Declarative programming languages such as HTML, CSS, and SQL are popular because they allow users to focus more on the desired *outcome* than the exact computational steps required to achieve that outcome. This can increase efficiency and code readability since programmers describe what they *want* – whether that be how their website is laid out (without worrying about how the browser computes this layout) or how a dataset is structured (regardless of how the database goes about obtaining and aggregating this data).

However, sometimes this additional layer of abstraction can introduce problems of its own. Most notably, the lack of common control flow can introduce a lot of redundancy. This is part of the motivation for *pre-processing* tools which use more imperative programming concepts such as local variables and for-loops to automatically generate declarative code. Common examples in the world of web development are Sass for CSS and Haml for HTML. Of course, such tools naturally come at a cost of their own by requiring developers to learn yet another tool.

For R (or, specifically `tidyverse`) users who need to generate SQL code, recent advances in `dplyr v1.0.0` and `dbplyr v2.0.0` pose an interesting alternative. By using efficient, readable, and most important *familiar* syntax, users can generate accurate SQL queries that could otherwise be error-prone to write. For example, computing sums and means for a large number of variables. Coupled with the power of `sqlfluff`, an innovative SQL styler which was announced at DBT's recent coalesce conference, these queries can be made not only accurate but also imminently readable.

The basic approach

In the following example, I'll briefly walk through the process of generating readable, well-styled SQL using `dbplyr` and `sqlfluff`.

```
library(dbplyr)
library(dplyr)
library(DBI)
```

First, we would connect to our database using the `DBI` package. For the sake of example, I simply connect to an "in-memory" database, but a wide range of database connectors are available depending on where your data lives.

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

Again, *for the sake of this tutorial only*, I will write the `palmerpenguins::penguins` data to our database. Typically, data would already exist in the database of interest.

```
copy_to(con, palmerpenguins::penguins, "penguins")
```

For reference, the data looks like this:

```
head(palmerpenguins::penguins)
```

```
#> # A tibble: 6 x 8
#>   species island bill_length_mm bill_depth_mm flipper_length_忻牯
body_mass_g sex
#>
#> 1 Adelie  Torge忻牯          39.1          18.7               181
3750 male
```

```
#> 2 Adelie  Torge忻犊          39.5          17.4               186
3800 fema忻犊a…
#> 3 Adelie  Torge忻犊          40.3          18                 195
3250 fema忻犊a…
#> 4 Adelie  Torge忻犊          NA            NA                  NA
NA NA
#> 5 Adelie  Torge忻犊          36.7          19.3               193
3450 fema忻犊a…
#> 6 Adelie  Torge忻犊          39.3          20.6               190
3650 male
#> # 忻犊 with 1 more variable: year m
```

Now, we're done with set-up. Suppose we want to write a SQL query to calculate the sum, mean, and variance for all of the measures in our dataset measured in milimeters (and ending in "mm"). We can accomplish this by using the `tbl()` function to connect to our database's data and describing the results we want with `dplyr`'s elegant syntax. This is now made especially concise with select helpers (e.g. `ends_with()`) and the `across()` function.

```
penguins <- tbl(con, "penguins")

penguins_aggr <-
  penguins %>%
  group_by(species) %>%
  summarize(
    N = n(),
    across(ends_with("mm"), sum, .names = "TOT_{.col}"),
    across(ends_with("mm"), var, .names = "VAR_{.col}"),
    across(ends_with("mm"), mean, .names = "AVG_{.col}"),
  )
penguins_aggr
```

```
#> # Source:   lazy query [?? x 11]
#> # Database: sqlite 3.33.0 [:memory:]
#>   species     N TOT_bill_length忻犊 TOT_bill_depth_忻犊
TOT_flipper_len忻犊en…
#>
#> 1 Adelie    152           5858.            2770.            28683
#> 2 Chinst忻犊    68           3321.            1253.
133166
#> 3 Gentoo    124           5843.            1843.            26714
#> # 忻犊 with 6 more variables: VAR_bill_length_mm , VAR_bill_depth_mm
,m
#> #   VAR_flipper_length_mm , AVG_bill_length_mm ,
#> #   AVG_bill_depth_mm , AVG_flipper_length_mm
```

