

# Relational Databases and SQL Basics

Department of Biostatistics and Bioinformatics

Steve Pittard [wsp@emory.edu](mailto:wsp@emory.edu)

February 3, 2020

# Motivations

“Excel is the World’s Most Used Database”



# Motivations

There are limitations on the types (and size) of data that programming languages and stat analysis packages can handle well:

- Extremely large data sets are difficult to manage. Memory (RAM) and disk space is an issue
- Concurrency is an issue. One user at a time access.
- Persistence of data between analysis sessions is a problem. How do you maintain data ?

# Advantages of RDBMS

Relational Database Management Systems, (RDBMS), are designed to do all of these things well.

- Fast access to arbitrary parts of large data warehouses
- Powerful ways to summarize and cross-tabulate data
- Store data in more optimal and organized ways than the rectangular grid model of spreadsheets and R data frames
- Concurrent access from multiple clients running on multiple hosts while enforcing security constraints on access to the data

Taken from <http://cran.r-project.org/doc/manuals/R-data.html#Relational-databases>

# Advantages of RDBMS

Databases are useful when the size and organization of the data is large, complex, or requires security, (e.g. patient data, proprietary information).

- Commonly used in areas such as bioinformatics, medical records, machine learning, text mining, search algorithms
- Users search, extract, and collate data from the RDBMS and then do statistical analyses on the extracted info.
- You can create your own databases. Take a class - Coursera and Edx have free courses.

# Some Examples

Here are some popular RDBMS packages. Some are free - some aren't.

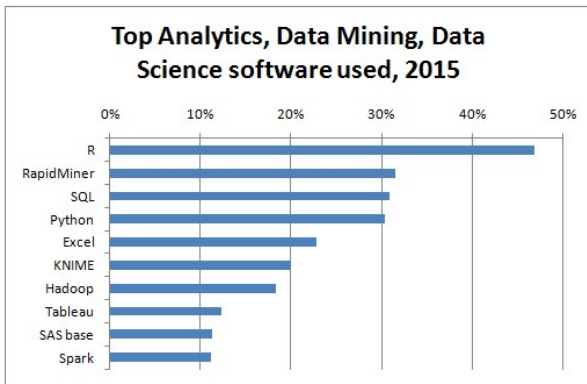
**IBM DB2**



Note this survey is biased by the Kdnuggets site (RapidMinder in particular)

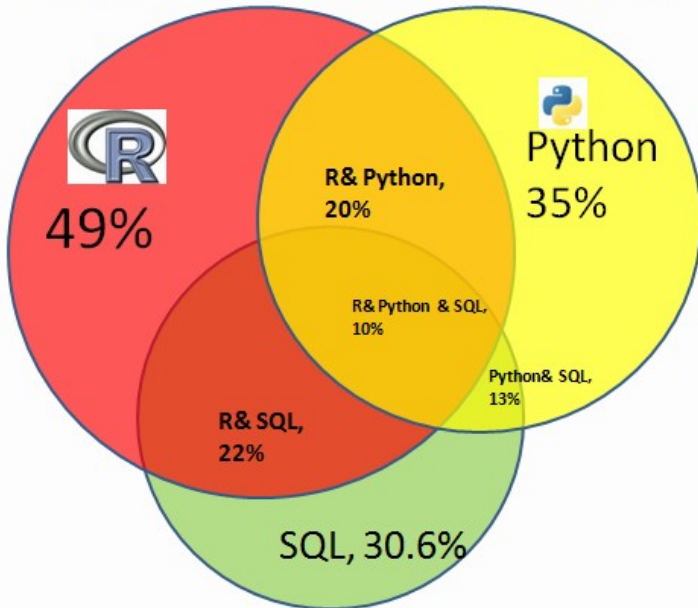
### Top Analytics Tools and Trends

Here are the top 10 tools by share of usage:



Taken from <http://www.kdnuggets.com/2015/05/poll-r-rapidminer-python-big-data-spark.html>

## KDnuggets 2014 Poll: Languages used for Analytics/Data Mining



Taken from <http://www.kdnuggets.com/images/languages-data-mining-r-python-sql.jpg>



# Database Jobs

There are different jobs for databases and users of SQL. The traditional ones are:

- **Database Administrator (DBA)** - Responsible for installing, configuring and maintaining a database management system (DBMS). Often tied to a specific platform such as Oracle, MySQL, DB2, SQL Server and others.
- **Database Architect** - Prepare and map out how the databases should look.
- **Database Designer/Database Architect** - Researches data requirements for specific applications or users, and designs database structures and application capabilities to match.
- **Database Developer** - Works with generic and proprietary APIs to build applications that interact with DBMSs (also platform specific, as with DBA roles).

