Using the **dplyr** frontend to query MIMIC-III

Beverly Anne Setzer Lauren Nicole Geiser Jason Cory Brunson

05 March 2021

Introduction

This tutorial shows how MIMIC-III can be queried using **dplyr**. Only several basic queries are performed, though the **dbplyr** package, which powers SQL queries in **dplyr**, is still maturing, and much functionality exists that is not showcased here.

Example workflow

The following workflow is a simplified version of several scripts used to study medical and social risk factors for heart attack patients. The goal is to prepare an analytic table containing, for each eligible admission, values of several variables that might be used in a statistical analysis. My syntactical conventions — using **tidyverse** packages and, in particular, ending piped function compositions with `%>% print() -> <object name>` — were chosen to make the programming steps as clear as possible.

Acknowledgments

Thanks to Tom Agresta for valuable advice on the tutorial.

If you think this notebook omits some essential functionality, or if it has become out of date, feel free to contact me to suggest it! Or, if you have a clear idea how it could be used in this example workflow, follow the guidelines in the README to contribute to the repo.

Setup

This R Markdown notebook relies on **knitr** to render an HTML document, but users should be able to reproduce its content without that package.

Attach R packages

This minimal introduction relies (directly) on two packages: **RPostgreSQL** connects to databases using PostgreSQL ("Postgres") functionality from the **DBI** package, while **dplyr** provides a grammar of data manipulation based on the same relational algebra as SQL itself. Internally, **dplyr** calls upon the **dbplyr** package in order to connect to a database, translate **dplyr** verbs into SQL queries, and display their results.[1] I'll also use functions from **stringr** in a few queries.

```
library(RPostgreSQL)
```

```
## Loading required package: DBI
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

[1] I originally tried to produce this notebook using **RPostgres**, a newer interface to Postgres developed by the **tidyverse**-adjacent **r-dbi** team. I failed, but that shouldn't discourage anyone from giving it a try. I'll try again myself in a future draft or a separate notebook.

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
library(stringr)
```

Instantiate the MIMIC-III database

I followed the Unix/Mac instructions at PhysioNet to create an instance of MIMIC-III.[2] In particular, i used the user name (`mimicuser`), database name (`mimic`), and schema name (`mimiciii`) suggested there, with the password `mimic`. If you make different choices, then you'll need to change the corresponding parameter values in the `dbConnect()` call below. While the queries in this notebook can be performed on the entire database, a user new to MIMIC, Postgres, or R may want to install the demo data set instead, following the same process except for the CSV files used.[3]

Connect to MIMIC-III

I save the database connection to the variable name `mimic`. Once connected, the `dbListTables()` function should return the names of all tables in the database:

```
mimic <- dbConnect(
  PostgreSQL(),
  dbname = "mimic",
  host = "localhost",
  port = 5432,
  user = "mimicuser",
  password = "mimic"
)
dbListTables(mimic)
```

```
##  [1] "admissions"         "d_icd_diagnoses"    "callout"
##  [4] "caregivers"         "chartevents_1"      "chartevents_2"
##  [7] "chartevents_3"      "chartevents_4"      "chartevents_5"
## [10] "chartevents_6"      "chartevents_7"      "chartevents_8"
## [13] "chartevents_9"      "chartevents_10"     "chartevents_11"
## [16] "chartevents_12"     "chartevents_13"     "chartevents_14"
## [19] "chartevents_15"     "chartevents_16"     "chartevents_17"
## [22] "chartevents"        "cptevents"          "datetimeevents"
## [25] "diagnoses_icd"      "drgcodes"           "d_cpt"
## [28] "d_icd_procedures"   "d_items"            "d_labitems"
## [31] "icustays"           "inputevents_cv"     "inputevents_mv"
## [34] "labevents"          "microbiologyevents" "noteevents"
## [37] "outputevents"       "patients"           "prescriptions"
## [40] "procedureevents_mv" "procedures_icd"     "services"
## [43] "transfers"          "dbplyr_001"
```

[2] The instantiation required some changes to the Postgres commands, e.g. `alter user mimic nosuperuser;` should in fact be `alter user mimicuser nosuperuser;`.

[3] When installing a new database, i find it much more efficient to wrap my steps in an R script that i can execute from the top to erase the baggage from experimentation and errors. I just discovered the **etl** package, and i hope in future to prepare an instantiation process for MIMIC-III from within R using it or a similar framework.

