How to Query the YGDP Database

Kaija Gahm

10/6/2020

Contents

Load packages

Introduction

Welcome! This is a guide to querying the YGDP relational database, created by Kaija Gahm in 2020. I'm assuming that you have at least some familiarity using R—you've installed R and RStudio; you've maybe written some very basic R commands before; you know what a data frame is.[1] But I won't assume much advanced knowledge. This is a guide to making some very basic queries from a relational database in R, assuming that you don't know any SQL (which is the database query language that hardcore database people typically use to query databases).

Once you've pulled data out of the database in the format that you need, you can do whatever you want with it. If you already know R, great—you can keep working with the data you queried in an R script. If you prefer to work in Excel or GIS or Stata or whatever other program, I will provide a brief guide to saving your queried data as a .csv file, so that you can access it in a more familiar spreadsheet format and use it to your heart's content.

[1] If you're totally new to R, you can try to follow along here. But if you get stuck, I recommend that you check out the (free, online) book *R for Data Science*. It provides a nice, friendly introduction to R, without assuming any previous programming knowledge.

Why use a relational database?

You're probably familiar with data in spreadsheets, which are "flat" data structures—that is, they have just two dimensions: rows and columns. The problem with storing data in flat sheets is that it limits your ability to efficiently store many different types of data, and you end up with a lot of repeats. Repeat data is a problem not only because it takes up unnecessary space, but also because it makes it much easier for errors to be introduced.

For example, let's say we're storing data about some students. We might want to know their majors[2] and what classes they're taking.

Stored flat, our data table might look something like this (each row is one unique combination of student * class)

```
##    studentID student major                                course
## 1          1   flora  ECON                          Primates 101
## 2          1   flora  ECON            History of Human Language
## 3          1   flora  ECON                               S&DS 230
## 4          1   flora  ECON       Evolution of Social Structures
## 5          2  rashid  LING            History of Human Language
## 6          2  rashid  LING                               Syntax I
## 7          2  rashid  LING Grammatical Diversity in U.S. English
## 8          2  rashid  LING                            English 120
## 9          2  rashid  LING                                  ASL I
## 10         3    ming  E&EB       Evolution of Social Structures
## 11         3    ming  E&EB                            Ornithology
## 12         3    ming  E&EB                            English 120
## 13         3    ming  E&EB                            Physics 170
## 14         3    ming  E&EB                               S&DS 230
```

You can already see that there are lots of repeats. Because each student takes many classes, we have to have several rows per student. But because each student has only one major and only one ID number, that information has to be repeated over and over again.

Database anomalies

There are a few particular types of errors, known as **anomalies**, that can happen if we store data with lots of repeats like this.

Update Anomaly

An **update anomaly** occurs when some data in the database is updated, but because that data appears in multiple places, not all occurrences of the information get updated.

For example, let's say that someone tells us that Flora's major is actually Anthropology (ANTH), not Economics (ECON). In order to correct this information, we have to change "ECON" to "ANTH" in four separate places. It would be easy to miss one, ending up with something like:

```
##    studentID student major                                course
## 1          1   flora  ANTH                          Primates 101
## 2          1   flora  ECON            History of Human Language
## 3          1   flora  ANTH                               S&DS 230
## 4          1   flora  ANTH       Evolution of Social Structures
## 5          2  rashid  LING            History of Human Language
## 6          2  rashid  LING                               Syntax I
## 7          2  rashid  LING Grammatical Diversity in U.S. English
## 8          2  rashid  LING                            English 120
## 9          2  rashid  LING                                  ASL I
## 10         3    ming  E&EB       Evolution of Social Structures
## 11         3    ming  E&EB                            Ornithology
## 12         3    ming  E&EB                            English 120
## 13         3    ming  E&EB                            Physics 170
## 14         3    ming  E&EB                               S&DS 230
```

[2]For the purposes of this toy example, let's pretend that each student only has one major. Just bear with me :)

Now, depending on which row you look at, you might conclude that flora is either an Anthropology or an Economics major. Which one is the truth? By failing to update consistently, we've introduced uncertainty about the correctness of the information in our database.

Deletion Anomaly

A deletion anomaly occurs when data is lost unintentionally when *other* data is deleted.

For example, let's say that when COVID-19 hits, Rashid decides to take a leave of absence because he's a sophomore and wasn't going to be allowed back on campus for the fall semester. Previously, Rashid had been registered for five classes, but now we have to update this semester's data to show that he's now registered for none.

Our table will now look like this (assuming we've fixed the problem with Flora's major above:

```
##   studentID student major                         course
## 1         1   flora  ANTH                   Primates 101
## 2         1   flora  ANTH      History of Human Language
## 3         1   flora  ANTH                       S&DS 230
## 4         1   flora  ANTH                          ASL I
## 5         3    ming  E&EB Evolution of Social Structures
## 6         3    ming  E&EB                    Ornithology
## 7         3    ming  E&EB                    English 120
## 8         3    ming  E&EB                    Physics 170
## 9         3    ming  E&EB                       S&DS 230
```

Now Rashid's classes aren't listed—but neither is any information about him! In addition to his class registrations, we have lost Rashid's student ID number and his major. Because that information was stored all together, it got unintentionally deleted when the courses were deleted. Losing information like this is really dangerous. While of course, with a small table like this, it would have been really easy to see ahead of time that this would be a problem, it's much harder to anticipate errors like this when your data is huge and complicated. Storing your data in a relational database is a way to have peace of mind that you won't accidentally delete extra information when you make changes.

Insertion Anomaly

An insertion anomaly occurs when data cannot be added to the database unless additional information is known. This can be a bit of a moot point if your data structure allows missing values, but let's assume for the sake of demonstration that it doesn't. Now we have a problem. Let's say we'd like to add a first-year student, James, to our table, but they haven't declared a major yet. Because the column "major" cannot be blank, we're stuck. Even if James knows very well which classes they're taking this semester, our data setup won't allow us to enter them unless James declares a major.

How can we avoid these problems?

The best way to avoid problems like these, and to make our data structure efficient and consistent so that it can contain data for years to come, is to use a normalized relational database. A relational database is nothing more than a collection of flat data sheets, each of which *relate* to others by "key" columns.

I won't go into detail about normalization and the rules governing database structure here (Google "database normalization"! It's fascinating!), but the basic idea is to separate data into different tables that represent information minimally. Each table has a "primary key" column, whose values serve as a unique identifier for each row.

Example relational database

Here's how we could make a database out of the student information table above.

```
## STUDENTS
```

```
##   studentID studentName
## 1        S1       flora
## 2        S2      rashid
## 3        S3        ming
```

```
## MAJORS
```

```
##   majorID                      majorName
## 1    ANTH                    Anthropology
## 2    LING                     Linguistics
## 3    E&EB Ecology & Evolutionary Biology
```

```
## COURSES
```

```
##    courseID                            courseName
## 1        C1                          Primates 101
## 2        C2             History of Human Language
## 3        C3                               S&DS 230
## 4        C4        Evolution of Social Structures
## 5        C5                               Syntax I
## 6        C6 Grammatical Diversity in U.S. English
## 7        C7                            English 120
## 8        C8                                  ASL I
## 9        C9                            Ornithology
## 10      C10                            Physics 170
```

Each of these tables has a *primary key,* whose name (in this case) ends with "ID". Each row in STUDENTS represents one student, and is identified by `studentID`. Each row in MAJORS represents one major, and is identified by `majorID`. Each row in COURSES represents one course, and is identified by `courseID`.