# Database Jobs

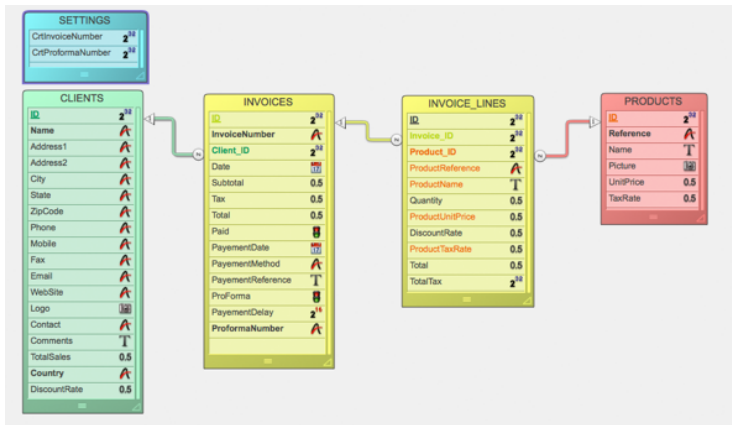
But there are newer career directions that rely upon a knowledge of SQL such as Data Scientist and Business Intelligence Analyst. Here is a set of job skill requirements from a job posting at Coursera.

## Your skills

- 3+ years of industry experience in a relevant role
- Demonstrated aptitude for independent research based on past project
- Strong applied statistics and data visualization skills
- Proficient with relational databases and SQL
- Proficient with at least one scripting language (e.g. Python)
- Proficient with at least one statistical software package (e.g. R, MatLab, or NumPy, SciPy, Pandas)
- Communicates technical concepts clearly and concisely in oral and written form
- B.S., M.S., or PhD. in Applied Mathematics, Statistics, Computer Science, Economics, Operations Research, or related technical field

Data is organized in *schemas* which describe the objects, (tables). Tables are stored in a database.

A single database can contain one or more tables. The schema of the table describes the nature of the columns (e.g. numeric, character, integer, etc)



# R concepts vs. Database concepts

R Concept	Database Concept
multiple related yet separate dataframes	Database
namespaces (sort of)	schema
data frame	table
variable (column)	column (attribute)
observation (row)	row(tuple)
subset(), [ ], transform, filter, select etc	SQL

Table: See <https://goo.gl/YNC5ML>

# Introducing SQL

- Databases can be created and maintained locally on your computer or they can exist on a remote server. As long as you know the address of the server, and have permission, you can query the remote database.
- SQL, (Structured Query Language), is a language based on relational algebra that allows us to search and extract data. SQL includes capabilities to do data insert, query, update, delete, schema creation, and modification.
- SQL Server might support a slightly different SQL command set than MySQL or Postgres. These differences can usually be addressed with minor adjustments in the query.

# Introducing SQL

- S.Q.L is generally pronounced “sequel” or like “Ess Queue El”.
- Declarative - Write simple queries to extract data. You don’t need to know how that happens behind the scenes.
- SQL can be sent to a database interactively via a command line or GUI client but more common to happen from within a high level programming language such as R,Java,Python,C,C++,SAS, etc.
- Supported by all major RDBMS

# Introducing SQL

There are two types of command sets in SQL: 1) DDL (Data Definition Language) and 2) DML (Data Manipulation Language)

- Data Definition Language

- ▶ Create Table
- ▶ Drop Table
- ▶ Alter Table

- Data Manipulation Language

- ▶ Select
- ▶ Insert
- ▶ Delete
- ▶ Update

# Typical Scenarios

There are two scenarios, (at least), that are common when working with databases:

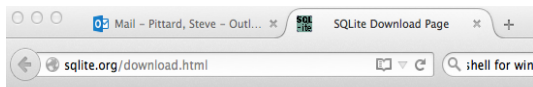
- Users search for and extract data to an intermediate file, (e.g. a .csv file) after which they import it into their favorite analysis package.
- Users search for and extract data from within a program they have written. This is useful if repeated sampling is necessary based on some computation as the program executes.



# SQLite

For this presentation we will use SQLite because it is free, lightweight, yet powerful. It comes preinstalled on Apple OSX and can be installed on Windows.

Go to <http://www.sqlite.org/download.html> to download the package. It is command line based although there are GUI front ends available.