Queries

Inspect and read tables

In **dplyr**, the `tbl()` function, which passes a database connection to the `dplyr:::tbl.DBIConnection()` method, produces a SQL tbl, i.e. an object of class `"tbl_sql"`, which also inherits class `"tbl_lazy"`. (Henceforth i'll just call this a "query table".) This object stores a simple inspection query on a single table and executes it whenever the object is used (e.g., when printed to the console). Indeed, a query table occupies only the memory necessary to recover the query, so that many such objects can be stored in a lightweight R session.

```
patients <- tbl(mimic, dbplyr::in_schema("mimiciii", "patients"))
object.size(patients)
```

```
## 4176 bytes
```

Since conventional practice in R is to read a table into memory in its entirety, this functionality also saves time — at least, until it becomes necessary to manipulate a table in ways that don't translate easily into SQL. When this does become necessary, a `"tbl_sql"` object can be read into R using `collect()`.

```
rbenchmark::benchmark(
  tbl_query = tbl(mimic, dbplyr::in_schema("mimiciii", "d_icd_diagnoses")),
  dbi_read = dbReadTable(mimic, c("mimiciii", "d_icd_diagnoses")),
  tbl_read = collect(tbl(mimic, dbplyr::in_schema("mimiciii", "d_icd_diagnoses"))),
  replications = 24L
)
```

```
##        test replications elapsed relative user.self sys.self user.child
## 2  dbi_read           24   1.032    4.914     0.591    0.094          0
## 1 tbl_query           24   0.210    1.000     0.157    0.007          0
## 3  tbl_read           24   1.388    6.610     0.899    0.094          0
##   sys.child
## 2         0
## 1         0
## 3         0
```

As illustrated above, installing MIMIC-III in a schema imposes the additional step of specifying this schema in each query. I prefer to shortcut this step by defining a MIMIC-specific `tbl` function:

```
tbl_mimic <- function(table) {
  table <- as.character(substitute(table))
  tbl(mimic, dbplyr::in_schema("mimiciii", table))
}
tbl_mimic(patients)
```

```
## # Source:   table<"mimiciii"."patients"> [?? x 8]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    row_id subject_id gender dob                 dod
##     <int>      <int> <chr>  <dttm>              <dttm>
##  1    234        249 F      2075-03-13 00:00:00 NA
##  2    235        250 F      2164-12-27 00:00:00 2188-11-22 00:00:00
##  3    236        251 M      2090-03-15 00:00:00 NA
##  4    237        252 M      2078-03-06 00:00:00 NA
##  5    238        253 F      2089-11-26 00:00:00 NA
##  6    239        255 M      2109-08-05 00:00:00 NA
##  7    240        256 M      2086-07-31 00:00:00 NA
##  8    241        257 F      2031-04-03 00:00:00 2121-07-08 00:00:00
##  9    242        258 F      2124-09-19 00:00:00 NA
```

```
## 10    243        260 F      2105-03-23 00:00:00 NA
## # ... with more rows, and 3 more variables: dod_hosp <dttm>, dod_ssn <dttm>,
## #   expire_flag <int>
```

Beware, though, that this shortcut is not as adaptable as **dplyr** functions and may cause confusion if used in unintended ways, e.g. loops. A less flexible but safer function would omit the first line, requiring the user to always pass a character string to the `table` parameter.

Subset and join query tables

A great deal more can be done with **dbplyr** — that is, without or before reading tables into R — than inspect them. In the code chunks below, i combine data from several tables to build an analytic table of heart attack patients seen at the coronary care unit (CCU).

To begin, i create a query table for the unique admission events for each patient, which serves as the anchor for the rest of the analysis. I want to limit my analysis to patients admitted directly to the CCU, so i'll need care unit information for each admission. For this reason, i query admissions from `transfers`, which includes the `prev_careunit` and `curr_careunit` fields, rather than from `admissions`. I use `filter()` to restrict to CCU admissions; `prev_careunit` takes the value `NA` for admissions from outside the hospital, and in these instances `curr_careunit` indicates the starting unit.

```
tbl_mimic(transfers) %>%
  select(subject_id, hadm_id, prev_careunit, curr_careunit) %>%
  filter(is.na(prev_careunit) & curr_careunit == "CCU") %>%
  select(subject_id, hadm_id) %>%
  distinct() %>%
  print() -> ccu_admissions
```

```
## # Source:   lazy query [?? x 2]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id hadm_id
##         <int>   <int>
##  1      85245  181887
##  2       8861  109419
##  3       4875  155268
##  4      14975  113621
##  5      21265  140906
##  6       9253  185074
##  7      49094  129618
##  8      31153  199555
##  9       7102  112951
## 10       9070  109439
## # ... with more rows
```