The tables above store the key information about courses, students, and majors. But we still need to link them. For this, we'll use linker tables. For example, we could have a table called `STUDENT_COURSES` that simply links students to which courses they're taking:

```
## STUDENT_COURSES
```

```
##    studentID courseID
## 1         S1       C1
## 2         S1       C2
## 3         S1       C3
## 4         S1       C4
## 5         S2       C2
## 6         S2       C5
## 7         S2       C6
## 8         S2       C7
## 9         S2       C8
## 10        S3       C4
## 11        S3       C9
## 12        S3       C7
## 13        S3      C10
## 14        S3       C3
```

Unlike the tables I showed above, this `STUDENT_COURSES` table does *not* have a single primary key column. Instead, each row represents a unique combination of `studentID` and `courseID`, so we would say that this table has a "composite primary key". In addition, each of those columns serves as a *foreign key*: a column in a table that joins to the *primary key* in a different able. Think of it as a cross-reference. So, `studentID` is a foreign key that joins to the `STUDENTS`t table, and `courseID` is a foreign key that joins to the `COURSES` table.

We could imagine a similar linker table to define which courses correspond to which major. We could also add additional tables for students' personal information, course reviews and professors, and on and on, keeping the information in each table minimal and joining tables together by their primary and foreign keys.

There's a lot more database terminology I could introduce here, but I think the best way to learn is by doing. At the end of this document, I provide links to additional resources, if you'd like to learn more. I will also explain some R terminology and commands that we use to explore database tables, and I'll show some examples of how to query a database from R.

First, though, I'm going to take the time to describe the tables you'll find as part of the YGDP database. For each table, I'll give a plain description of what information it contains. I'll note what each unique row represents and what the primary key is. And I'll list out the foreign keys (connections to other tables). I hope these descriptions, as well as the example queries that follow, will be helpful to you as you plan out your own queries and start to use the YGDP data in your analyses.

Database schema

Here is a database schema for the YGDP database. A diagram like this is also called an "Entity-Relationship Diagram", where entities are the database tables and relationships are the ways in which the tables join together by key variables.

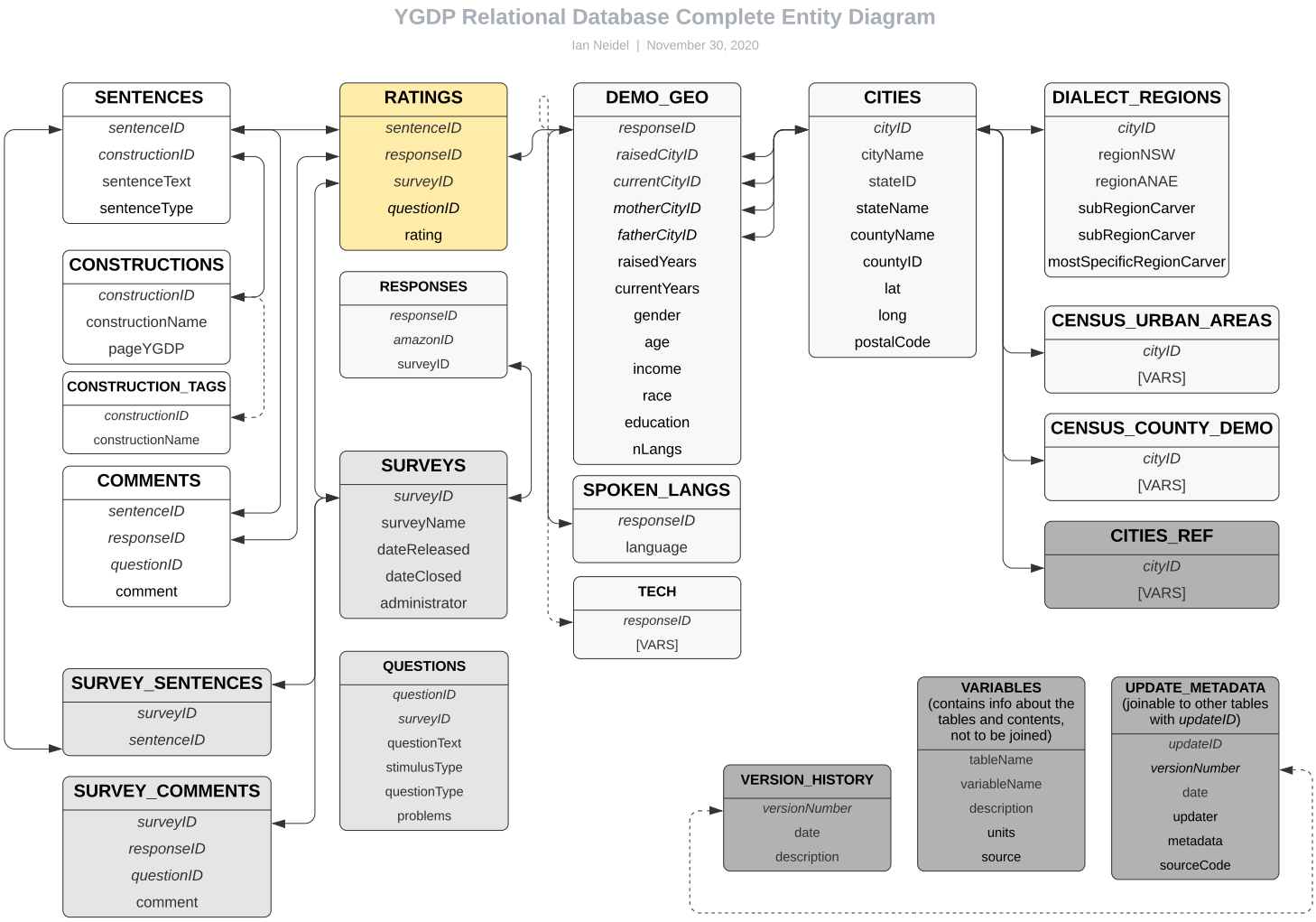

Figure 1: Entity-Relationship Diagram for the YGDP Database (by Ian Niedel)

Table descriptions

CENSUS_COUNTY_DEMO

This table contains county-level demographic information from the 2010 U.S. Census. The information comes from these ESRI shapefiles.

Each row corresponds to one `cityID`, but the values in the columns are population/demographic/area values for the *county in which that city is located.*

Relations:

`cityID` joins to CITIES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "census_county_demo")
```

```
##  [1] "cityID"       "countyName"   "stateName"    "STATE_FIPS"   "CNTY_FIPS"
##  [6] "FIPS"         "POPULATION"   "POP_SQMI"     "POP2010"      "POP10_SQMI"
## [11] "WHITE"        "BLACK"        "AMERI_ES"     "ASIAN"        "HAWN_PI"
## [16] "HISPANIC"     "OTHER"        "MULT_RACE"    "MALES"        "FEMALES"
```

```
## [21] "AGE_UNDER5"   "AGE_5_9"      "AGE_10_14"    "AGE_15_19"    "AGE_20_24"
## [26] "AGE_25_34"    "AGE_35_44"    "AGE_45_54"    "AGE_55_64"    "AGE_65_74"
## [31] "AGE_75_84"    "AGE_85_UP"    "MED_AGE"      "MED_AGE_M"    "MED_AGE_F"
## [36] "HOUSEHOLDS"   "AVE_HH_SZ"    "HSEHLD_1_M"   "HSEHLD_1_F"   "MARHH_CHD"
## [41] "MARHH_NO_C"   "MHH_CHILD"    "FHH_CHILD"    "FAMILIES"     "AVE_FAM_SZ"
## [46] "HSE_UNITS"    "VACANT"       "OWNER_OCC"    "RENTER_OCC"   "NO_FARMS17"
## [51] "AVE_SIZE17"   "CROP_ACR17"   "AVE_SALE17"   "SQMI"         "Shape_Length"
## [56] "Shape_Area"   "updateID"
```

CENSUS_URBAN_AREAS

Contains information on census legal/statistical areas, focusing on land use (urban, rural, land area, water area), etc. The information comes from the US Census TIGER/Line Shapefiles.

Each row corresponds to one `cityID`, but the values in the columns are population/demographic/area values for the *census area in which that city is located*.

Relations:

`cityID` joins to CITIES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "census_urban_areas")
```

```
##  [1] "cityID"     "UACE10"     "GEOID10"    "NAME10"     "NAMELSAD10"
##  [6] "LSAD10"     "MTFCC10"    "UATYP10"    "FUNCSTAT10" "ALAND10"
## [11] "AWATER10"   "INTPTLAT10" "INTPTLON10" "updateID"
```

CITIES

Contains information about individual cities. Each row is a unique pair of lat/long coordinates that has been assigned a unique `cityID`. Lat/long coordinates, `postalCode`s, `cityName`s, and `countryID`'s were obtained by geocoding using the HERE REST API, via the hereR package. However, the `stateID`, `stateName`, and `countyName` columns are sourced instead from the ESRI USA Counties shapefile. This was done because there were occasionally slight discrepancies in state/county values, for example if the lat/long coordinates fell very close to a county and/or state boundary. I elected to defer to the state/county designations from the census shapefiles in order to obtain correct demographic information for these points on joining to the CENSUS_COUNTY_DEMO or CENSUS_URBAN_AREAS table.

