Managing Data with SQL

An Introduction

Programming Practices for Research in Economics

Economics@Zurich

Fall 2017

Where have we been?

- Learned how to write modular code in Python and R
- Read in data to Python or R and store *in memory*
  - What to do when data is bigger than memory?
  - One solution: use a database management system

Why use a Database Management System?

- Cope with **LARGE** and complex data sets.
  - Keeps data stored on disk
- Provides very flexible access to the data set at good speed.
- Provides efficient data storage.
- Standard languages for controlling a DBMS.

What can we do with data stored in a DBMS?

- Our focus: basic data operations
  - select subsets of the data (rows and columns)
  - group subsets of data
  - do math and other calculations
  - combine data across spreadsheets
  - input new data
- Covers *most* of what a typical economist might do with a database
  - But only scratches the surface of what you can do with with a database

Types of databases

- Different types of database:
  - Hierarchical and Network databases.
  - Relational databases.
  - Object-Oriented databases and NoSQL databases.
- We will only deal with relational databases
  - Different vocabs, similar ideas

What's a Relational Database?

- A relational database stores data in relations made up of records with fields.
- The relations are usually represented as tables; each record is usually shown as a row, and the fields as columns.
- In most cases, each record will have a unique identifier, called a key, which is stored as one of its fields.
- Records may also contain keys that refer to records in other tables, which enables us to combine information from two or more sources.

What is SQL?

- SQL is a `query language` that is easy to type into a computer
  - Think of it as an human readable translation of a 'research question' to go and get the data needed to answer it
- Syntax rests on 'relational algebra' or relational calculus to generate a query
  - We are going to skip this theory heavy stuff

SQL and Relational databases

- We write queries to extract data using the `SQL` query language
- The query is sent to a relational database through some program
  - We use `SQLite`
- We get data as output

Roadmap

1. Open a relational database using SQLite
2. Use SQL syntax to query the database and return records using SQLite
3. Plug-in a SQLite database to R and Python to execute queries

Opening a Relational Database

Getting Started

- We are going to work with some fictious auctions data
- Change into that directory

```
$ cd examples/auctions
```

- Take a look around:

```
$ ls -lR
```

- We are going to work with the database `database/auctions_data.db`

Working interactively with SQLite

- Load the data into SQLite

```
$ sqlite3 database/auctions_data.db
```

- Expected output:

```
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite>
```

- Can exit SQLite with the command `.exit`

```
sqlite> .exit
```

Understanding Database Structure

- SQLite has `.dot` commands which offer useful functionality to understand the structure of the database:
  - `.dot` commands are SQLite specific
  - See `dotCommands.pdf` on our Cheatsheet pagefor a list of all available
  - Other database engines have similar commands
- `.tables` returns the different tables in a database:

```
sqlite> .tables
auctions  bidders   bids
```

Understanding Database Structure

- `.schema` returns the structure of each table

```
sqlite> .schema
CREATE TABLE auctions
(
      AuctionID        INTEGER
    , Volume           INTEGER NOT NULL
    , District         INTEGER NOT NULL
    , Date             TEXT NOT NULL
    , PRIMARY KEY (AuctionID)
);
<...>
```

- Possible data-types are summarized in `sql-data-types.pdf`

Basic Queries

Selecting a Column of Data

- To select data, we write a query that `SELECT`s columns `FROM` a table within our data base

```
SELECT
AuctionID
FROM
auctions
;
```

- Notes:
  - SQL commands are case insensitive
  - Structure doesn't matter to SQL, matters for readability

Selecting Multiple Columns of Data

```
SELECT
      AuctionID
    , Volume
    , District
FROM
    auctions
;
```

- Note `SELECT * FROM tblName;` selects *all* columns, but we don't generally recommend using it. (Why?)