Note my use of the pipe operator `%>%` from the **magrittr** package, which is re-exported by **dplyr**. There are several good arguments for using the pipe in manual R scripts, my personal favorite being that i can, in RStudio (where my manual scripting happens), select–execute the first several steps in a piped sequence using `command+return` (`Ctrl+Enter` in Windows).

To restrict to heart attack patients, i need to identify a suitable set of diagnosis codes. I could enter these manually if necessary, but for efficiency i can search for the string "myocardial infarction" in the `"long_title"` field of the `d_icd_diagnoses` table. String searches using **stringr** are translatable into SQL as of 2017), and i use `tolower()` to allow any capitalization.

```
tbl_mimic(d_icd_diagnoses) %>%
  filter(str_detect(tolower(long_title), "myocardial infarction")) %>%
  print() -> mi_codes
```

```
## # Source:   lazy query [?? x 4]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    row_id icd9_code short_title        long_title
##     <int> <chr>     <chr>              <chr>
##  1   4335 41000     AMI anterolateral~ Acute myocardial infarction of anterolat~
##  2   4336 41001     AMI anterolateral~ Acute myocardial infarction of anterolat~
##  3   4337 41002     AMI anterolateral~ Acute myocardial infarction of anterolat~
##  4   4338 41010     AMI anterior wall~ Acute myocardial infarction of other ant~
##  5   4339 41011     AMI anterior wall~ Acute myocardial infarction of other ant~
##  6   4340 41012     AMI anterior wall~ Acute myocardial infarction of other ant~
##  7   4341 41020     AMI inferolateral~ Acute myocardial infarction of inferolat~
##  8   4342 41021     AMI inferolateral~ Acute myocardial infarction of inferolat~
##  9   4343 41022     AMI inferolateral~ Acute myocardial infarction of inferolat~
## 10   4344 41030     AMI inferopost, u~ Acute myocardial infarction of inferopos~
## # ... with more rows
```

I can now look for myocardial infarction (MI) in the diagnosis record for each admission, stored in the `diagnoses_icd` table. Since the relevant codes are contained in a query table, i can use `semi_join()` to restrict to admission entries that match these codes, without keeping any fields from the codes table.

```
tbl_mimic(diagnoses_icd) %>%
  semi_join(mi_codes, by = "icd9_code") %>%
  print() -> mi_admissions
```

```
## # Source:   lazy query [?? x 5]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    row_id subject_id hadm_id seq_num icd9_code
##     <int>      <int>   <int>   <int> <chr>
##  1 175785      15805  188616       1 41000
##  2 281839      25208  167918       7 41000
##  3    594         73  194730       1 41001
##  4   6395        543  115307       1 41001
##  5   6630        571  193189       1 41001
##  6  11015        947  122379       1 41001
##  7  12916       1114  164691       2 41001
##  8  14945       1317  198886       1 41001
##  9  18340       1626  117062       1 41001
## 10  30343       2700  100335       1 41001
## # ... with more rows
```

MI may not be listed as the principal diagnosis; as explained in the documentation for the `patients` table, the `seq_num` field is a priority ranking for the diagnoses generated at the end of stay. In order to focus on patients for whom MI was central to their hospitalization, i will include records with MI in any of the first five diagnosis positions, according to the `"seq_num"` field. To avoid duplicate admissions, i use `group_by()` and `top_n()` to limit the query to the first MI diagnosis for each admission.

```
mi_admissions %>%
  filter(seq_num <= 5) %>%
  group_by(subject_id, hadm_id) %>%
  slice_min(order_by = seq_num) %>%
  ungroup() %>%
  select(subject_id, hadm_id, icd9_code, seq_num) %>%
  print() -> mi_admissions
```

```
## # Source:     lazy query [?? x 4]
## # Database:   postgres 9.6.20 [mimicuser@localhost:5432/mimic]
```

```
## # Ordered by: seq_num
##    subject_id hadm_id icd9_code seq_num
##         <int>   <int> <chr>       <int>
##  1         24  161859 41041           1
##  2         28  162569 412             4
##  3         42  119203 412             4
##  4         53  155385 41021           1
##  5         73  194730 41001           1
##  6         79  181542 41011           1
##  7        108  123552 412             5
##  8        111  192123 41081           5
##  9        123  195632 41011           1
## 10        149  154869 41011           1
## # ... with more rows
```