Relations:

`cityID` is the primary key and joins to CENSUS_COUNTY_DEMO, CENSUS_URBAN_AREAS, and CITIES_REF. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "cities")
```

```
##  [1] "cityID"     "cityName"   "stateID"    "stateName"  "countyName"
##  [6] "countryID"  "lat"        "long"       "postalCode" "updateID"
```

CITIES_REF

This table attempts to join participants' "raw" survey responses to the cityID they were eventually assigned.

There is no single primary key. The composite primary key is made up of `city`, `state`, and `country`. Each row is one city/state/country combination, as entered by participants into the survey (and modified for processing by me).

Modifications were as follows:

- Converted all state abbreviations to state names
- Made everything lowercase
- Corrected some misspellings manually, e.g. "untied states of america"

- In surveys that had one column for each state, of which one contained a city name, I condensed the information to put it in city, state format.
- Some additional modifications depending on the survey—see individual survey processing scripts for more details.

After geocoding these city/state/country combinations, they were assigned a `cityID` in the process of creating the CITIES table. I then chose to preserve the combination of the `cityID` and the `city`, `state`, and `country` information.

In future survey processing, the participants' entries can be first checked against the entries in this table to see if they can be assigned a `cityID` before moving on to the geocoding step. In fact, I have already used this method in processing many of the surveys (see individual survey processing scripts), and it cut down on the number of geocoding queries considerably.

Relations:

`cityID` joins to CITIES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "cities_ref")
```

```
## [1] "city"     "state"    "country"  "cityID"   "updateID"
```

COMMENTS

This table contains comments from survey participants on individual sentences that they judged. The composite primary key is the combination of `responseID` and `sentenceID`. I have removed response/sentence combinations for which the participant did not leave a comment.

Relations:

`responseID` joins to RESPONSES. `sentenceID` joins to SENTENCES. `questionID` joins to QUESTIONS. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "comments")
```

```
## [1] "responseID" "sentenceID" "comment"    "questionID" "updateID"
```

CONSTRUCTIONS

Basic information about grammatical constructions, including their names and a link to their YGDP page (if applicable). `constructionID` is the primary key; each row is a single construction.

Relations:

`constructionID` joins to SENTENCES and CONSTRUCTION_TAGS. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "constructions")
```

```
## [1] "constructionID"   "constructionName" "pageYGDP"         "updateID"
```

CONSTRUCTION_TAGS

This table is still very incomplete. In the future, it will contain "tags", such as "negation" or "objects" or "modals" to help categorize the grammatical constructions. This will allow you to ask questions like "show me all ratings on sentences related to negation".

We are still trying to figure out how to handle nested data. For example, "Needs Washed–Need" is a subcategory of "Needs Washed". Do we list each subcategory separately? If so, does "Needs washed" itself become a tag (even though most people would probably think of it as a construction in its own right)? Or do we introduce another table to deal with hierarchical constructions like this?

```
dbListFields(con, "construction_tags")
```

```
## [1] "constructionID"   "constructionName" "updateID"
```

DIALECT_REGIONS

This table places each `cityID` into each of several dialect region classification schemes. The `regionNSW` categories divide the country into "The North", "The South", and "The West"–I don't know exactly where they came from, but I think Jim created them somehow. The `region` column (which will soon be renamed to `regionANAE`–that's a mistake) is for the regions defined by the Atlas of North American English (ANAE). Then there are four columns for the Carver dialect regions: `regionCarver`, `subRegionCarver`, `subSubRegionCarver`, and `mostSpecificRegionCarver`. The first three are defined by Carver; the last one is defined by me. `mostSpecificRegionCarver` is equal to `subSubRegionCarver` if it exists for that city; if not, it is equal to `subRegionCarver` if that exists; if not, it is equal to `regionCarver` if *that* exists, and if not, then it is NA.

Relations:

`cityID` joins to CITIES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "dialect_regions")
```

```
## [1] "cityID"                   "regionNSW"
## [3] "regionANAE"               "regionCarver"
## [5] "subRegionCarver"          "subSubRegionCarver"
## [7] "mostSpecificRegionCarver" "updateID"
```

DEMO_GEO

This table contains the demographic and geographic information provided by survey participants. The primary key is `responseID`, so each row is a single response (typically equivalent to a single individual person, although theoretically a person could take more than one survey and end up with more than one responseID. We generally assume this to account for a negligible number of participants).

The four geographic columns in this table are `currentCityID`, `raisedCityID`, `motherCityID`, and `fatherCityID`, which each join to the CITIES table to provide the geographic information for the respective city. For most analyses, we use `raisedCityID` because we want to know about where the participant grew up. `currentYears` is the number of years that the participant has spent in their current location, and `raisedYears` is the number of years they spent in their primary childhood residence (the place represented by `raisedCityID`. We typically filter out survey participants who spent less than 8 years in their primary childhood residence. For more information on these filters and on our survey methods, see the mapbook.](https://ling.auf.net/lingbuzz/005277).)

The rest of the columns are demographic information. `income` is sorted into brackets: the number in the column (e.g. 12500, 87500) represents the low end of each bracket.[3] There are two race-related columns. `race` is a semi-standardized column that includes the race designation that the participant selected (or entered as text, if they selected "other") on the survey. It has been converted to lowercase but otherwise has been minimally modified. `raceCats` has race values sorted into standardized categories: other, asian, black, hispanic/latinx, NA, native american, pacific islander, white.[4]

`nLangs` is the number of languages besides English spoken by the survey participant. Note that information on what these languages actually are is stored in the SPOKEN_LANGS table, because the structure is different—more than one row per participant is allowed in that table.

Relations:

`responseID` joins to RESPONSES (and many other tables which also have it as a foreign key). `currentCityID`, `raisedCityID`, `motherCityID`, and `fatherCityID` each join to CITIES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "demo_geo")
```

```
##  [1] "responseID"    "currentCityID" "raisedCityID"  "motherCityID"
##  [5] "fatherCityID"  "currentYears"  "raisedYears"   "gender"
##  [9] "age"           "income"        "race"          "raceCats"
## [13] "education"     "nLangs"        "updateID"
```

[3] This means that some participants are recorded as having an income of "1". This may or may not be the case—"1" represents the income bracket ranging from $1 to $12,499 per year.

[4] These categories correspond to the choices on the surveys, with some spelling/terminology edits for brevity. NA indicates a missing value; it is not shorthand for "native american". Note that I have converted these categories to lowercase letters, not to ignore preferred capitalizations such as "Black", but simply to standardize the data for data cleaning/management purposes. Analyses and visualizations that use these data should edit capitalization appropriately.