Improving Output Readability

- The output from our query looks quite messy
  - which column is which?
  - not aligned
- Can be tidied with the commands

```
.mode column
.header on
```

Limiting Output

- Sometimes we don't want all output streamed to us
  - Particularly of there are '000s of returned
- `LIMIT n` at the end of our query limits results returned

```
SELECT
  AuctionID
, Volume
, District
FROM
auctions
LIMIT
5
;
```

Selecting Unique Values

- Sometimes we are interested in the unique values of a column or set of columns:

```sql
SELECT
District
FROM
auctions
;
```

Your Turn

Challenge: `SELECT`-ing data

- Write a query that returns the first and last names of all registered bidders in our data set
- Write a query that returns the first and last names of 5 registered bidders in our data set

Moving away from interactive queries

- So far we have been working interactively with SQLite
  - This is not how we want to work usually (Why?)
- We can write queries in a text file, and run SQLite from the command line
  - Files containing queries typically end with '.sql'
- We can also export the results to a file
  - We will export to '.csv'

Structuring a Query in a File I

```
/*
    Comments in a header
*/;

.mode csv
.headers on
.output ./out/01-selectData.csv

SELECT
      varName1      -- In line comment
    , varName2

FROM
    tblName
;
.output stdout      -- return to default
```

Executing a SQL Query from a File

- Above query run from inside SQLite:

```
sqlite> .read queries/01-select.sql
```

Structuring a Query in a File I

```sql
/*
    Description

    Author: @lachlandeer
*/;

SELECT
    <...>
FROM
    <...>
;
```

Executing SQLite from Command Line

- From command line to get csv output

```
sqlite3 -batch -header -csv some_database.db \
< some_query.sql > some_csv_file.csv
```

Filtering Data

- We can also filter data - selecting only the data meeting certain criteria, using the `WHERE` command:

```
SELECT
      varName1
    , varName2

FROM
    tblName
WHERE
    someCondition
;
```

Filtering Data: Example

- Let's get all auctions that took place in District 4:

```
SELECT
      AuctionID
    , Date
    , District

FROM
    auctions
WHERE
    District == 4
;
```

Filtering Data: Example II

- Let's get all auctions that took place in District 4:

```
SELECT
      AuctionID
    , Date
    , District

FROM
    auctions
WHERE
    (District == 4) AND (Date == '20120604')
;
```

Filtering Data: Example III

- Let's get all auctions that took place in District 4 or 7:

```
SELECT
      AuctionID
    , Date
    , District

FROM
    auctions
WHERE
    District IN (4, 7)
;
```

Ordering Results

- We can also sort the results of our queries by using `ORDER BY`

```sql
SELECT
      varName1
    , varName2

FROM
    tblName
WHERE
    someCondition
ORDER BY
    varName1 ASC    -- or DESC for descending
;
```

Ordering Results: Example

- Let's get all auctions that took place in District 4 or 7, ordered by District:

```
SELECT
      AuctionID
    , Date
    , District

FROM
    auctions
WHERE
    District IN (4, 7)
ORDER BY
    District ASC
;
```

Your Turn

Challenge: Filtering and Ordering Data

1. Write a query that selects all bids for bidder 1
2. Write a query that selects all bids for bidder 1 and 4, ordered by bidder
3. Write a query that selects all bids for bidder 1 and 4, in auctions with an even AuctionID
4. Write a query that selects all bids for bidder 1 and 4, in auctions with an even AuctionID, ordered by AuctionID and bidderID

Order of Operations

- What order does SQL execute our query?
  - Important for understanding and debugging
- For what we have so far...