I now have one query table of admissions to the CCU and another of admissions that included an MI diagnosis. To get the information contained in either table for the admission events contained in both, i inner-join them. While the resulting new table will be annotated with additional fields, it will not be subsetted further, so i just call it `study_admissions`. For a thorough discussion of the joins implemented in **dplyr**, check out the chapter on relational algebra in the book *R for Data Science*.

```
ccu_admissions %>%
  inner_join(mi_admissions, by = c("subject_id", "hadm_id")) %>%
  print() -> study_admissions
```

```
## # Source:   lazy query [?? x 4]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id hadm_id icd9_code seq_num
##         <int>   <int> <chr>       <int>
##  1      49094  129618 41011           1
##  2       6070  118910 41091           1
##  3      90418  195748 41011           3
##  4      54971  112750 41041           1
##  5      83977  195416 41091           1
##  6      12987  156603 41041           1
##  7      11531  161667 41011           1
##  8      17647  180968 41091           5
##  9      87228  194663 41011           1
## 10      77696  121542 41041           1
## # ... with more rows
```

Transform and augment query tables

I made the decision earlier to focus on admissions for which MI was entered into one of the first five diagnosis fields, but it may be useful in the analysis to control for MI being the principal diagnosis. I can introduce a new variable to flag those admissions for which it is first, according to `seq_num`, using the `mutate()` function:

```
study_admissions %>%
  mutate(principal_dx = seq_num == 1) %>%
  select(-seq_num) %>%
  print() -> study_admissions
```

```
## # Source:   lazy query [?? x 4]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id hadm_id icd9_code principal_dx
##         <int>   <int> <chr>     <lgl>
```

```
##  1      49094  129618 41011     TRUE
##  2       6070  118910 41091     TRUE
##  3      90418  195748 41011     FALSE
##  4      54971  112750 41041     TRUE
##  5      83977  195416 41091     TRUE
##  6      12987  156603 41041     TRUE
##  7      11531  161667 41011     TRUE
##  8      17647  180968 41091     FALSE
##  9      87228  194663 41011     TRUE
## 10      77696  121542 41041     TRUE
## # ... with more rows
```

Some records include additional information about the severity of patients' ailments, used for billing purposes. The `drgcodes` table contains, for DRG codes from the All Payers Registry (APR), severity and mortality indicators. I restrict to APR drug codes using another string search, then join severity scores to the admissions query table, using a right-join so as not to drop any admissions who happened to not receive APR drugs. I assign patients with no APR codes the lowest severity score.

```
tbl_mimic(drgcodes) %>%
  filter(str_detect(drg_type, "APR")) %>%
  select(subject_id, hadm_id, drg_severity) %>%
  right_join(study_admissions, by = c("subject_id", "hadm_id")) %>%
  mutate(drg_severity = ifelse(is.na(drg_severity), 1, drg_severity)) %>%
  print() -> study_admissions
```

```
## # Source:   lazy query [?? x 5]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id hadm_id drg_severity icd9_code principal_dx
##         <int>   <int>        <dbl> <chr>     <lgl>
##  1      49094  129618            1 41011     TRUE
##  2      49094  129618            1 41011     TRUE
##  3       6070  118910            1 41091     TRUE
##  4      90418  195748            4 41011     FALSE
##  5      90418  195748            4 41011     FALSE
##  6      54971  112750            1 41041     TRUE
##  7      54971  112750            1 41041     TRUE
##  8      83977  195416            3 41091     TRUE
##  9      83977  195416            3 41091     TRUE
## 10      12987  156603            1 41041     TRUE
## # ... with more rows
```

Finally, i adopt a common outcome measure for critical care: 30-day mortality. I'm interested in survival after discharge, so i must restrict to patients who did *not* die in hospital. This information is recorded in the `"hospital_expire_flag"` field (though not yet described in the `admissions` table documentation; see the tutorial on querying MIMIC-III). I also require the dates (admission and discharge) of each stay from the `admissions` table and the date of death (where available) of each patient from the `patients` table. While i'm working with dates, i'll also calculate each patient's age on the day of admission.