QUESTIONS

This table is still incomplete, but it will contain information about the questions asked on the surveys. `questionID` is the primary key, a unique combination of the question ID number and the survey on which it occurs (since the same question format could occur on multiple surveys).

`problems` is a column I thought to add when I was looking through participants' comments and one of them mentioned a problem with one of the questions. We can use this column in the future to flag questions.

Relations:

`questionID` is the primary key and joins to COMMENTS, RATINGS, and SURVEY_COMMENTS. `surveyID` joins to RESPONSES, SURVEYS, SURVEY_COMMENTS, and SURVEY_SENTENCES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "questions")
```

```
## [1] "questionID"   "questionText" "surveyID"     "stimulusType" "scaleOptions"
## [6] "problems"     "updateID"
```

RATINGS

This is a very central table: it's where the **data** in the form of participants' ratings (or, "judgments") is located. The primary key is a composite of `responseID` and `sentenceID`–so, one row for each participant/sentence combination.

`rating` is an integer between 1 and 5, or (rarely) NA if the participant somehow didn't judge a sentence they were supposed to.

Relations:

`responseID` joins to RESPONSES. `sentenceID` joins to SENTENCES. `questionID` joins to QUESTIONS. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "ratings")
```

```
## [1] "responseID" "sentenceID" "surveyID"   "questionID" "rating"
## [6] "updateID"
```

RESPONSES

The function of this table is twofold: to match `responseID`'s to `amazonID`'s, and to match `responseID`'s to `surveyID`'s. `responseID` is the primary key.

Relations:

`responseID` is the primary key and joins to several other tables. `surveyID` joins to SURVEYS. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "responses")
```

```
## [1] "responseID" "amazonID"   "surveyID"   "updateID"
```

SENTENCES

This table stores sentence-level information. Its primary function is to relate `sentenceID` to `sentenceText` (so you can see what the sentence actually *says*, without us having to use the sentence text itself over and over again in different tables).

`sentenceType` categorizes sentences as a "test sentence" (TS), a grammatical control (CG), or an ungrammatical control (CU). I haven't attempted to make any more specific categorizations, since the same test sentence could take on different roles in different surveys (e.g. pilot vs. not, etc.) If that kind of information is something we need to capture, we would need to create an additional table.

`constructionID`, of course, tell us which grammatical construction/phenomenon the sentence is designed to test.

Relations:

`sentenceID` is the primary key and joins to COMMENTS, RATINGS, and SURVEY_SENTENCES. `constructionID` joins to CONSTRUCTIONS and CONSTRUCTION_TAGS. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "sentences")
```

```
## [1] "sentenceID"     "sentenceText"   "sentenceType"   "constructionID"
## [5] "updateID"
```

SPOKEN_LANGS

This table contains information on which languages, other than English, participants speak. Despite the instructions to this effect in the surveys, many participants also listed English; I have removed English in those cases. I have also put all language names into lowercase.

The primary key is a composite of `responseID` and `language` —there are several rows per participant if the participant listed several languages.

Relations:

`responseID` joins to RESPONSES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "spoken_langs")
```

```
## [1] "responseID" "language"   "updateID"
```

SURVEYS

This table contains data about surveys: who, what, when.

When I didn't have actual `dateReleased` and `dateClosed` values from Jim, I filled in the information by looking at the earliest and latest dates when a participant took the survey. So, it is possible that there was some time at the beginning or end of the survey period during which no one took the survey, and that would not be reflected in these dates.

Relations:

`surveyID` is the primary key and joins to QUESTIONS, RESPONSES, SURVEY_COMMENTS, and SURVEY_SENTENCES. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "surveys")
```

```
## [1] "surveyID"      "surveyName"    "dateReleased"  "dateClosed"
## [5] "administrator" "updateID"
```

SURVEY_COMMENTS

Contains comments that people had about surveys (rather than sentences). Usually the survey question for these was called "General comments" or similar. The primary key is a composite of `responseID` and `surveyID`.

Relations:

`responseID` joins to RESPONSES (and a whole bunch of other tables). `surveyID` joins to SURVEYS, of course. `questionID` joins to QUESTIONS. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "survey_comments")
```

```
## [1] "responseID" "surveyID"   "questionID" "comment"    "updateID"
```

SURVEY_SENTENCES

This table shows which sentences were tested on which surveys. The primary key is a composite of `sentenceID` and `surveyID`– there is no other information, because this is a linker table.

Relations:

`sentenceID` joins to SENTENCES. `surveyID` joins to SURVEYS. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "survey_sentences")
```

```
## [1] "sentenceID" "surveyID"   "updateID"
```

TECH

This table contains all the tech information for each participant (well, for each responseID, at least). I don't know why we would need this information, but Qualtrics collects it, so why not, right?

`locationLat` and `locationLong` are a little cryptic, but I guess that was the location of the participant's computer when they took the survey? Not necessarily the same as their childhood or current residence.

Each row is a single `responseID`.

Relations:

`responseID` is the primary key and joins to RESPONSES and to all the other various `responseID` tables. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "tech")
```

```
##  [1] "responseID"      "dateTimeStart"   "dateTimeEnd"     "ipAddress"
##  [5] "progress"        "durationSeconds" "finished"        "recordedDate"
##  [9] "locationLat"     "locationLong"    "userLanguage"    "consent"
## [13] "browser"         "version"         "operatingSystem" "resolution"
## [17] "hasAmazonID"     "completionCode"  "updateID"
```

UPDATE_METADATA

This is the table we use to track updates to the database. Notice how all the other database tables before this one have an `updateID` column that joins to this table?

In UPDATE_METADATA, the `updateID` is the primary key. For each updateID, this table contains information on when the update happened (`date`), who made the update (`updater`), and what the update consisted of: what change was made, why, how many rows were affected, etc. (`metadata`). `versionNumber` connects individual updates to the database version they belong to, and `sourceCode` points to the exact script where the update code was written.

Relations:

`updateID` is the primary key and joins to literally every other table except for VERSION_HISTORY. `versionNumber` joins to VERSION_HISTORY.

```
dbListFields(con, "update_metadata")
```

```
## [1] "updateID"      "date"          "updater"       "metadata"
## [5] "versionNumber" "sourceCode"
```

VARIABLES

This table gives the units (if applicable) and a description for the variables in each database table. `source` notes where the variable came from: e.g. from the census shapefiles, generated by Qualtrics, entered by the survey participant, or generated/created internally to the YGDP.

Relations:

`tableName` doesn't *join,* exactly, but it refers to the names of all the other database tables. Same with `variableName`: it refers to the names of the variables (columns) in the other tables. `updateID` joins to UPDATE_METADATA.

```
dbListFields(con, "variables")
```

```
## [1] "tableName"    "variableName" "description"  "units"        "source"
## [6] "updateID"
```

VERSION_HISTORY

Summarizes the updates/changes made in each new version of the database. Toward the beginning of the database's life, when I was adding data from each survey, I treated each survey addition as a new version. Later on, when the changes to the database are mainly fixes affecting different tables, I'll build up a collection of fixes and then push them through all at once and call it a new version.

Relations:

`versionNumber` joins to UPDATE_METADATA.