However, since we are using a remote backend, the `penguins_aggr` object does not contain the resulting data that we see when it is printed (forcing its execution). Instead, it contains a reference to the database's table and an accumulation of commands than need to be run on the table in the future. We can access this underlying SQL translation with the `dbplyr::show_query()` and use `capture.output()` to convert that query (otherwise printed to the R console) to a character vector.

```
penguins_query <- capture.output(show_query(penguins_aggr))
```

```
penguins_query

#> [1] ""
#> [2] "SELECT `species`, COUNT(*) AS `N`, SUM(`bill_length_mm`) AS
`TOT_bill_length_mm`, SUM(`bill_depth_mm`) AS `TOT_bill_depth_mm`,
SUM(`flipper_length_mm`) AS `TOT_flipper_length_mm`,
VARIANCE(`bill_length_mm`) AS `VAR_bill_length_mm`,
VARIANCE(`bill_depth_mm`) AS `VAR_bill_depth_mm`,
VARIANCE(`flipper_length_mm`) AS `VAR_flipper_length_mm`,
AVG(`bill_length_mm`) AS `AVG_bill_length_mm`, AVG(`bill_depth_mm`) AS
`AVG_bill_depth_mm`, AVG(`flipper_length_mm`) AS
`AVG_flipper_length_mm`"
#> [3] "FROM `penguins`"
#> [4] "GROUP BY `species`"
```

At this point, we already have a function SQL query and have saved ourselves the hassle of writing nine typo-free aggregation functions. However, since `dbplyr` was not written to generate "pretty" queries, this is not the most readable or well-formatted code. To clean it up, we can apply the `sqlfluff` linter and styler.

As a prerequisite, we slightly reformat the query to remove anything that isn't native to common SQL and will confuse the linter, such as the first line of the query vector: .

```
penguins_query <- penguins_query[2:length(penguins_query)]
penguins_query <- gsub("`", "", penguins_query)
penguins_query

#> [1] "SELECT species, COUNT(*) AS N, SUM(bill_length_mm) AS
TOT_bill_length_mm, SUM(bill_depth_mm) AS TOT_bill_depth_mm,
SUM(flipper_length_mm) AS TOT_flipper_length_mm,
VARIANCE(bill_length_mm) AS VAR_bill_length_mm, VARIANCE(bill_depth_mm)
AS VAR_bill_depth_mm, VARIANCE(flipper_length_mm) AS
VAR_flipper_length_mm, AVG(bill_length_mm) AS AVG_bill_length_mm,
AVG(bill_depth_mm) AS AVG_bill_depth_mm, AVG(flipper_length_mm) AS
AVG_flipper_length_mm"
#> [2] "FROM penguins"
#> [3] "GROUP BY species"
```

After cleaning, we can write the results to a temp file.

```
tmp <- tempfile()
writeLines(penguins_query, tmp)
```

The current state of our file looks like this:

```
SELECT species, COUNT(*) AS N, SUM(bill_length_mm) AS
TOT_bill_length_mm, SUM(bill_depth_mm) AS TOT_bill_depth_mm,
SUM(flipper_length_mm) AS TOT_flipper_length_mm,
VARIANCE(bill_length_mm) AS VAR_bill_length_mm, VARIANCE(bill_depth_mm)
AS VAR_bill_depth_mm, VARIANCE(flipper_length_mm) AS
VAR_flipper_length_mm, AVG(bill_length_mm) AS AVG_bill_length_mm,
AVG(bill_depth_mm) AS AVG_bill_depth_mm, AVG(flipper_length_mm) AS
AVG_flipper_length_mm
```

```
FROM penguins
GROUP BY species
```

Finally, we are ready to use `sqlfluff`. The `lint` command highlights errors in our script, and the `fix` command automatically fixes them (with flags `--no-safety` and `-f` requesting that it apply all rules and does not ask for permission to overwrite the file, respectively). However, note that if your stylistic preferences differ from the defaults, `sqlfluff` is imminently customizable via YAML.

```
system(paste("sqlfluff lint", tmp), intern = TRUE)

#> Warning in system(paste("sqlfluff lint", tmp), intern = TRUE):
running command 'sqlfluff lint C:\Users\emily\AppData\Local\
Temp\RtmpiC6w7c\file8b828936a1' had status 65

#>  [1] "== [C:\\Users\\emily\\AppData\\Local\\Temp\\RtmpiC6w7c\\
file8b828936a1] FAIL"
#>  [2] "L:   1 | P:  29 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#>  [3] "L:   1 | P:  55 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#>  [4] "L:   1 | P:  97 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#>  [5] "L:   1 | P: 142 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#>  [6] "L:   1 | P: 193 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#>  [7] "L:   1 | P: 240 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#>  [8] "L:   1 | P: 290 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#>  [9] "L:   1 | P: 336 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#> [10] "L:   1 | P: 378 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#> [11] "L:   1 | P: 423 | L014 | Inconsistent capitalisation of
unquoted identifiers."
#> [12] "L:   1 | P: 444 | L016 | Line is too long."
#> attr(,"status")
#> [1] 65

# intern = TRUE is only useful for the sake of showing linter results
for this blog post
# it is not needed for interactive use

system(paste("sqlfluff fix --no-safety -f", tmp))

#> [1] 0
```