I first join the necessary date fields into `study_admissions`. The syntax gets a bit cluttered here in order to keep the query to one pipeline. This is my own preference; you may prefer, especially while familiarizing yourself with **dplyr**, to cut these into smaller chunks.

```
study_admissions %>%
  left_join(
    select(
      tbl_mimic(admissions),
      subject_id, hadm_id, admittime, dischtime, hospital_expire_flag
```

```
    ),
    by = c("subject_id", "hadm_id")
  ) %>%
  filter(hospital_expire_flag == 0) %>%
  select(-hospital_expire_flag) %>%
  left_join(
    select(tbl_mimic(patients), subject_id, dob, dod),
    by = "subject_id"
  ) %>%
  print() -> study_admissions
```

```
## # Source:   lazy query [?? x 9]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id hadm_id drg_severity icd9_code principal_dx admittime
##         <int>   <int>        <dbl> <chr>     <lgl>        <dttm>
##  1      49094  129618            1 41011     TRUE         2177-05-01 21:03:00
##  2      49094  129618            1 41011     TRUE         2177-05-01 21:03:00
##  3       6070  118910            1 41091     TRUE         2195-01-20 23:53:00
##  4      90418  195748            4 41011     FALSE        2200-07-04 01:24:00
##  5      90418  195748            4 41011     FALSE        2200-07-04 01:24:00
##  6      54971  112750            1 41041     TRUE         2174-01-12 02:07:00
##  7      54971  112750            1 41041     TRUE         2174-01-12 02:07:00
##  8      83977  195416            3 41091     TRUE         2163-05-06 19:25:00
##  9      83977  195416            3 41091     TRUE         2163-05-06 19:25:00
## 10      12987  156603            1 41041     TRUE         2180-02-22 23:30:00
## # ... with more rows, and 3 more variables: dischtime <dttm>, dob <dttm>,
## #   dod <dttm>
```

Functionality for working with dates and times is not yet implemented in **dbplyr**, but an invaluable feature of its SQL translation is that unrecognized functions pass through verbatim, where Postgres will attempt to interpret them. This allows to use `date_part()` below to extract components of timestamp fields as numbers. (Postgres also has a convenient `age()` function that would simplify the code chunk below, but this produces a character string that doesn't lend itself to analysis purposes.) The documentation for the `patients` table explains that patients of 90 years and older had their ages artificially inflated, so i've removed these patients from my analysis. I reorder the fields toward the end in order to show the results of the date calculations. In the last transformation step, `everything()` adds in all the fields i don't explicitly select.

```
study_admissions %>%
  mutate(tt_death = date_part("day", dod) - date_part("day", dischtime)) %>%
  mutate(mortality = tt_death <= 30) %>%
  mutate(age = date_part("year", admittime) - date_part("year", dob)) %>%
  filter(age < 90) %>%
  mutate(age = age - ifelse(
    date_part("month", admittime) < date_part("month", dob) |
      (
        date_part("month", admittime) == date_part("month", dob) &
          date_part("day", admittime) < date_part("day", dob)
      ),
    1,
    0
  )) %>%
  select(-admittime, -dischtime, -dob, -dod, -tt_death) %>%
  select(subject_id, hadm_id, age, mortality, everything()) %>%
  print() -> study_admissions
```

```
## # Source:   lazy query [?? x 7]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id hadm_id   age mortality drg_severity icd9_code principal_dx
##         <int>   <int> <dbl> <lgl>            <dbl> <chr>     <lgl>
##  1      49094  129618    32 NA                   1 41011     TRUE
##  2      49094  129618    32 NA                   1 41011     TRUE
##  3       6070  118910    60 TRUE                 1 41091     TRUE
##  4      90418  195748    38 NA                   4 41011     FALSE
##  5      90418  195748    38 NA                   4 41011     FALSE
##  6      54971  112750    46 NA                   1 41041     TRUE
##  7      54971  112750    46 NA                   1 41041     TRUE
##  8      83977  195416    66 TRUE                 3 41091     TRUE
##  9      83977  195416    66 TRUE                 3 41091     TRUE
## 10      12987  156603    57 TRUE                 1 41041     TRUE
## # ... with more rows
```

Many `mortality` indicators are missing, due to neither the hospital database nor the social security database having a record of these patients' deaths. I could convert these to `FALSE` values, but it may be helpful to retain in the analytic table this information on whether deaths were recorded at all, e.g. for validation or sensitivity testing.