  1. FROM
  2. WHERE
  3. SELECT
  4. DISTINCT
  5. ORDER BY

Aggregations in SQL

Aggregate Functions and Group Statistics

- Often we want more than individual rows filtered by conditions
  - Might want summary information for certain slices of data
- Can be done using 'Aggregate Functions' and the `GROUP BY` command
  - See `sqlFunctions.pdf` for a useful list of aggregate functions availble in SQL

Query Format

```
SELECT
      FUNC(varName1)
    , varName2
FROM
    tblName
GROUP BY
      groupVar1
    , groupVar2
;
```

The COUNT function

- Let's Count the number of bids placed by each BidderID

```
SELECT
      COUNT(*)
    , bidderID
FROM
    bids
GROUP BY
    bidderID
```

Aliases in SQL

- Notice that the output of our counting exercise yields pretty ugly variable names for an aggregate function.
- SQL provides the alias function `AS` to name a variable

```
SELECT
      COUNT(*)       AS nBids
    , bidderID
FROM
    bids
GROUP BY
    bidderID
```

- Can also alias non-aggregate columns

Your Turn

Challenge: Aggregate Functions

1. Write a query that returns the minimum, maximum and average bid for each bidder
2. Write a query that returns the minimum, maximum and average bid in each Auction

Rounding Values

- We might want numbers rounded to a certain number of decimal places
  - The function `ROUND()`

```
SELECT
      AuctionID
    , ROUND(AVG(Bid),2)  AS AverageBid
FROM
    bids
GROUP BY
    AuctionID
;
```

Filtering Based on Aggregate Statistics

- the keywords `WHERE` allowed to filter the rows according to some criteria.
  - SQL offers a mechanism to filter based on aggregate functions, through the `HAVING` keyword.

Filtering with Having

```sql
SELECT
      FUNC(varName1)
    , varName2
FROM
    tblName
GROUP BY
      groupVar1
    , groupVar2
HAVING
    condition1
;
```

Your Turn

Challenge: Filtering Aggregate Functions with `Having`

1. Write a query that returns the minimum, maximum and average bid for each bidder for bidders who's average big is greater than 11.0
2. Write a query that returns the minimum, maximum and average bid for each bidder for bidders who's average big is greater than 11.0, and have an even bidder ID
3. Write a query that returns the minimum, maximum and average bid in each Auction where the largest bid is smaller than 13.0, and the smallest bid is at least 8.0

Order of Operations Redux

- For what we have so far...
  1. FROM
  2. WHERE
  3. GROUP BY
  4. HAVING
  5. SELECT
  6. DISTINCT
  7. ORDER BY

Joining Data

Joins

- To combine data from two tables we use the SQL `JOIN` command
  - Comes after the `FROM` command
- Need to tell the computer which columns provide the link between the tables using the word `ON`
- Also need to specify from which `table` a column belongs
  - Replace `colName` with `tblName.colName`

Query Format

```
SELECT
      tblName1.varName1
    , <...>
    , tblName2.varName1
    , <...>
FROM
    tblName1
JOIN
    tblName2
ON
    tblName1.identifer = tblName2.identifer
;
```

Example: Linking Auction Info to Bidding Data

- Query that links volume and district information from the auction table to the bids table

```
SELECT
      auctions.AuctionID  AS AuctionID
    , auctions.Volume     AS Volume
    , auctions.District   AS District
    , bids.bidderID       AS bidderID
    , bids.bid            AS bid
FROM
    auctions
INNER JOIN
    bids
ON
    auctions.AuctionID = bids.Auctions.ID
;
```

Your Turn

Challenge: `INNER JOINs`

1. Write a query that links bids to the bidders first and last names
2. Write a query that returns the minimum, maximum and average bid for each bidder linked to their first and last names

Complicated Joins and Sub-Queries

- When query involves a step of data-wrangling during the `JOIN` process we want to use 'sub-queries'
- Example: Compute the first and last bid date for each bidder
  - Data wrangling step: compute on the date string
- Approach:
  1. Write a subquery that formats the date string into a SQL date format for each auction, then `SELECT`s the new date variable
  2. `JOIN` that information from the bidders table whilst computing aggregate statistics.
- Let's do that together, in `08-joins-with-subquery.sql`