The results of these commands are a well-formatted and readable query.

```
SELECT
    species,
```

```
    COUNT(*) AS n,
    SUM(bill_length_mm) AS tot_bill_length_mm,
    SUM(bill_depth_mm) AS tot_bill_depth_mm,
    SUM(flipper_length_mm) AS tot_flipper_length_mm,
    VARIANCE(bill_length_mm) AS var_bill_length_mm,
    VARIANCE(bill_depth_mm) AS var_bill_depth_mm,
    VARIANCE(flipper_length_mm) AS var_flipper_length_mm,
    AVG(bill_length_mm) AS avg_bill_length_mm,
    AVG(bill_depth_mm) AS avg_bill_depth_mm,
    AVG(flipper_length_mm) AS avg_flipper_length_mm
FROM penguins
GROUP BY species
```

A (slightly) more realistic example

One situation in which this approach is useful is when engineering features that might include many subgroups or lags. Some flavors of SQL have `PIVOT` functions which help to aggregate and reshape data by group; however, this can vary by engine and even those that do (such as Snowflake) require manually specifying the names of each field. Instead, our `dbplyr` and `sqlfluff` can help generate an accurate query to accomplsh this more concisely.

Now assume we want to find the mean for each measurement separately for years 2007 through 2009. Ultimately, we want these measures organized in a table with one row per species. We can concisely describe this goal with `dplyr` instead of writing out the definition of each of 9 variables (three metrics for three years) separately.

```
penguins_pivot <-
  penguins %>%
  group_by(species) %>%
  summarize_at(vars(ends_with("mm")),
               list(in09 = ~mean(if_else(year == 2009L, ., 0)),
                    in08 = ~mean(if_else(year == 2008L, ., 0)),
                    in07 = ~mean(if_else(year == 2007L, ., 0)))
               )
penguins_pivot

#> # Source:   lazy query [?? x 10]
#> # Database: sqlite 3.33.0 [:memory:]
#>   species bill_length_mm_忻牿 bill_depth_mm_i忻牿 flipper_length_忻牿
bill_length_mm_忻牿mm_…
#>
#> 1 Adelie              13.3            6.19             65.7
12.7
#> 2 Chinst忻牿             17.3            6.47             69.9
12.99
#> 3 Gentoo              17.0            5.34             76.4
17.4
#> # 忻牿 with 5 more variables: bill_depth_mm_in08 ,m
#> #   flipper_length_mm_in08 , bill_length_mm_in07 ,
#> #   bill_depth_mm_in07 , flipper_length_mm_in07
```

Following the same process as before, we can convert this to a SQL query.

```
query <- capture.output(show_query(penguins_pivot))
query <- query[2:length(query)]
query <- gsub("`", "", query)
tmp <- tempfile()
writeLines(query, tmp)
system(paste("sqlfluff fix --no-safety -f", tmp))
```

```
#> [1] 0
```

The following query shows the basic results. In this case, the `sqlfluff` default is significantly more aggressive with identations for the `CASE WHEN` statements than I personally prefer. If I were to use this in practice, I could refer back to the customizable `sqlfluff` rules and either change their configuration or restrict rules I perceived as unaesthetic or overzealous from running.

```
SELECT
    species,
    AVG(
        CASE
            WHEN
                (year = 2009) THEN (bill_length_mm)
            WHEN NOT(year = 2009) THEN (0.0)
        END
    ) AS bill_length_mm_in09,
    AVG(
        CASE
            WHEN
                (year = 2009) THEN (bill_depth_mm)
            WHEN NOT(year = 2009) THEN (0.0)
        END
    ) AS bill_depth_mm_in09,
    AVG(
        CASE
            WHEN
                (year = 2009) THEN (flipper_length_mm)
            WHEN NOT(year = 2009) THEN (0.0)
        END
    ) AS flipper_length_mm_in09,
    AVG(
        CASE
            WHEN
                (year = 2008) THEN (bill_length_mm)
            WHEN NOT(year = 2008) THEN (0.0)
        END
    ) AS bill_length_mm_in08,
    AVG(
        CASE
            WHEN
                (year = 2008) THEN (bill_depth_mm)
            WHEN NOT(year = 2008) THEN (0.0)
        END
    ) AS bill_depth_mm_in08,
```

```
    AVG(
        CASE
            WHEN
                (year = 2008) THEN (flipper_length_mm)
            WHEN NOT(year = 2008) THEN (0.0)
        END
    ) AS flipper_length_mm_in08,
    AVG(
        CASE
            WHEN
                (year = 2007) THEN (bill_length_mm)
            WHEN NOT(year = 2007) THEN (0.0)
        END
    ) AS bill_length_mm_in07,
    AVG(
        CASE
            WHEN
                (year = 2007) THEN (bill_depth_mm)
            WHEN NOT(year = 2007) THEN (0.0)
        END
    ) AS bill_depth_mm_in07,
    AVG(
        CASE
            WHEN
                (year = 2007) THEN (flipper_length_mm)
            WHEN NOT(year = 2007) THEN (0.0)
        END
    ) AS flipper_length_mm_in07
FROM penguins
GROUP BY species
```

When you can’t connect to you data

Even if, for some reason, you cannot connect to R with your specific dataset, you may still use this approach.

