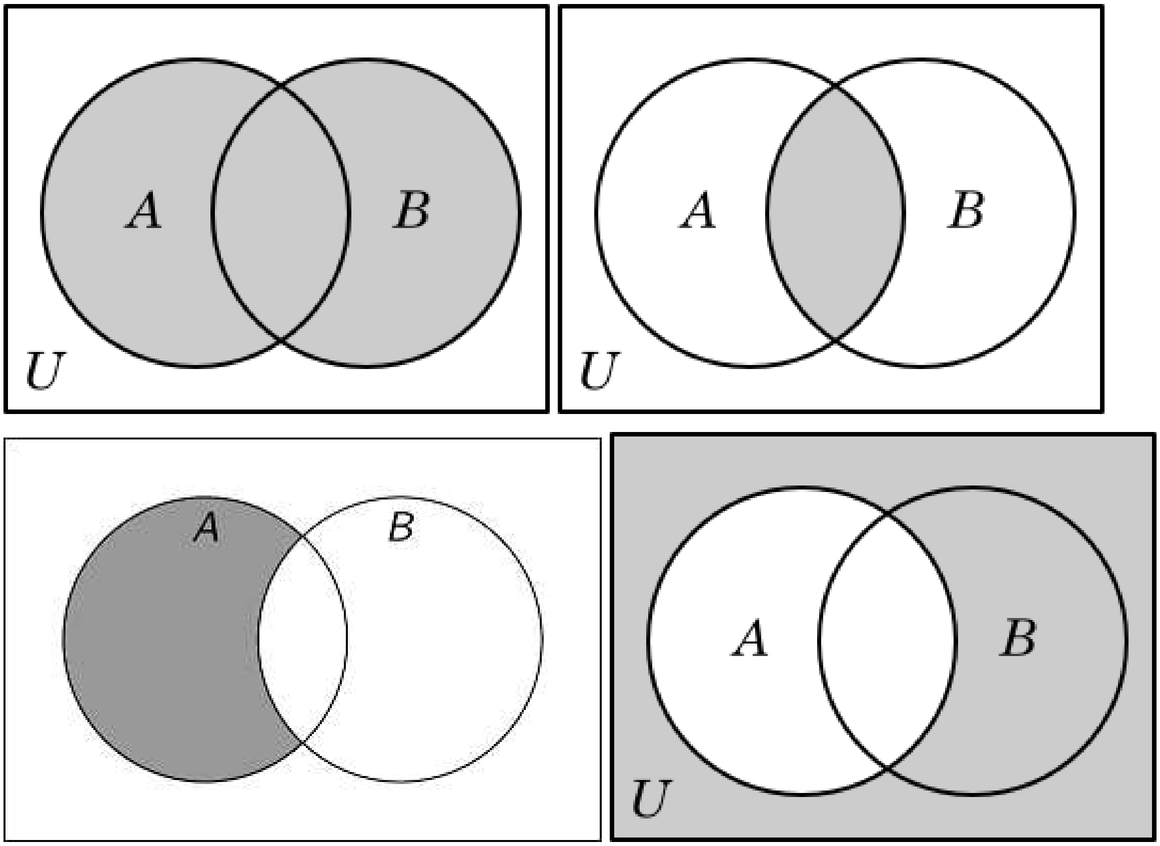
```r
# anti-join (which rows in ids don't have matches in ages?)
anti_join(ids, ages)
```

```
## Joining, by = "id"
```

```
##    name  id
## 1 Carly 103
## 2 Dylan 104
## 3   Eva 106
## 4 Frank 107
```

4. Set operations

A set is a collection of distinct objects. There are four set operations best explained with Venn Diagrams: union, intersection, difference, and complement. *(Left to right, top to bottom)*



R doesn't cover complement because calculating the complement of a dataframe requires knowing the entire sample space which isn't feasible in practice. Set operations from `dpylr` will take in 2 dataframes (A and B) and return the result of the set operation, removing duplicates. You can also provide other data types such as vectors or dataframe columns to these functions; however, whatever arguments you provide, both need to have the same columns.

- `union()`: returns all rows that appear in either A or B, removing duplicates.
- `intersect()`: returns all rows in common.
- `setdiff()`: returns all rows in A that aren't in B.
- `setequal()`: tests if A and B contain the exact same data (in any order) and returns a boolean.

Let's continue with the dataframes we defined earlier.

```r
# add duplicate data to the second dataframe
newEmployees2 <- rbind(newEmployees,
                       data.frame(name = c("Alice", "Bob"), id = c(110L, 102L),
                                  stringsAsFactors = FALSE))
```

```r
# union of employee tables
union(ids, newEmployees2)
```

```
##    name  id
## 1 Alice 101
## 2 Alice 110
## 3   Bob 102
## 4 Carly 103
## 5 Dylan 104
## 6   Eva 106
## 7 Frank 107
```

Notice that we lose the "Bob", "Eva", and "Frank" rows since they were entirely duplicates, but "Alice" stays since the rows had different IDs.

```r
# intersection of employee tables
intersect(ids, newEmployees2)
```

```
##    name  id
## 1   Eva 106
## 2 Frank 107
## 3   Bob 102
```

This time we only see the duplicated rows. (You can use `intersect()` to see what will be deduplicated in a call to `union()`).

```r
# difference of employee tables
setdiff(ids, newEmployees2)
```

```
##    name  id
## 1 Alice 101
## 2 Carly 103
## 3 Dylan 104
```

The result contains all the rows from A that we would lose if we ran `intersect()`.

```r
# check if ids and newEmployees2 are the same
setequal(ids, newEmployees2)
```

```
## FALSE: Different number of rows
```

```r
# check if these are equal (order doesn't matter)
setequal(c("Alice", "Bob"), c("Bob", "Alice"))
```

```
## [1] TRUE
```

II. Exercises

Let's do some practice problems to challenge your understanding.

