# Lesson 5: Blending data from multiple files and sources

*Randall Julian, Adam Zabell*

## Joining Relational Data

The database example for this class has three different data.frames (we use data.frame interchangably with the word tibble, even though we usually mean tibble) : one for batch-level information (calibration $R^2$, instrument name); one for sample-level information (sample type, calculated concentration); and one for peak-level information (quant peak area, modification flag). Accessing the relationships across these three sources – reporting the quant and qual peak area of only the qc samples, for example – requires the tools of relational data. In the tidyverse, these tools are part of the **dplyr** package and involve three 'families of verbs' called *mutating joins*, *filtering joins*, and *set operations*, which in turn expect a unique key in order to correctly correlate the data. To begin, read in the batch, sample, and peak data from the month of January. For simplicity, we will reduce size of our working examples to only those rows of data associated with one of two batches.

```
jan.b <- read_excel(here::here("class_data","2017-01-06.xlsx"))
jan.s <- read_csv(here::here("class_data","2017-01-06.csv"))
jan.p <- read_tsv(here::here("class_data","2017-01-06.txt"))
byBatch <- jan.b[jan.b$batchName=="b802253" | jan.b$batchName=="b252474",]
bySample <- jan.s[jan.s$batchName=="b802253" | jan.s$batchName=="b252474",]
byPeak <- jan.p[jan.p$batchName=="b802253" | jan.p$batchName=="b252474",]
```

Let's first talk about two simple commands in R which bind data, and the pitfalls of relying too heavily upon them.

## Blending Data

### Simple *rbind* and *cbind*

Sometimes, you need data stored across more than one file. For example, managing the QC deviations across twelve separate months of reports. Rather than hold multiple printouts side-by-side to monitor a trend, you can read each file and knit them together either by row, or by column. Working with tidy data means a 'wide' data.frame where each column is a specific variable, and each row is a specific measurment. If you know that your data sources have the same format, you can safely combine them with an `rbind` to append the second source of data at the end of the first.

```
january <- read_csv(here::here("class_data","2017-01-06.csv"))
as.data.frame(january[187195:187200,])
february <- read_csv(here::here("class_data","2017-02-06.csv"))
as.data.frame(february[1:5,])
twoMonths <- rbind(january,february)
```

Notice the continuation from the last rows of `january` to the first rows of `february` and that the number of rows in the combined data.frame `twoMonths` is the sum of the first two months of sample-level data.

```
twoMonths[187195:187204,]
c(nrow(january), nrow(february), nrow(twoMonths))
```

Because of how R treats columns, as long as the two files have the same number of columns and the same column names, the `rbind` command will correctly associate the data using the column order from the first variable. And if they aren't the same, you get an error that tells you what is wrong.

```
awkwardJanuary <- january[,c(10,1:9)]
missingJanuary <- january[,c(1:9)]
misnamedJanuary <- january
names(misnamedJanuary)[1] <- "bName"
```

> Exercise 1: Attempting an unsuccessful rbind
> Partially complete commands are commented out in the following code chunk. Perform an rbind between February and each of these three alternative January data.frames. Do any of them work? What does the data look like? What error messages do you get?

```
#rbind( , february)
```

There is an equivalent command called `cbind` which will append columns to a data.frame, but it is far more risky. The expectation within R that each row of information is correctly organized into a set of columns means this command will not check to make sure the order of values are correct between the two terms. Working to retain the correct association is the purpose of using keys as unique identifiers (see below).

```
incomplete_data <- tibble(sampleName="123456",
                          compoundName=c("morphine","hydromorphone","codeine","hydrocodone"),
                          concentration=c(34,35,44,45))
additional_columns <- tibble(expectedConcentration=c(20,30,40,40),
                             sampleType="standard")
erronious_columns <- tibble(expectedConcentration=c(40,30,20,40),
                             sampleType="standard")
desired_cbind <- cbind(incomplete_data,additional_columns)
undesired_cbind <- cbind(incomplete_data,erronious_columns)
```

There are also dplyr functions with equivalent function: `bind_rows()` and `bind_cols()`.