Collect and copy into query tables

The next several steps take advantage of **dbplyr** functionality to read ("collect") query tables into an R session and, more significantly, to join information from an R data frame into a query table *without* reading the query table into R. This can come in handy when, for example, augmenting large database tables with simple categorical values.

This illustration uses demographic information contained in MIMIC-III. Patients' needs vary by sex, and our experiences with health care also tend to reflect both ethnic and gender disparities. These disparities can be accounted for to some extent using two demographic variables: the `ethnicity` field in the `admissions` table and the `gender` field in the `patients` table. I combine them using a full join, so as to include even partial information on any patient for whom it is available, and then use a semi-join to restrict the result to those patients in the `study_admissions` query table.

```
tbl_mimic(admissions) %>%
  select(subject_id, ethnicity) %>%
  distinct() %>%
  print() -> study_subjects
```

```
## # Source:   lazy query [?? x 2]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id ethnicity
##         <int> <chr>
##  1      47247 ASIAN
##  2       2497 WHITE
##  3      32096 WHITE
##  4      12064 WHITE
##  5      29289 WHITE
##  6      24001 UNKNOWN/NOT SPECIFIED
##  7      65985 WHITE
##  8      29021 WHITE
##  9      17663 ASIAN
## 10      60733 WHITE
## # ... with more rows
```

```
tbl_mimic(patients) %>%
  select(subject_id, gender) %>%
  distinct() %>%
  full_join(study_subjects, by = "subject_id") %>%
  print() -> study_subjects
```

```
## # Source:   lazy query [?? x 3]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id gender ethnicity
##         <int> <chr>  <chr>
##  1      13317 M      WHITE
##  2       6317 F      BLACK/AFRICAN AMERICAN
##  3      98347 M      UNKNOWN/NOT SPECIFIED
##  4      98347 M      WHITE
##  5      28215 F      BLACK/AFRICAN AMERICAN
##  6      29208 M      UNKNOWN/NOT SPECIFIED
##  7       6422 F      WHITE
##  8      22426 M      WHITE
##  9      24792 M      WHITE
## 10       2194 F      WHITE
## # ... with more rows
```

```
study_subjects %>%
  semi_join(study_admissions, by = "subject_id") %>%
  print() -> study_subjects
```

```
## # Source:   lazy query [?? x 3]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id gender ethnicity
##         <int> <chr>  <chr>
##  1      29208 M      UNKNOWN/NOT SPECIFIED
##  2       3674 M      UNKNOWN/NOT SPECIFIED
##  3       7616 F      OTHER
##  4      26638 F      WHITE
##  5       3569 M      WHITE
##  6       2534 M      UNKNOWN/NOT SPECIFIED
##  7      22588 M      WHITE
##  8      61115 F      WHITE
##  9       6581 M      WHITE
## 10      12708 F      WHITE
## # ... with more rows
```

There is much diversity and inconsistency in the `ethnicity` field, along with many small numbers. I therefore collapse the field into four main categories (Asian, Black, Hispanic, and white), with a fifth `NA` value for smaller groups. These assignments could be done within query tables; but the `study_admissions` table is already subsetted to the final set of patient admissions, so at this stage it's just as well to commit queries to session memory. This allows us to use the convenient `case_when()` function to collapse the ethnic categories.

```
unknown_ethnicity <- c(
  "OTHER",
  "UNABLE TO OBTAIN",
  "UNKNOWN/NOT SPECIFIED",
  "MULTI RACE ETHNICITY",
  "PATIENT DECLINED TO ANSWER",
  "UNKNOWN"
```

```
)
study_subjects %>%
  collect() %>%
  mutate(ethnic_group = case_when(
    str_detect(ethnicity, "^ASIAN") ~ "ASIAN",
    str_detect(ethnicity, "^BLACK") ~ "BLACK",
    str_detect(ethnicity, "^HISPANIC") ~ "HISPANIC",
    str_detect(ethnicity, "^WHITE") ~ "WHITE",
    ethnicity %in% unknown_ethnicity ~ NA_character_,
    TRUE ~ NA_character_
  )) %>%
  select(subject_id, gender, ethnic_group) %>%
  print() -> study_subjects
```

```
## # A tibble: 1,203 x 3
##    subject_id gender ethnic_group
##         <int> <chr>  <chr>
##  1         24 M      WHITE
##  2         42 M      <NA>
##  3         53 M      <NA>
##  4         73 F      WHITE
##  5        111 F      WHITE
##  6        123 M      HISPANIC
##  7        154 M      WHITE
##  8        158 M      WHITE
##  9        160 F      WHITE
## 10        194 M      BLACK
## # ... with 1,193 more rows
```