1. Query for two dataframes: (1) all AS in your office along with their employee IDs and (2) the accounts in the US and the AS employee ID associated with them. Use `dplyr` filtering joins to (a) preview the results that will be lost from dataframe (1) if you do an inner join on both tables and (b) preview the results that will remain in dataframe (1) if you do an inner join. (c) inner join dataframes (1) and (2) and confirm your results.

```r
library(dplyr)
```

```r
# packages have been loaded along with QueryVertica()
# username/password have been predefined

# query for all AS in NY office
ny_as_query <- "
SELECT
    employee_id
    , full_name
FROM
  [redacted]
WHERE
    cost_center_country = 'NY'
    AND job_name = [redacted]
GROUP BY
    employee_id
    , full_name
"

nyAS <- QueryVertica(username, ny_as_query, password)

# query for US accounts
us_accounts_query <- "
SELECT
    account_strategist_employee_id
    , merchant_name
FROM
  [redacted]
WHERE
    ranking = 'TIER 1'
    AND client_country_code = 'US'
GROUP BY
    account_strategist_employee_id
    , merchant_name"

usAccounts <- QueryVertica(username, us_accounts_query, password)
```

```r
# look at number of rows in each table
nrow(nyAS)
```

```
## [1] 17
```

```r
nrow(usAccounts)
```

```
## [1] 888
```

```r
# preview what will be lost from nyAS in inner join above (these AS don't have accounts)
asWithoutAccounts <- anti_join(nyAS, usAccounts,
                               by = c("employee_id" = "account_strategist_employee_id"))
nrow(asWithoutAccounts)
```

```
## [1] 2
```

```r
head(asWithoutAccounts)
```

```
##   employee_id        full_name
## 1 [redacted]
## 2
```

```r
# preview what will be kept from nyAS in inner join above (all these AS have accounts)
asWithAccounts <- semi_join(nyAS, usAccounts,
                            by = c("employee_id" = "account_strategist_employee_id"))
nrow(asWithAccounts)
```

```
## [1] 15
```

```r
head(asWithAccounts)
```

```
##   employee_id           full_name
## 1 [redacted]
## 2
## 3
## 4
## 5
## 6
```

```r
# inner join the two tables
nyAccounts <- inner_join(nyAS, usAccounts,
                         by = c("employee_id" = "account_strategist_employee_id"))
nrow(nyAccounts)
```

```
## [1] 148
```

```r
head(nyAccounts)
```

```
##   employee_id       full_name          merchant_name
## 1 [redacted]
## 2
## 3
## 4
## 5
## 6
```

```r
# how many employees are left after the join
length(unique(nyAccounts$employee_id))
```

```
## [1] 15
```

2. Pull in the first names of every employee *currently* working at [redacted] (cost centers US, NY, IL, SF), and, in a second dataframe, the first names of every employee that currently works at [redacted], but *not* in the US. Be sure to write a dynamic query, so that you only have to write one query! Use an `rbind()` to get the complete list of employee first names in a separate dataframe.

```
# dynamic query for first names
name_query <- "
SELECT
    first_name
FROM
    [redacted]
WHERE
    cost_center_country %s
  AND job_status = 'LIVE'
"

usNames <- QueryVertica(username,
                        sprintf(name_query, " IN ('US', 'NY', 'IL', 'SF')"),
                        password)
notUSNames <- QueryVertica(username,
                           sprintf(name_query,  "NOT  IN ('US', 'NY', 'IL', 'SF')"),
                           password)

# all employee first names
allNames <- rbind(usNames, notUSNames)
head(allNames)
```

```
##        first_name
## 1[redacted]
## 2
## 3
## 4
## 5
## 6
```

```
tail(allNames)
```

```
##       first_name
## 2893[redacted]
## 2894
## 2895
## 2896
## 2897
## 2898
```

3. Using the two dataframes you queried for in `(2)` and set operations, (a) find all first names that are either in [redacted] (cost centers US, NY, IL, SF) **or** any other office, but **not** in both; (b) count how many people have each name, and sort it from most common to least common and by name alphabetically. Then, (c) flag and return the top 10 most common along with their counts, and (d) find the first names of the employees that are the only one in the company with that name, and (e) compare this result to the result from (a). (*Hint use* ***`setequal()`***).

```
# names either in [redacted] OR elsewhere (a)
noOverlap <- usNames %>%
  union(notUSNames) %>%
  setdiff(intersect(usNames, notUSNames))
head(noOverlap)
```

```
##       first_name
## 1 [redacted]
## 2 [redacted]
## 3 [redacted]
## 4 [redacted]
## 5 [redacted]
## 6 [redacted]
```

```
# count names and sort (b)
nameCount <- allNames %>%
  select(first_name) %>%
  group_by(first_name) %>%
  summarize(count = n()) %>%
  arrange(desc(count), first_name)
```

```
# top 10 most common (c)
head(nameCount, 10) %>% mutate(top_10 = TRUE)
```

```
## # A tibble: 10 × 3
##    first_name count top_10
##         <chr> <int>  <lgl>
## 1[redacted] TRUE
## 2[redacted] TRUE
## 3[redacted] TRUE
## 4[redacted] TRUE
## 5[redacted] TRUE
## 6[redacted] TRUE
## 7[redacted] TRUE
## 8[redacted] TRUE
## 9[redacted] TRUE
## 1[redacted] TRUE
```

```
# people with unique first names (d)
uniqueNames <- nameCount %>%
  filter(count == 1) %>%
  select(first_name)

# compare result from (d) to (a)
setequal(noOverlap, uniqueNames)
```

```
## FALSE: Different number of rows
```