```
dbListFields(con, "version_history")
```

```
## [1] "versionNumber" "date"          "description"
```

Some important R and database terminology

In this section, I will introduce and explain some functions in R that I will use to write sample queries in the next section. I'll also go over some terms and conventions that you might see me use.

Some of these functions are core R functions; others come from the `dplyr` package, which I will use extensively to work with data and pull data from the YGDP database.

Packages

A "package" in R (also known as a library) is like an external toolkit. To use a package in R, you have to do two things.

1. **Install the package.** For example, `install.packages("dplyr")`. You only do this *once per R session*—that is, you should put this code at the top of your script and run it over and over again. If a package is a special toolkit from the hardware store, think of installing the package as "buying the toolkit". Once you've bought it, you have it, and you don't need to buy it again.
2. **Load the package.** For example, `library(dplyr)` . Think of this as "taking the toolkit off the shelf and opening it". You have to do this *in every script*. Usually, R scripts start with a bunch of `library()` lines to load all the required packages. In this document, you can see those `library()` calls here.

NOTE: Sometimes you might see the syntax `package::function()`. The `::` is used if you just want to import one particular function from a package without loading the whole package (or even if you *have* loaded the whole package but you just want to be really specific about where the function comes from).[5] So, for example, you might see a code snippet that says something like `df %>% select()`, but equivalently, you might see `df %>% dplyr::select()`. Don't panic. That means the same thing, it just specifies that the `select()` function comes from the `dplyr` package, and it avoids any confusion with other packages that might have a `select()` function too.

Queries

- A "query" is a "request for information" from a database. In order to retrieve information efficiently from a database, we phrase our request in very specific terms that R can understand and send to the database in order to retrieve the information we asked for. Usually, database queries are written in SQL ("Structured Query Language"), a programming language designed specifically for querying databases. However, the R packages `dplyr` and `dbplyr` are designed to translate R syntax into SQL queries. This is useful if you (like me!) know R but don't know SQL. If you know SQL but not R, and you'd rather query the YGDP database using SQL, that's fine too! The database is stored as an SQLite (.db) file.

General R syntax

- `%>%` is called a "pipe". It takes the output of the thing to its left and "pipes" that into the function on its right. You can think of it as translating to "and then. . . ". For example:

[5] You might do this to avoid what are called "namespace conflicts", which is when two different packages have a function by the same name.

```r
df <- data.frame(animal = c("fish", "rabbit", "bird"),
                 locomotion = c("swimming", "hopping", "flying"),
                 habitat = c("water", "land", "air")) # create some sample data

df %>% # take the object df AND THEN...
  # filter it to keep only rows where "animal" is not
  # equal to "fish" AND THEN...
  filter(animal != "fish") %>%
  # select the "locomotion" and "habitat" columns.
  select(locomotion, habitat)
```

```
##   locomotion habitat
## 1    hopping    land
## 2     flying     air
```

- `==` means "is equal to", as opposed to `=`, which is used for assignment and to define arguments inside of functions.
- `!=` means "is not equal to".
- `<=` means "is less than or equal to", and `>=` means "is greater than or equal to". Naturally, `<` means "is less than" and `>` means "is greater than".
- `%in%` means "is contained in". For example, `"cat" %in% c("cat", "dog")` would evaluate to `TRUE`, because the string `"cat"` is one of the elements of the vector `c("cat", "dog")`.
- `filter` is used to filter **rows** based on certain characteristics. For example:

```r
# filters the data frame to contain only rows where
# the value in the "animal" column is equal to (==) "fish".
df %>% filter(animal == "fish")
```

```
##   animal locomotion habitat
## 1   fish   swimming   water
```

```r
# you can combine multiple filters in one filter() command
# using AND (&) or OR (|).
df %>%
  filter(animal == "fish"|locomotion == "flying")
```

```
##   animal locomotion habitat
## 1   fish   swimming   water
## 2   bird     flying     air
```

```r
# this filters the data frame to contain only rows where
# the value in the "animal" column is "fish", OR where the
# value in the "locomotion" column is "flying".
```

- `select` is used to select **columns** from a data frame. For example:

```r
df %>%
  select(animal, locomotion)
```

```
##   animal locomotion
## 1   fish   swimming
## 2 rabbit    hopping
## 3   bird     flying
```

```
# selects the "animal" and "locomotion" columns from `df`
```

- `left_join` joins multiple data frames together by a common key. There are many types of join operations, but most of the examples below just use `left_join`, so I'll keep it simple for now. (Read more about relational data and joins) For example:

```
# Make a second data frame to join to the first
df2 <- data.frame(animal = c("fish", "bird", "frog"),
                  bodyPart = c("fin", "wing", "leg"))

# Join it to the first with a left_join(),
# using "animal" as the common column
df %>%
  left_join(df2, by = "animal")
```

```
##   animal locomotion habitat bodyPart
## 1   fish   swimming   water      fin
## 2 rabbit    hopping    land     <NA>
## 3   bird     flying     air     wing
```

```
# notice that we have an NA (a missing value) in the bodyPart column,
# since df2 did not contain any information for "rabbit".
# Also notice that the row for "frog" did not get added: since
# this is a left_join, we take the lefthand argument (df) as the
# base, and only add information from df2 when there is a match
# on the key column. If data in df2 do not match data in df,
# then those data are left out--so since df doesn't include an
# entry for "frog", we don't include the "frog" data from df2.

# We can also specify more than one column to join on.
df3 <- data.frame(animal = c("fish", "bird", "frog"),
                  bodyPart = c("fin", "wing", "leg"),
                  locomotion = c("swimming", "flying",
                                 "hopping and swimming"))

df %>%
  left_join(df3, by = c("animal", "locomotion"))
```

```
##   animal locomotion habitat bodyPart
## 1   fish   swimming   water      fin
## 2 rabbit    hopping    land     <NA>
## 3   bird     flying     air     wing
```

```
# And we can join on columns that match even if their names aren't
# the same. In that case, the syntax is by = c("lefthandColumnName",
# "righthandColumnName")
df4 <- data.frame(organism = c("fish", "bird", "frog"),
                  bodyPart = c("fin", "wing", "leg"))

df %>%
  left_join(df4, by = c("animal" = "organism"))
```

```
##   animal locomotion habitat bodyPart
## 1   fish   swimming   water      fin
## 2 rabbit    hopping    land     <NA>
## 3   bird     flying     air     wing
```

```
# "animal" is kept because it is from the lefthand data frame.
```

- **pull** lets you "pull out" a single column from a data frame as a vector.

```
df %>%
  pull(animal) # pull out the "animal" column as a vector
```

```
## [1] fish   rabbit bird
## Levels: bird fish rabbit
```

Connecting to the database

In order to access data from the database, you first have to establish a connection, which I will call "con" throughout this document (but you can call it anything—just like any other variable in R).

Here's how to do that.

```
con <- dbConnect(drv = RSQLite::SQLite(),
                 here("database", "currentDB", "ygdpDB.db"))
```

I'm not going to talk much about the mechanics of the `dbConnect()` function, but if you follow the syntax above, you will be fine. The first argument is the "driver", which is... something I don't really understand. But you specify it using the function `SQLite()` from the `RSQLite` package. You don't need to put anything in the parentheses.

The second argument is the path to the database file, which has the extension ".db". The path needs to be a full, not a relative, path (I can't quite figure out why, since most everything else in R can take relative paths), so I have used the `here` package to generate a full path.

Double check to make sure you have this file path correct. The connection won't work if you don't. For more information on how to use the `here` package, see this page.

Pulling data from the database

A relational database stores data in a separate file, which in this case has a .db extension.

When you pull data into R's memory from the database, there are basically two ways to go about it.

1. Load the *entire table* that you're going to work with into memory, and then do any filtering etc. in R.
2. Send a more complicated query to the database, and only retrieve the *final, filtered result* of that query.

As you may be able to guess, option 1 is simpler to understand, but it's not preferred. Especially for larger databases or when you're running a whole bunch of queries, this method is inefficient, since you're pulling in a lot more data than you need. If your table is large enough, you might not even be *able* to load in its entire contents—it might be too large to fit in R's memory.

The YGDP database is not very large. That means you can get away with method 1 in most cases. But the example queries below use method 2 (writing the full query before pulling in the data) as much as possible, so I hope you will default to that.

Method 1:

`dbTable` reads an entire database table into R. It takes two arguments, the connection object (in our case, `con`), and the name of the table to read. **Note:** the table name is not case sensitive, so even though the names of the YGDP database tables are in all caps, you can write them in lowercase in your code.

`dbTable` is a function that I wrote. It's based on the RSQLite function `dbReadTable`, but it has a few special bells and whistles. Most importantly, it reads any literal "NA" strings, or empty cells like "" and " ", as <NA>, which is how R represents missing values. You can feel free to use dbReadTable if you prefer, but be cautious then when working with NA's in your data.

Example: (note that I've commented out these lines to avoid printing out the entire table into this instructional document. Simply un-comment the lines in your own code to run them yourself).