Order of Operations Redux

- For what we have so far...
  1. FROM
  2. ON
  3. (INNER) JOIN
  4. WHERE
  5. GROUP BY
  6. HAVING
  7. SELECT
  8. DISTINCT
  9. ORDER BY

Additional Topics (optional)

Data Hygiene

- Tables in SQL should obey some formatting rules:
  1. Eliminate Duplicative Columns
  2. Each table must have a column that has a unique value for every row; this column is the **primary key** for the table.
  3. Every column in a table should relate to the primary key
- These rules are collectively knows as the *Second normal form* of data
  - There are others - but we don't want to bore on details
  - Think of this as a minimum standard your data should comply to
  - Useful outside of SQL too

Data Hygiene: Eliminate duplicative columns

- A table column must have only one “piece” of information in it.
- This table has a column with the several pieces of information in it:

Father	Children
Paul	Dominic, Matthew, Christina
Simon	Joshua
Ray	David, Alex

Figure 1:

Data Hygiene: Eliminate duplicative columns

- A table column must have only one "piece" of information in it.
- This table has multiple columns with the same sort of information:

Father	Child	Child	Child
Paul	Dominic	Matthew	Christina
Simon	Joshua		
Ray	David	Alex	

Figure 2:

Data Hygiene: Eliminate duplicative columns

- A table column must have only one "piece" of information in it.
- This table has no duplicative columns.

Father	Child
Paul	Dominic
Paul	Matthew
Paul	Christina
Simon	Joshua
Ray	David
Ray	Alex

Figure 3:

Data Hygiene: Primary Key

- In the previous table, could use the Child column, but if I add a row with an existing name the table is broken.

Father	Child
Paul	Dominic
Paul	Matthew
Paul	Christina
Simon	Joshua
Ray	David
Ray	Alex
Brian	David

Figure 4:

Data Hygiene: Primary Key

- This table now conforms to 1NF.

Father	Child Name	Child ID
Paul	Dominic	1
Paul	Matthew	2
Paul	Christina	3
Simon	Joshua	4
Ray	David	5
Ray	Alex	6
Brian	David	7

Figure 5:

Data Hygiene: No Redundant Information

- This table has columns which do not relate to the primary key

Father	Father Country	Child Name	**Child ID**
Paul	NZ	Dominic	1
Paul	NZ	Matthew	2
Paul	NZ	Christina	3
Simon	UK	Joshua	4
Ray	NZ	David	5
Ray	NZ	Alex	6
Brian	UK	David	7

Figure 6:

Data Hygiene: No Redundant Information

- The solution is to place information which does not relate to the primary key in a separate table

Father ID	Child Name	**Child ID**
1	Dominic	1
1	Matthew	2
1	Christina	3
...	...	...

Father	Country	**Father ID**
Paul	NZ	1
...	...	...

Figure 7:

Data Hygiene: No Redundant Information

- The new table has its own primary key and the new table is related to the original table using a foreign key (the Father ID column).

Father ID	Child Name	**Child ID**
1	Dominic	1
1	Matthew	2
1	Christina	3
...	...	...

Father	Country	**Father ID**
Paul	NZ	1
...	...	...

Figure 8:

Creating New Tables

- So far looked at how to get information out of a database,
  - More frequent than adding information. I
- If we want to create and modify data, we need to know two other sets of commands.
  - `CREATE TABLE` will create a new table
  - `DROP TABLE` will delete an existing table

CREATE TABLE

- Let's create a replica of our auctions database, auctions_data2.db
- Open a new, empty database

```
$ sqlite3 ./database/auctions_data2.db
```

- Create a table using the following syntax:

```
CREATE TABLE tblName(
      varName       Type
    , ...
    , PRIMARY KEY (varName)
)
;
```

- **Always** impose a fixed variable type

CREATE TABLE

- Create the auctions table:

```
CREATE TABLE auctions
(
      AuctionID        INTEGER
    , Volume           INTEGER NOT NULL
    , District         INTEGER NOT NULL
    , Date             TEXT NOT NULL
    , PRIMARY KEY (AuctionID)
)
;
```