In rare cases, a patient is coded as belonging to more than one ethnic group. To resolve these inconsistencies, i've defined a helper function to pick the modal value from a vector of values in R, which can be used by the `summarize()` function to choose one ethnic group for each patient.

```
most <- function(x) {
  if (all(is.na(x))) return(NA_character_)
  y <- table(x, useNA = "no")
  if (length(which(y == max(y))) > 1) return(NA_character_)
  return(names(y)[which.max(y)])
}
study_subjects %>%
  group_by(subject_id) %>%
  summarize(ethnic_group = most(ethnic_group)) %>%
  ungroup() %>%
  mutate(ethnic_group = ifelse(is.na(ethnic_group), "UNKNOWN", ethnic_group)) %>%
  print() -> subject_ethnic_groups
```

```
## # A tibble: 1,188 x 2
##    subject_id ethnic_group
##  *      <int> <chr>
##  1         24 WHITE
##  2         42 UNKNOWN
##  3         53 UNKNOWN
##  4         73 WHITE
##  5        111 WHITE
##  6        123 HISPANIC
```

```
##  7        154 WHITE
##  8        158 WHITE
##  9        160 WHITE
## 10        194 BLACK
## # ... with 1,178 more rows
```

```
study_subjects %>%
  select(subject_id, gender) %>%
  left_join(subject_ethnic_groups, by = "subject_id") %>%
  print() -> study_subjects
```

```
## # A tibble: 1,203 x 3
##    subject_id gender ethnic_group
##         <int> <chr>  <chr>
##  1         24 M      WHITE
##  2         42 M      UNKNOWN
##  3         53 M      UNKNOWN
##  4         73 F      WHITE
##  5        111 F      WHITE
##  6        123 M      HISPANIC
##  7        154 M      WHITE
##  8        158 M      WHITE
##  9        160 F      WHITE
## 10        194 M      BLACK
## # ... with 1,193 more rows
```

While these subject data are small enough to store in memory, it's conceivable that the admissions data remain prohibitively large. So, as a final step, i can join these demographic data into the `study_admissions` query table by introducing a temporary copy of `study_subjects` in the MIMIC-III database, as described in the **dbplyr** documentation.

```
study_admissions %>%
  left_join(study_subjects, by = "subject_id", copy = TRUE) %>%
  print() -> study_admissions
```

```
## Warning in postgresqlWriteTable(conn, name, value, ...): table dbplyr_001 exists
## in database: aborting assignTable
```

```
## # Source:   lazy query [?? x 9]
## # Database: postgres 9.6.20 [mimicuser@localhost:5432/mimic]
##    subject_id hadm_id   age mortality drg_severity icd9_code principal_dx gender
##         <int>   <int> <dbl> <lgl>            <dbl> <chr>     <lgl>        <chr>
##  1      49094  129618    32 NA                   1 41011     TRUE         M
##  2      49094  129618    32 NA                   1 41011     TRUE         M
##  3       6070  118910    60 TRUE                 1 41091     TRUE         M
##  4      90418  195748    38 NA                   4 41011     FALSE        M
##  5      90418  195748    38 NA                   4 41011     FALSE        M
##  6      54971  112750    46 NA                   1 41041     TRUE         M
##  7      54971  112750    46 NA                   1 41041     TRUE         M
##  8      83977  195416    66 TRUE                 3 41091     TRUE         F
##  9      83977  195416    66 TRUE                 3 41091     TRUE         F
## 10      12987  156603    57 TRUE                 1 41041     TRUE         M
## # ... with more rows, and 1 more variable: ethnic_group <chr>
```

The analytic table is now analysis-ready! This query table can be piped into statistical summaries, data visualizations, and other operations that perhaps don't require saving intermittent steps; just beware that some operations may not work with query tables, in which case `collect()` should resolve the problem.