```
# dbTable(con, "sentences")
# dbTable(con, "SENTENCES") # same thing.
```

Method 2:

dbListFields shows you just the column names of a table. It takes the same arguments as **dbTable**.

Example:

```
dbListFields(con, "sentences")
```

```
## [1] "sentenceID"     "sentenceText"   "sentenceType"   "constructionID"
## [5] "updateID"
```

```
#dbListFields(con, "SENTENCES") # same thing
```

dbListFields is a lot more efficient than names(dbTable(con, "sentences")), although you can do that too. But if you only need the names, why bother reading in the entire table?

tbl is good if you want to access the entire contents of the table but you don't want to pull it in yet. I will use **tbl** a lot in the coming examples. First, you use **tbl** to connect to a specific database table; then you apply filters etc. to narrow down your query; finally, you use **collect** to retrieve the results of the entire query at the end.

Example:

```
# Get only the ratings from Survey 11
tbl(con, "ratings") %>% # start with the RATINGS table
  filter(surveyID == "S11") %>% # apply your desired filters
  collect() # retrieve the data
```

```
## # A tibble: 36,485 x 6
##    responseID       sentenceID surveyID questionID rating updateID
##    <chr>            <chr>      <chr>    <chr>      <chr>  <chr>
##  1 R_2OYMtHqTw0A8a~ 1018.1     S11      S11_QID74  5      ratingsAddSurveyID.20~
##  2 R_2OYMtHqTw0A8a~ 1201       S11      S11_QID91  2      ratingsAddSurveyID.20~
##  3 R_2OYMtHqTw0A8a~ 1206       S11      S11_QID89  4      ratingsAddSurveyID.20~
##  4 R_2OYMtHqTw0A8a~ 1211       S11      S11_QID87  2      ratingsAddSurveyID.20~
##  5 R_2OYMtHqTw0A8a~ 1216       S11      S11_QID93  5      ratingsAddSurveyID.20~
##  6 R_2OYMtHqTw0A8a~ 1021.1     S11      S11_QID95  5      ratingsAddSurveyID.20~
##  7 R_2OYMtHqTw0A8a~ 1019.1     S11      S11_QID103 2      ratingsAddSurveyID.20~
##  8 R_2OYMtHqTw0A8a~ 1221       S11      S11_QID99  1      ratingsAddSurveyID.20~
##  9 R_2OYMtHqTw0A8a~ 1202       S11      S11_QID101 2      ratingsAddSurveyID.20~
## 10 R_2OYMtHqTw0A8a~ 1207       S11      S11_QID97  3      ratingsAddSurveyID.20~
## # ... with 36,475 more rows
```

Looking at your data

So, you've successfully queried some data (more examples in the next section). Now, how do you look at it to see if you got it right, or to see your results?

The most basic way to see an object in R is to print out the results by typing the name of the object.

```
df
```

```
##   animal locomotion habitat
## 1   fish   swimming   water
## 2 rabbit    hopping    land
## 3   bird     flying     air
```

However, your data might be really large, so that may or may not be a useful way to go about things.

You can use **head(data, n)** and **tail(data, n)** to see the first/last n rows of your data.[6]

```
head(df, 2) # first 2 rows
```

```
##   animal locomotion habitat
## 1   fish   swimming   water
## 2 rabbit    hopping    land
```

```
tail(df, 2)
```

```
##   animal locomotion habitat
## 2 rabbit    hopping    land
## 3   bird     flying     air
```

nrow(data) and **ncol(data)** give the number of rows and columns, respectively, of a data frame. **dim(data)** gives both **dim**ensions at the same time.

```
nrow(df) # number of rows
```

```
## [1] 3
```

```
ncol(df) # number of columns
```

```
## [1] 3
```

```
dim(df) # dimensions
```

```
## [1] 3 3
```

Example queries

What comments did people make about sentence 1207?

```
sentence1207Comments <- tbl(con, "comments") %>%
  filter(sentenceID == "1207") %>% # filter for target sentence
  collect() # send the query and retrieve the data
```

What comments did people make about Survey 11?

```
# start from the SURVEY_COMMENTS table
survey11Comments <- tbl(con, "survey_comments") %>%
  filter(surveyID == "S11") %>% # filter for target sentence
  collect() # send the query and retrieve the data
```

When was each survey conducted?

```
# start from the SURVEYS table
surveyDates <- tbl(con, "surveys") %>%
  select(surveyID, dateReleased, dateClosed) %>% # selects target cols
  collect() # retrieve the data
```

[6] If you leave the **n** argument blank and just do **head(data)**, the default is 6 rows.

What measurement scales were used on each question?

```
# start from the QUESTIONS table
questionScales <- tbl(con, "questions") %>%
  select(questionID, stimulusType, scaleOptions) %>% # select target cols
  collect() # retrieve the data
```

Show me all ratings from Survey 7.

```
# start from the RATINGS table
ratingsSurvey7 <- tbl(con, "ratings") %>%
  filter(surveyID == "S7") %>% # get the target survey
  collect() # retrieve data
```

Show me ratings from Survey 11, but only for participants in the South and West portions of the country.

This one is a little more complicated because we have to join to the DIALECT_REGIONS and DEMO_GEO tables.

The logic goes like this: start from RATINGS, and filter it to only Survey 11. Then, using `responseID` as the key (the common column), pull in some demographic information from DEMO_GEO for each participant. You could include whatever demographic information you want, but I'm going to focus only on the `raisedCityID` column, because that gets us each participant's location information. Then, using that cityID, we will join that to the DIALECT_REGIONS table.

I will show the code two ways: once with the different pieces separated out before the joins, so you can inspect each one, and once with everything all together in a single pipeline.

Method 1: define pieces of the three tables separately, then join.

```
ratingsSurvey11 <- tbl(con, "ratings") %>%
  filter(surveyID == "S11") %>%
  select(responseID, sentenceID, rating) %>% # (optional step) subset cols
  collect() # same as above. This is familiar.

allParticipantsRaisedCityID <- tbl(con, "demo_geo") %>%
  select(responseID, raisedCityID) %>%
  collect()
  # this shows, for each participant in our whole database
  # (not just survey 11), where they were raised.

citiesNSW <- tbl(con, "dialect_regions") %>%
  select(cityID, regionNSW) %>%
  collect()
  # for each cityID in the entire database,
  # which region (N, S, or W) does it fall into?

### now for the joins.
step1 <- ratingsSurvey11 %>%
  left_join(allParticipantsRaisedCityID, by = c("responseID"))

head(step1)
```

```
## # A tibble: 6 x 4
##   responseID        sentenceID rating raisedCityID
##   <chr>             <chr>      <chr>  <chr>
## 1 R_2OYMtHqTw0A8a3M 1018.1     5      C1000040
## 2 R_2OYMtHqTw0A8a3M 1201       2      C1000040
## 3 R_2OYMtHqTw0A8a3M 1206       4      C1000040
## 4 R_2OYMtHqTw0A8a3M 1211       2      C1000040
## 5 R_2OYMtHqTw0A8a3M 1216       5      C1000040
## 6 R_2OYMtHqTw0A8a3M 1021.1     5      C1000040
```

```
  # take a peek. Now each rating is associated with
  # that participant's raisedCityID.

step2 <- step1 %>%
  left_join(citiesNSW, by = c("raisedCityID" = "cityID"))
  # note the format of the "by" argument here.
  # This is for when you're joining two columns that have different
  # names. The format is by = c("[name in table 1]" =
  # "[name in table 2]"). More on this in the general
  # R syntax section.