- Notice that no data is stored in the table, we have only imposed structure

Your Turn

Challenge: Creating Tables

1. Create the table `bids`
2. Create the table `bidders`

Inserting Data

One can manually insert rows of data

```
INSERT INTO auctions
    values(1, 4567, 3, '2017-08-13')
;
INSERT INTO auctions
    values(2, 2000, 7, '2017-07-01')
;
```

Updating values

- If we made a mistake when entering values of the last INSERT statement

```
UPDATE Site
SET
      Volume = 2500
    , Date   = `2013-08-01`
WHERE
    AuctionID = 1
;
```

Deleting Records

- Deleting is tricker than inserting
  - We have to ensure that the database remains internally consistent
- If all we care about is a single table, we can use the `DELETE` command with a `WHERE` clause that matches the records we want to discard

```
DELETE FROM
    auctions
WHERE
    Date = '2013-08-01'
;
```

Deleting Records - Referential Integrity

- What if we removed a record that references a foreign key?
  - Most databases will stop you from deleting a record if it references a valid foreign key
- Problem is called referential integrity
- Need to ensure that all references between tables can always be resolved correctly.
  - If our database manager supports it, can use cascading delete
  - Outside the scope of our module

DROP TABLE

- Sometimes we want to remove an entire table
  - `DROP TABLE` allows us to do that

```
DROP TABLE
    auctions
;
```

- Your task: recreate an empty auctions table

Importing Data

- Rarely manually insert data, usually we import from a file
  - SQLite has the command `.import` to import data
- Let's import the auctions table data from a csv file

```
.separator ,
.import raw-data/AuctionsTable.csv auctions
```

Your Turn

Challenge: Import data

1. Import the bidders data from `raw-data/BiddersTable.csv`
2. Import the bids data from raw-data/BidsTable.csv

Using SQLite with R

Databases and R

- R can connect to almost any existing database type
- The `dplyr` package you learned previously, in conjunction with `dbplyr` supports connecting to the widely-used open source databases
- Interfacing with databases using `dplyr` focuses on retrieving and analyzing datasets by generating `SELECT` -style SQL statements,
- It doesn't modify the database itself.
  - `dplyr` does not offer functions to UPDATE or DELETE entries
  - Will need to use additional R packages (e.g., `RSQLite`)

Getting Started

- Loading packages:

```r
library(dbplyr)
library(dplyr)
```

- Connect to database

```r
auctions_db <- src_sqlite("../database/
            auctions_data.db")
```

- Note: self contained in a Rmd file:
  - `examples/auctions/r-and-sql/databases-in-R.rmd`

Queries with SQL Syntax

- To connect to tables within a database, you can use the tbl() function from dplyr.
  - This function can be used to send SQL queries to the database.
- To demonstrate this functionality, let's select the columns "AuctionsID", "Volume", and "District" from the surveys table:

```
tbl(auctions_db,
    sql("SELECT AuctionID, Volume,
           District FROM auctions"))
```

Queries with `dplyr` Syntax

- The same operation can be done using dplyr's verbs.
  - First, select the table on which to do the operations by creating the auctions object
  - Then we use the standard `dplyr` syntax as if it were a data frame:

```
auctions <- tbl(auctions_db, "auctions")

auctions %>%
    select(AuctionID, Volume, District)
```

Using R Functions on a database

- Several functions that can be used with data frames can also be used on tables from a database.

```
bids <- tbl(auctions_db, "bids")

head(bids, n=10)
```

Using R Functions on a database II

- Some functions don't work quite as expected.
- For instance, let's check how many rows there are in total:

```
nrow(bids)

## [1] NA
```

- The first line of the head() output included ?? indicating that the number of rows wasn't known.
  - Because unlike `read.csv()` data not in memory