There are better ways of adding per-column data. Creating a new named vector directly into the existing data.frame is straightforward and shown below, but a true merge of two distinct data.frames needs more explanation.

```
incomplete_data$batchName <- "batch01"
incomplete_data
```

### Primary and foreign keys

A key is the variable in a data.frame – or combination of variables in a data.frame – that uniquely defines every row. In our data, `batchName` is present in each data.frame but always insufficient to define a specific row. In fact, no single column in our data operates as a key. We can build a key by combinging two (or three) columns.

```
byBatch$keyB <- paste(byBatch$batchName, byBatch$compoundName, sep=":")
bySample$keyS <- paste(bySample$sampleName, bySample$compoundName, sep=":")
byPeak$keyP <- paste(byPeak$sampleName, byPeak$compoundName, byPeak$chromatogramName, sep=":")
```

Doing this creates a **primary key**, which is the unique identifier for that data.frame. A **foreign key** by contrast would uniqely identify an item in another table. The *left_join* command (described soon) does it's job correctly but because the compound names for the internal standards are not identical to the analyte compound names, this command incompletely adds the `byBatch` information to `byPeak`. The second command completes the population of `byPeak$keyB` as a foregin key, but only because the row order in `byPeak` follows a very specific format – exactly the problem we were worried about when using *cbind*! There are safer ways of populating this variable which take advantage of set operations (described next).

```
byPeak <- left_join(byPeak, byBatch)
byPeak$keyB[is.na(byPeak$keyB)] <- byPeak$keyB[!is.na(byPeak$keyB)] # dangerous!
```

> Exercise 2: Join the batch and sample data using only the batch-specific key
> Partially complete commands are commented out in the following code chunk. Since `keyB` is already built for one data.frame, creating this variable in `bySample` is the next step. How would you specify that only this variable which should be used for the join? Notice what that does for all of the variables in the joined data.frame.

```
byBatch$keyB
# bySample$keyB <- paste( , ,sep=":")
# exerciseTwo <- left_join( , , )
```

### Set operations *union*, *intersect*, and *setdiff*

These three commands will return a vector which is the unduplicated combination of the two input vectors. `union(A,B)` includes all the values found in both A and B. `intersect(A,B)` returns only those values found in both A and B. `setdiff(A,B)` is order dependent, and returns the values of the first vector which are not also in the second vector.

```
A <- rep(seq(1, 10), 2)
B <- seq(2, 20, 2)
union(A, B)
intersect(A, B)
setdiff(A, B)
setdiff(B, A)
```

These commands are good for checking matches between two vectors, and we can use them to rebuild the `byPeak$keyB` foreign key without the risk of incorrect naming. First, let's show a couple of different ways of resetting `byPeak`. The first is familiar to people using **R** without the tidyverse, and we increase our sophistication in subsequent examples. Any of them work, but the last example is the most tidyverse way of doing things and makes it more readable to your colleagues who are also working in a tidy way.

```
byPeak <- jan.p[jan.p$batchName=="b802253" | jan.p$batchName=="b252474",] # reset
byPeak$keyP <- paste(byPeak$sampleName, byPeak$compoundName, byPeak$chromatogramName, sep=":")

byPeak <- jan.p %>%
  filter(batchName %in% c("b802253","b252474")) # reset the tidyverse way, but no keyP

byPeak <- unite(filter(jan.p,
                       batchName %in% c("b802253","b252474")),
                keyP,
                sampleName,
                compoundName,
                chromatogramName,
                sep=":",
                remove=FALSE) # reset and add the variable, without pipes

byPeak <- jan.p %>% # all of the above in one tidyverse pipe statement
  filter(batchName %in% c("b802253","b252474")) %>%
  unite(keyP, sampleName, compoundName, chromatogramName, sep=":", remove=FALSE)
```

Now let's construct our `byPeak$keyB` foreign key by creating and then using a new variable called *analyte*, taking advantage of set operations.

```
allNames <- unique(byPeak$compoundName)
byPeak$analyte <- NA
for (name in allNames[1:6]) {
  compoundPairIdx <- grep(name, allNames)
  theCompound <- intersect(allNames[compoundPairIdx], name)
  theInternalStandard <- setdiff(allNames[compoundPairIdx], name)
  byPeak$analyte[byPeak$compoundName == theInternalStandard] <- theCompound
  byPeak$analyte[byPeak$compoundName == theCompound] <- theCompound
}
```