# now we filter to get only the participants from the S and W.
step3 <- step2 %>%
  filter(regionNSW %in% c("The South", "The West"))
```

Method 2: do it all in one big pipeline

```
final <- tbl(con, "ratings") %>%
  filter(surveyID == "S11") %>%
  select(responseID, sentenceID, rating) %>%
  left_join(tbl(con, "demo_geo") %>%
              select(responseID, raisedCityID),
            by = "responseID") %>%
  left_join(tbl(con, "dialect_regions") %>%
              select(cityID, regionNSW),
            by = c("raisedCityID" = "cityID")) %>%
  collect()
```

Notice that methods 1 and 2 give you the same results: same number of rows and columns; same column headers

```
# Same dimensions (rows and columns)
dim(final) == dim(step2)
```

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

```
# Same column headers
names(final) == names(step2)
```

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

Show me ratings for Survey 6b, but only for participants who are 40 and older, female, and with either a bachelor's or graduate degree.

We will start from RATINGS and filter for Survey 6b, similar to above. Then we need to join in the information about `age`, `gender`, and `education`, which is stored in DEMO_GEO, so we'll join those columns from DEMO_GEO to RATINGS by `responseID`.

Again, I will show two methods: piece by piece or all together.

Method 1:

```
ratingsSurvey6b <- tbl(con, "ratings") %>%
  filter(surveyID == "S6b") %>% # only 6b
  select(responseID, sentenceID, rating) %>% # only target cols
  collect()

demographic <- tbl(con, "demo_geo") %>%
  select(responseID, age, gender, education) %>%
  collect()
```

```r
# Join these together
step1 <- ratingsSurvey6b %>%
  left_join(demographic, by = "responseID")

# Filter by our desired demographic limitations
step2 <- step1 %>%
  filter(age >= 40,
         gender == "female",
         education %in% c("bachelor's", "graduate"))
```

Method 2:

```r
final <- tbl(con, "ratings") %>%
  filter(surveyID == "S6b") %>%
  select(responseID, sentenceID, rating) %>%
  left_join(tbl(con, "demo_geo") %>%
              select(responseID, age, gender, education),
            by = "responseID") %>%
  filter(age >= 40,
         gender == "female",
         education %in% c("bachelor's", "graduate")) %>%
  collect()
```

Show me ratings for Sentence 1002, "Here's you a piece of pizza."

```r
# start from the RATINGS table
ratingsSentence1002 <- tbl(con, "ratings") %>%
  filter(sentenceID == "1002") %>% # only the target sentence
  collect() # retrieve data
```

Show me all ratings for sentences representing the Needs Washed construction, but only for participants who identify as male.

Method 1:

```r
ratings <- tbl(con, "ratings") %>%
  collect()
  # leaving all the other columns in case we want that info later.

# Take a look at the construction abbreviations
tbl(con, "sentences") %>%
  pull(constructionID) %>% # look at constructionID column only
  unique() %>% # get unique construction abbreviations
  sort() # sort unique values in alphabetical order
```

```
##  [1] "AC"   "AP"   "AtF"  "BD"   "BG"   "CG"   "CRel" "CRsg" "CU"   "CW"
## [11] "DAR"  "DMH"  "DMHS" "DPH"  "DPT"  "DPW"  "EB"   "ET"   "FT"   "FTI"
## [21] "HL"   "HYTA" "HYTD" "HYTG" "HYTH" "HYTN" "IN"   "LA"   "LT"   "MH"
## [31] "NC"   "NI"   "NWLi" "NWLo" "NWN"  "NWR"  "NWW"  "PC"   "PD"   "PsA"
## [41] "PtA"  "PtD"  "SDI"  "SS"   "SW"   "VRBI" "VRBP" "VREI" "VREP" "WI"
## [51] "WK"   "WN"   "YM"
```

```r
# Okay, the constructionID's corresponding to "Needs Washed" are
# NWLi, NWLo, NWN, NWR, and NWW. I just happened to know that;
# if you don't already know the constructionID's you want,
# you can look in the CONSTRUCTIONS table.
```

```
# Which sentences belong to the needs washed construction?
needsWashedSentenceIDs <- tbl(con, "sentences") %>%
  filter(constructionID %in% c("NWLi", "NWLo", "NWN", "NWR", "NWW")) %>%
  pull(sentenceID) %>% # pull out the sentenceID column
  unique()
  # get only the unique sentenceID's that are associated
  # with Needs Washed, so we can use them to filter.

# get participant gender identities
participantGender <- tbl(con, "demo_geo") %>%
  select(responseID, gender) %>%
  collect()

# Join genders to ratings by responseID
step1 <- ratings %>%
  left_join(participantGender, by = "responseID")

# Filter down to only Needs Washed sentences
step2 <- step1 %>%
  filter(sentenceID %in% needsWashedSentenceIDs)

# Filter down to only male participants
step3 <- step2 %>%
  filter(gender == "male")
```

Method 2:

```
# Start with RATINGS table
final <- tbl(con, "ratings") %>%
  left_join(tbl(con, "sentences") %>% # join constructionIDs
              select(sentenceID, constructionID),
            by = "sentenceID") %>%
  left_join(tbl(con, "demo_geo") %>% # join gender identities
              select(responseID, gender),
            by = "responseID") %>%
  filter(gender == "male",
         constructionID %in% c("NWLi", "NWLo", "NWN", "NWR", "NWW")) %>%
  collect()
```

How many different Massachusetts cities are represented as childhood residences for our survey participants?

We will start by getting a list of all the unique childhood cities that appear in our database. Then we'll use that to filter CITIES down to only childhood cities. Then we will filter it down further to only Massachusetts cities, and then we'll count the number of rows to get the number of cities.

```
# list of cityID's where at least one person grew up.
childhoodCityIDs <- tbl(con, "demo_geo") %>%
  pull(raisedCityID) %>% # get the childhood cities only
  unique() # only unique cities; avoid repeats.

# Get MA childhood cities
MAChildhoodCities <- tbl(con, "cities") %>% # start with CITIES
  filter(cityID %in% childhoodCityIDs) %>% # limit to childhood cities
  filter(stateID == "MA") %>% # limit to MA
  collect()

# How many MA childhood cities do we have?
nrow(MAChildhoodCities) #99
```

```
## [1] 99
```

```
# Check out a random sample of 10 of them
MAChildhoodCities %>%
  sample_n(10)
```

```
## # A tibble: 10 x 10
##    cityID cityName stateID stateName countyName countryID lat   long  postalCode
##    <chr>  <chr>    <chr>   <chr>     <chr>      <chr>     <chr> <chr> <chr>
##  1 C1004~ Chelsea  MA      Massachu~ Suffolk    USA       42.3~ -71.~ 02150
##  2 C1002~ Lynn     MA      Massachu~ Essex      USA       42.4~ -70.~ 01901
##  3 C1002~ New Bed~ MA      Massachu~ Bristol    USA       41.6~ -70.~ 02740
##  4 C1006~ Monson   MA      Massachu~ Hampden    USA       42.0~ -72.~ 01057
##  5 C1002~ Shelbur~ MA      Massachu~ Franklin   USA       42.6~ -72.~ 01370
##  6 C1002~ Millbury MA      Massachu~ Worcester  USA       42.1~ -71.~ 01527
##  7 C1002~ Fairhav~ MA      Massachu~ Bristol    USA       41.6~ -70.~ 02719
##  8 C1002~ Waterto~ MA      Massachu~ Middlesex  USA       42.3~ -71.~ 02472
##  9 C1007~ Centerv~ MA      Massachu~ Barnstable USA       41.6~ -70.~ 02632
## 10 C1002~ Mansfie~ MA      Massachu~ Bristol    USA       42.0~ -71.~ 02048
## # ... with 1 more variable: updateID <chr>
```

Which sentences were tested on Survey 12?