Under the hood SQL Translations

- Behind the scenes, dplyr:
  1. translates your R code into SQL
  2. submits it to the database
  3. translates the database's response into an R data frame
- dplyr's `show_query()` function reveals which SQL commands are actually sent to the database:

```
show_query(head(bids, n=10))

## <SQL>
## SELECT *
## FROM `bids`
## LIMIT 10
```

Example: Simple Database Query

- Let's reproduce one of the queries we wrote directly using SQL syntax earlier in the module.
- We will select all bids from bidders 1 and 4, and keep only the bid, bidderID and auctionID:

```
bids %>%
    filter(bidderID %in%  c(1,4)) %>%
    select(Bid, BidderID, AuctionID) %>%
    arrange(BidderID, AuctionID)
```

Example: Simple Database Query II

- Or we can filter bids for bidders that have a BidderID that's and even number:

```
bids %>%
    filter(bidderID %% 2 == 0) %>%
    select(Bid, BidderID, AuctionID) %>%
    arrange(BidderID, AuctionID)
```

- Why are only 10 rows returned?

`dplyr`'s Lazy Evaluation:

Hadley Wickham, the author of `dplyr` explains:

> *When working with databases, `dplyr` tries to be as lazy as possible:*
>
> - *It never pulls data into `R` unless you explicitly ask for it.*
> - *It delays doing any work until the last possible moment - it collects together everything you want to do and then sends it to the database in one step.*

Returning all results

- Append the `collect()` function to your dplyr code

```
subset_bids <- bids %>%
                filter(bidderID %% 2 == 0) %>%
                select(Bid, BidderID, AuctionID) %>%
                arrange(BidderID, AuctionID) %>%
                collect()
```

- The result is a `data.frame`
  - Can continue to work with the data in R, without communicating with the database.

SQL Joins with dplyr

- Can use the dplyr inner_join function to combine data from two tables.
- Let's again collect the minimum, maximum and average bid for each bidder, and link this to the bidder's name:

```
bidders <- tbl(auctions_db, "bidders")

bids %>%
    group_by(BidderID) %>%
    summarise(smallestBid = min(bid,  na.rm = TRUE),
              averageBid  = mean(bid, na.rm = TRUE),
              largestBid  = max(bid,  na.rm = TRUE)
              ) %>%
    inner_join(bidders) %>%
    select(FirstName, LastName, smallestBid,
            averageBid, largestBid) %>%
    collect()
```

Using SQLite with `Python`

Databases and `Python`

- `Python` can connect to almost any existing database type
- Unlike `R`, there is generally a different library to interface with each database type
  - Our focus: `sqlite3`
- `sqlite3` provides functionality to:
  1. Make `SELECT`-style queries
  2. Modifying database rows
  3. Setting up dynamic queries
  4. Creating and altering tables
- These slides only focus on (1)
- All code in a Jupyter notebook
  - `examples/auctions/sql-and-python.ipynb`

Importing Packages and Connecting to Database

- Load `sqlite3` to interact with database, `pandas` to store results

```
import sqlite3
import pandas as pd
```

- Connect to database using `sqlite3.connect()`

```
connection = sqlite3.connect(
                  '../database/auctions_data.db')
```

- Create cursor object to to send SQL queries:

```
cursor = connection.cursor()
```

List Tables in the Database

- No direct function to do this
  - Table Names lie in a ‘master table’
- Run a query using `.execute()`
- Retrieve results with `.fetchall()`

```
cursor.execute("SELECT name FROM sqlite_master
                WHERE type='table';")
print(cursor.fetchall())
```

Close Connections

- After finished querying - close connection to database to avoid it freezing

```
cursor.close()
connection.close()
```