Appendix

For reference, here are my system specs and session info while knitting this notebook:

```
sessioninfo::session_info()
```

```
## - Session info ---------------------------------------------------------------
##  setting  value
##  version  R version 4.0.4 (2021-02-15)
##  os       macOS High Sierra 10.13.6
##  system   x86_64, darwin17.0
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       America/New_York
##  date     2021-03-05
##
## - Packages -------------------------------------------------------------------
##  package     * version date       lib source
##  assertthat    0.2.1   2019-03-21 [1] CRAN (R 4.0.0)
##  blob          1.2.1   2020-01-20 [1] CRAN (R 4.0.0)
##  cli           2.3.1   2021-02-23 [1] CRAN (R 4.0.2)
##  crayon        1.4.1   2021-02-08 [1] CRAN (R 4.0.2)
##  DBI         * 1.1.0   2019-12-15 [1] CRAN (R 4.0.0)
##  dbplyr        2.0.0   2020-11-03 [1] CRAN (R 4.0.2)
##  debugme       1.1.0   2017-10-22 [1] CRAN (R 4.0.0)
##  digest        0.6.27  2020-10-24 [1] CRAN (R 4.0.2)
##  dplyr       * 1.0.4   2021-02-02 [1] CRAN (R 4.0.2)
##  ellipsis      0.3.1   2020-05-15 [1] CRAN (R 4.0.0)
##  evaluate      0.14    2019-05-28 [1] CRAN (R 4.0.0)
##  fansi         0.4.2   2021-01-15 [1] CRAN (R 4.0.2)
##  generics      0.1.0   2020-10-31 [1] CRAN (R 4.0.2)
##  glue          1.4.2   2020-08-27 [1] CRAN (R 4.0.0)
##  htmltools     0.5.0   2020-06-16 [1] CRAN (R 4.0.0)
##  knitr         1.30    2020-09-22 [1] CRAN (R 4.0.2)
##  lifecycle     1.0.0   2021-02-15 [1] CRAN (R 4.0.2)
##  magrittr      2.0.1   2020-11-17 [1] CRAN (R 4.0.2)
##  pillar        1.5.0   2021-02-22 [1] CRAN (R 4.0.0)
##  pkgconfig     2.0.3   2019-09-22 [1] CRAN (R 4.0.0)
##  purrr         0.3.4   2020-04-17 [1] CRAN (R 4.0.0)
##  R6            2.5.0   2020-10-28 [1] CRAN (R 4.0.2)
##  rbenchmark    1.0.0   2012-08-30 [1] CRAN (R 4.0.2)
##  rlang         0.4.10  2020-12-30 [1] CRAN (R 4.0.2)
##  rmarkdown     2.5     2020-10-21 [1] CRAN (R 4.0.0)
##  RPostgreSQL * 0.6-2   2017-06-24 [1] CRAN (R 4.0.0)
##  rstudioapi    0.13    2020-11-12 [1] CRAN (R 4.0.2)
##  sessioninfo   1.1.1   2018-11-05 [1] CRAN (R 4.0.0)
##  stringi       1.5.3   2020-09-09 [1] CRAN (R 4.0.2)
##  stringr     * 1.4.0   2019-02-10 [1] CRAN (R 4.0.0)
##  tibble        3.1.0   2021-02-25 [1] CRAN (R 4.0.2)
##  tidyselect    1.1.0   2020-05-11 [1] CRAN (R 4.0.0)
##  utf8          1.1.4   2018-05-24 [1] CRAN (R 4.0.0)
##  vctrs         0.3.6   2020-12-17 [1] CRAN (R 4.0.2)
##  withr         2.4.1   2021-01-26 [1] CRAN (R 4.0.2)
```

```
##  xfun          0.19    2020-10-30 [1] CRAN (R 4.0.2)
##  yaml          2.2.1   2020-02-01 [1] CRAN (R 4.0.0)
##
## [1] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```