```
# start with SURVEY_SENTENCES, the join table between
# surveyID's and sentenceID's
survey12SentenceIDs <- tbl(con, "survey_sentences") %>%
  filter(surveyID == "S12") %>% # only survey 12 sentences
  pull(sentenceID)
  # grab only the sentenceID column so we get a vector
  # instead of a data frame.

# If we want to also see the text of those sentences, we can
# now use those sentenceID's to filter the SENTENCES table
survey12Sentences <- tbl(con, "sentences") %>%
  filter(sentenceID %in% survey12SentenceIDs) %>%
  pull(sentenceText)
```

What were the demographic characteristics of New Haven County, Connecticut on the 2010 census?

This is a bit tricky because we actually don't have a table where each row is one county. Decided it would be easier to extract the individual cityID's (lat/long coordinates) over the ESRI counties shapefile directly, and store the results. So each row in CENSUS_COUNTY_DEMO is the *county-level* information *for each city*. This is not ideal database format, but it's what we've got, and it's fine.

This means that to look at New Haven County, we could filter CENSUS_COUNTY_DEMO by countyName, but then that would give us a bunch of repeat information for New Haven County, since there are many `cityID`s that fall into New Haven County. So we end up only taking one row.

```
newHavenCountyInfo <- tbl(con, "census_county_demo") %>%
  filter(countyName == "New Haven") %>% # filter to new haven county
  select(-c(cityID, updateID)) %>% # remove any cols that pertain to
                # individual city rows, instead of the whole county.
  distinct()
```

When was the database last updated?

```
lastUpdateDate <- tbl(con, "version_history") %>%
  pull(date) %>% # pull out the date column
```

```
  last() # get the last (most recent) date

# Note that the above solution assumes that the last date
# is the most recent. This SHOULLD be the case, but what
# if the rows got scrambled somehow? If you actually want to
# make sure you're getting the most recent date, you could do this:
tbl(con, "version_history") %>%
  pull(date) %>% # pull out the update dates
  lubridate::ymd() %>% # convert to date format using `ymd()`
                  # in the package `lubridate`
  max() # get the maximum (most recent) date.
```

```
## [1] "2020-10-09"
```

What do the variables in DEMO_GEO mean and where do they come from?

```
demoGeoVars <- tbl(con, "variables") %>%
  filter(tableName == "DEMO_GEO") # filter down to only variables from DEMO_GEO
```

Who last updated CITIES and what changes did they make?

First, we will get a list of all the `updateID`'s in CITIES. Then we'll pull up the corresponding entries in UPDATE_METADATA to get the information associated with those `updateID`'s, including their dates. We will determine which one has the most recent date, and then we'll view the information for that most recent update.

```
# Get a list of unique updateID's in CITIES
citiesUpdateIDs <- tbl(con, "cities") %>%
  pull(updateID) %>% # pull the updateID's
  unique() # only unique entries

# Pull info for the most recent of these updateID's
mostRecentUpdate <- tbl(con, "update_metadata") %>%
  filter(updateID %in% citiesUpdateIDs) %>%
  collect() %>% # retrieve the data
  mutate(date = lubridate::ymd(date)) %>% # convert to date format
  filter(date == max(date)) # get the row for the max date only

# By looking at mostRecentUpdate, we can see who made the
# update and what they did.
mostRecentUpdate
```

```
## # A tibble: 1 x 6
##   updateID       date       updater    metadata       versionNumber sourceCode
##   <chr>          <date>     <chr>      <chr>          <chr>         <chr>
## 1 S8Add.20200910 2020-09-24 Kaija Gahm Added survey 8 1.7.0         S8_Add.Rmd
```

How many people took more than one survey?

```
multipleSurveys <- tbl(con, "responses") %>%
  filter(amazonID != "") %>% # remove blank amazonID's
  group_by(amazonID) %>% # group data by amazonID
  filter(n() > 1) %>% # limit to amazonID's with more than one responseID
  arrange(amazonID) %>% # sort by amazonID
  collect() # retrieve the data

# Look at the multiple-survey people
multipleSurveys
```

```
## # A tibble: 4,073 x 4
## # Groups:   amazonID [1,330]
##    responseID        amazonID                surveyID updateID
##    <chr>             <chr>                   <chr>    <chr>
##  1 R_5tKVw0M9ALNgjwV " A1X388C6SDCDEY"       S9       S9Add.20200904
##  2 R_aXdzdOHTVXcwlFj " A1X388C6SDCDEY"       S5b      S5bAdd.20200906
##  3 R_6XzyGxXIheh43oV " A1X388C6SDCDEY"       S6b      S6bAdd.20200907
##  4 R_af3oeJIbEZN2Tw9 " A32YLB6PX6QL71"       S6b      S6bAdd.20200907
##  5 R_eF1hqifdzy8MR6Z " A32YLB6PX6QL71"       S6b      S6bAdd.20200907
##  6 R_eYdR7tGfD0lGZSZ " A3CB5YPZ1UAHVV"       S6b      S6bAdd.20200907
##  7 R_10wQVHF1MKxeleL " A3CB5YPZ1UAHVV"       S8       S8Add.20200910
##  8 R_1IF6Vo2Whl3NyaV "A002160837SWJFPIAI7L7" S12      original.2020
##  9 R_0qgdVn1wd6YNmCN "A002160837SWJFPIAI7L7" S9       S9Add.20200904
## 10 R_1r9bT0bYQnqEuLY "A100UJALBX4U9V"        S12      original.2020
## # ... with 4,063 more rows
```

```
# How many people took multiple surveys?
length(unique(multipleSurveys$amazonID)) # Wow, that's a lot!
```

```
## [1] 1330
```

Saving your data

If you're not an R addict like me, you might want to export your data from R so that you can work with it in GIS, Excel, or some other format. Here's an example of how to export your data as a csv.

```
# Perform a query (any query will do!)
sampleData <- tbl(con, "ratings") %>%
  filter(sentenceID %in% c("1002", "1007")) %>%
  select(responseID, sentenceID, rating) %>%
  collect()

# save 'sampleData' as a csv
write.csv(sampleData, file = here("sampleData.csv"), row.names = F) # this line will save sampleData.csv to the

# The row.names = F argument specifies that you don't want a separate column in your exported data containing ro

# You can specify additional arguments to here() to save your file to a different directory. Or if you want to u
```

Disconnecting from the database

Remember to close the connection to the database once you're done pulling data by calling `dbDisconnect(con)`.

```
dbDisconnect(con)
```

Additional resources

Here are some additional resources I've compiled if you'd like to learn more about databases, R, or working with databases in R.

R

- You can install R here.
- You'll almost certainly want to run R through RStudio, its IDE ("interactive development environment"). RStudio provides lots of great support that makes working in R much easier and more enjoyable. You can install RStudio here–you definitely want the free version. The paid version is designed for big enterprises; unless you get really advanced, there's no reason you should ever have to pay.

- The best resource for getting started with R is, in my opinion, the book *R For Data Science*. It's a free online book, and you can access it here.
- Another great webinar is A Gentle Introduction to Tidy Statistics in R.
- Here are some other resources for beginners in R. I particularly like RStudio's online interactive tutorials.
- Here's another overview with some good links to intro R resources.

Understanding relational databases

- Here is a tutorial on relational databases (it focuses on database construction, which may or may not be relevant to you).
- Database normalization, explained.
- More on normalization, including normal forms.

Databases in R

- The R For Data Science book gives a pretty good introduction to working with relational data in R.
- Here is a tutorial on accessing databases from R, from The Carpentries.

Saving data from R to different formats

- Exporting data from R to several different file formats
- And another one that focuses specifically on csv/txt formats.