## Mutating join to add columns

Mutating joins operate in much the same way as the set operations, but on data.frames instead of vectors, and with one critical difference: repeated values are retained. We took advantage of this earlier when using the left_join command, so that the `byBatch$keyB` got repeated for both the Quant and the Qual peak entries in `byPeak`. Having built the `byBatch` primary key, and correctly included it as a foreign key in `byPeak`, correctly joining them into a single data.frame is straightforward.

```
byPeakWide <- left_join(byPeak,byBatch)
```

There are four kinds of mutating joins, differing in how the rows of the source data.frames are treated. In each case, the matching columns are identified automatically by column name and only one is kept, with row order remaining consistent with the principle (usually the left) source. All non-matching columns are returned, and which rows are returned depends on the type of join. An *inner_join(A,B)* only returns rows from A which have a column match in B. The *full_join(A,B)* returns every row of both A and B, using an NA in those columns which don't have a match. The *left_join(A,B)* returns every row of A, and either the matching value from B or an NA for columns with don't have a match. Finally, the *right_join(A,B)* returns every row of B, keeping the order of B, with either the matching value from columns in A or an NA for columns with no match.

Because these commands can duplicate rows, the potential for breaking things is pretty significant if the key isn't unique. Here are two examples, one where you do – and one where you do not – want that duplication:

```
goodDuplication <- inner_join(
  x = bySample[, c(1:3, 7)],
  y = byBatch[, c(1:6)],
  by = c("batchName", "compoundName")
)
badDuplication <- inner_join(
  x = bySample[, c(1:3, 7)],
  y = byBatch[, c(1:6)],
  by = c("compoundName")
)
```

## Filtering join to check the overlap

We created the `byBatch$keyB` explicitly, but it was effectively present already thanks to the *batchName* and *compoundName* columns. The compound naming scheme in `byPeak` remains problematic since the internal standard isn't identified in `byBatch` or `bySample`, but we fixed this using a new column *analyte*. We could have discovered the problem, and then resolved it, using the semi_join and anti_join commands. The *semi_join(A,B)* returns all rows of A where there is a match from B, but keeps only the columns of A, and does not duplicate a row if there are multiple matches. The *anti_join(A,B)* is the inverse, returning all

rows from A where there is no match from B. We still want to create the 'analyte' column for clarity, so one approach would be:

```
byBatch <- jan.b %>% # reset, no keyB
  filter(batchName %in% c("b802253","b252474"))
byPeak <- jan.p %>% # reset, no keyB or keyP
  filter(batchName %in% c("b802253","b252474")) %>%
  mutate(analyte=compoundName)

unique(byPeak$analyte) # notice the similar naming scheme?
byPeak$analyte <- sub("-.*$", "", byPeak$analyte) # notice how we used that similarity?

noMatch <- anti_join(byPeak,byBatch)
noMatch <- anti_join(byPeak,byBatch,by=c("batchName","analyte"="compoundName"))

justMatch <- semi_join(byPeak,byBatch,by=c("batchName","analyte"="compoundName"))
```

> Exercise 3: Join the batch and peak data
> Start from the reset data.frames built in the prior code chunk, so the `keyB` and `keyP` variables are not present. Partially complete commands are commented out in the following code chunk.

```
# exerciseThree <- left_join( , ,by=c( ) )
```

## Summary

- `rbind` and `cbind` add rows (or columns) to an existing data.frame
- `union`, `intersect`, and `setdiff` return a combination of two vectors
- Relational data merges two data.frames on the common columns, called keys
  - A primary key is a unique identifier for every row in a data.frame (the presence of `keyB` in `byBatch`)
  - A foreign key is a unique identifier for another data.frame (the presence of `keyB` in `byPeak`)
- `inner_join`, `full_join`, `left_join`, and `right_join` are mutating joins which add columns
- `semi_join` and `anti_join` are filtering joins which check for overlap