For example, suppose we cannot connect to the `penguins` dataset directly, but we a data dictionary we can obtain a list of all of the fields in the dataset.

```
penguins_cols <- names(palmerpenguins::penguins)
```

In this case, we can simple mock a fake dataset using the column names, write it to an in-memory database, generate SQL, and style the output as before.

```
# make fake dataset ----
penguins_mat <- matrix(rep(1, length(penguins_cols)), nrow = 1)
penguins_dat <- setNames(data.frame(penguins_mat), penguins_cols)
penguins_dat

#>   species island bill_length_mm bill_depth_mm flipper_length_mm
body_mass_g sex
#> 1       1      1              1             1                 1
1   1
```

```
#>   year
#> 1    1


# copy to database ----
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = ":memory:")
copy_to(con, penguins_dat, "penguins_mock")
penguins_mock <- tbl(con, "penguins_mock")

# generate sql ----
penguins_aggr <-
  penguins_mock %>%
  group_by(species) %>%
  summarize(
    N = n(),
    across(ends_with("mm"), sum, .names = "TOT_{.col}"),
    across(ends_with("mm"), var, .names = "AVG_{.col}"),
    across(ends_with("mm"), mean, .names = "VAR_{.col}"),
  )

show_query(penguins_aggr)

#>
#> SELECT `species`, COUNT(*) AS `N`, SUM(`bill_length_mm`) AS
`TOT_bill_length_mm`, SUM(`bill_depth_mm`) AS `TOT_bill_depth_mm`,
SUM(`flipper_length_mm`) AS `TOT_flipper_length_mm`,
VARIANCE(`bill_length_mm`) AS `AVG_bill_length_mm`,
VARIANCE(`bill_depth_mm`) AS `AVG_bill_depth_mm`,
VARIANCE(`flipper_length_mm`) AS `AVG_flipper_length_mm`,
AVG(`bill_length_mm`) AS `VAR_bill_length_mm`, AVG(`bill_depth_mm`) AS
`VAR_bill_depth_mm`, AVG(`flipper_length_mm`) AS
`VAR_flipper_length_mm`
#> FROM `penguins_mock`
#> GROUP BY `species`
```

The only caution with this approach is that one should not use *type-driven* select helpers such `summarize_if(is.numeric, ...)` because our mock data has some erroneous types (e.g. `species`, `island`, and `sex` are erroneously numeric). Thus, we could generate SQL that would throw errors when applied to actual data. For example, the following SQL code attempts to sum up islands. This is perfectly reasonably given our dummy dataset but would be illogical and problematic when applied in production.

```
penguins_mock %>%
  group_by(species) %>%
  summarize_if(is.numeric, sum) %>%
  show_query()

#>
#> SELECT `species`, SUM(`island`) AS `island`, SUM(`bill_length_mm`)
AS `bill_length_mm`, SUM(`bill_depth_mm`) AS `bill_depth_mm`,
SUM(`flipper_length_mm`) AS `flipper_length_mm`, SUM(`body_mass_g`) AS
`body_mass_g`, SUM(`sex`) AS `sex`, SUM(`year`) AS `year`
```

```
#> FROM `penguins_mock`
#> GROUP BY `species`
```

Caveats

I have found this combination of tools to be useful for generating readable, typo-free queries when doing a large number of queries. However, I will end by highlighting when this may not be the best approach.

`dbplyr` is not intended to generate SQL. There's always a risk when using tools for something other than their primary intent. `dbplyr` is no exception. Overall, it does an excellent job translating SQL and being aware of the unique flavor of various SQL backends. However, translating between languages is a challenging problem, and sometimes the SQL translation may not be the most computationally efficient (e.g. requiring more subqueries) or semantic approach. For multistep or multitable problems, you may wish to use this approach simple for generating a few painful SQL chunks instead of your whole script.

`dbplyr` *is* intended for you to *not* look at the SQL. One major benefit of `dbplyr` for R users is distinctly to *not* change languages and to benefit from a database's compute power while staying in R. Not only is this use case not the intended purpose, you could go as far as to argue it is almost antithetical. Nevertheless, I do think there are many cases where one should preserve SQL independently; for example, you might need to do data tranformations in a production pipeline that does not run R, not wish to take on additional code dependencies, not be able to connect to your database with R, or be collaborating with non-R users.

`sqlfluff` is still experimental. As the developers emphasized in their DBT talk, `sqlfluff` is still in its early changes and subject to change. While I'm optimistic that this only means this tool will only keep getting better, it's possible the exact rules, configuration, flags, syntax, etc. may change. Check out the docs for the latest documentation there.