Simple Queries

- Let's replicate an earlier example where we select AuctionID, Volume, District from the auctions table:

```
connection = sqlite3.connect(
                '../database/auctions_data.db'
                )
cursor = connection.cursor()

cursor.execute("SELECT AuctionID, Volume,
                    District FROM auctions;")
results = cursor.fetchall()
print(results)
```

- Output is a list of tuples

Read SQL Results into `DataFrame`

- Pandas has an inbuilt function read_sql_query that reads the results of a SQL query straight into a DataFrame.
- There are several advantages of this:
  1. Avoids the need to create a cursor object, and ‘fetch’ results at the end with `fetch_all`
  2. Pandas directly reads in column names from the SQL table headers
  3. The output is a DataFrame which we have already learned how to work with

Read SQL Results into `DataFrame`: Example

- Let's take the same query as above, but this time send the results into a pandas DataFrame

```
connection = sqlite3.connect(
                '../database/auctions_data.db'
                )

data = pd.read_sql_query(
                "SELECT AuctionID, Volume,
                    District FROM auctions;",
                connection
            )
```

- Pandas closes the connection for us when query complete

Writing Increasingly Complex Queries

- Query string inline is a little messy
  - Separate the query into its own variable
  - Pass that variable across to the `read_sql_query()` command

Writing Increasingly Complex Queries II

```
request = """
        SELECT
              bidders.FirstName
            , bidders.LastName
            , MIN(bids.Bid)                AS SmallestBid
            , ROUND(AVG(bids.Bid),2)       AS AverageBid
            , MAX(bids.Bid)                AS LargestBid
        FROM
            bids
        INNER JOIN
            bidders
        ON
            bidders.BidderID = bids.BidderID
        GROUP BY
              bids.BidderID
        ;
        """
```

Writing Increasingly Complex Queries III

- Return results:

```
data2 = pd.read_sql_query(request, connection)
data2.head()
```

- Returns output:

```
    FirstName LastName   SmallestBid AverageBid   LargestBid
0   Adam      Cooper     10.86       13.53        19.21
1   Bryan     Dykstra    8.81        11.38        13.09
2   Charles   Elan       7.39        10.58        15.62
3   David     Forester   7.93        12.27        15.67
4   Edward    Gulden     7.35        10.09        14.28
```

Conclusion

Why Databases? Why SQL?

- Sometimes we work with data that won't fit into memory
  - Don't 'buy more RAM' to plug and chug in STATA/R/Python
- Databases offer a way to access and efficiently compute with large data stored on disk
- SQL is an easy to read, standard language to work with databases

What you have learned

- How to open a database
- Query a database to select columns and rows
- Compute aggregate statistics for groups of data
- Join data from multiple tables within a database
- Access SQLite databases using `R` and `Python`

Still keen to know more?

- We haven't discussed **missing data** at all.
  - See the Software Carpentry and Data Carpentry references in the `Acknowledgements`
- More Practice?
  - The example `householdWealth` has a guided exercise for you to work through

Acknowledgements

- This module is designed after and borrows a lot from:
  - `Effective Programming Practices for Economists`, a course by Hans-Martin von Gaudecker
  - Software Carpentry's `Using Databases and SQL` lesson
  - Data Carpentry's `Ecology Workshop`
- The `Household Wealth` example borrows data and inspiration from the `Data Management` session of Hans-Martin's course
- The `Auctions` example borrows data and some modified source codes from the book `A Gentle Introduction to Effective Computing in Quantitative Research: What Every Research Assistant Should Know` by Harry J Paarsch and Konstantin Golyaev

License

- Suggested Citation:
  Deer, Lachlan; Adrian Etter, Julian Langer, Max Winkler (2017), `Managing Data with SQL`, Programming Practices for Research in Economics, University of Zurich.

Programming Practices Team

Programming Practices for Research in Economics was created by

```
* Lachlan Deer
* Adrian Etter
* Julian Langer
* Max Winkler
```

at the Department of Economics, University of Zurich in 2016. These slides are from the 2017 edition, conducted by the original course developers.