[About](#) [Sitemap](#) [Documentation](#) [Download](#) [License](#)

## SQLite Download Page

# SQLite

## SQLite advantages

- No server administration required (unlike MySQL, Postgres, Oracle)
- Data is stored in a single, cross-platform file
- Database sizes up to 140TB are theoretically possible
- Can be access from any modern language
- Cellphone and other mobile device applications commonly use SQLite

<http://goo.gl/BTIdr9>

# MySQL

However, for actual production work I would suggest using MySQL which is a free robust enterprise grade database server and client that you can run on your laptop. It is more involved to manage but that is only because it does more than SQLite.



# SQLite

## SQLite advantages

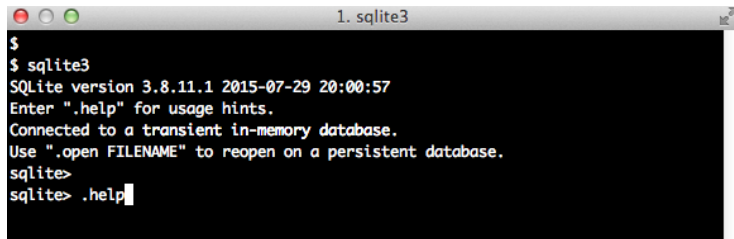
- No server administration required (unlike MySQL, Postgres, Oracle)
- Data is stored in a single, cross-platform file
- Database sizes up to 140TB are theoretically possible
- Can be access from any modern language
- Cellphone and other mobile device applications commonly use SQLite

<http://goo.gl/BTIdr9>

# SQLite

SQLite comes with a command line shell utility. On an Apple it is already installed by default. You just launch a terminal and type “sqlite3” to get it up and running.

If you are a Windows user then you will need to download and install the SQLITE shell as indicated above but once you get it installed it behaves the same as the client shell on Apple



```
1. sqlite3
$
$ sqlite3
SQLite version 3.8.11.1 2015-07-29 20:00:57
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite>
sqlite> .help
```

# SQLite GUI

There are various Graphical User Interfaces available for SQLite that try to make the process of creating and managing databases easier.

These are helpful to get you up and running with databases and I don't mind if you use them although you should over time gravitate towards the shell / command line because it offers more flexibility.



<http://sqlitebrowser.org/>

# SQLite

## SQLite Firefox Manager

*SQLite Manager 0.8.3.1-signed*

by [lazierthanthou](#)

Manage any SQLite database on your computer.

**+ Add to Firefox**

[Privacy Policy](#)

# What is a Database ?

- A database is a set of named relations which are also known as tables. Think of an Excel WorkBook that contains one or more WorkSheets.
- Each Worksheet can be related to another Worksheet via formulas,etc.
- In RDBMS we create a database name and THEN create tables within the database.
- We can specify things like what are the uniquely identifying aspects of the table (for example a unique student id.)
- This enables us to “link” between tables if necessary. We aren't obligated to do that but many SQL commands will span multiple tables.



# Creating a Database

Let's say we have .csv file containing info on three species of iris flowers, (Setosa, Virginica, and Versicolor). Columns 1-4 are numeric values representing sepal length, sepal width, petal length, petal width. Column 5 is species.

You can download this file from [https://raw.githubusercontent.com/pittardsp/info550\\_spring\\_2018/master/SUPPORT/iris.csv](https://raw.githubusercontent.com/pittardsp/info550_spring_2018/master/SUPPORT/iris.csv)

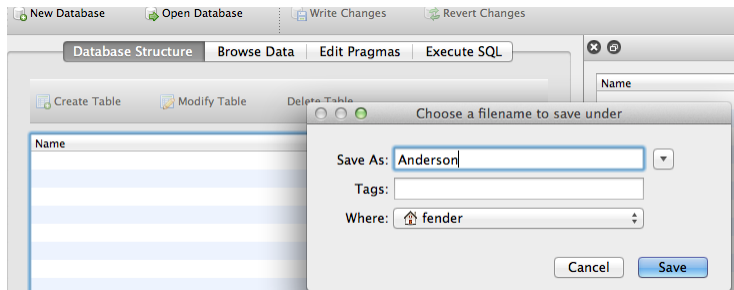
```
$ head iris.csv
5.1,3.5,1.4,0.2,setosa
4.9,3,1.4,0.2,setosa
4.7,3.2,1.3,0.2,setosa
4.6,3.1,1.5,0.2,setosa
5,3.6,1.4,0.2,setosa
5.4,3.9,1.7,0.4,setosa
4.6,3.4,1.4,0.3,setosa
```

We could read this into R or Excel and work with it there but let's see what SQLite can do for us.

# Create a Database and a Single Table - GUI

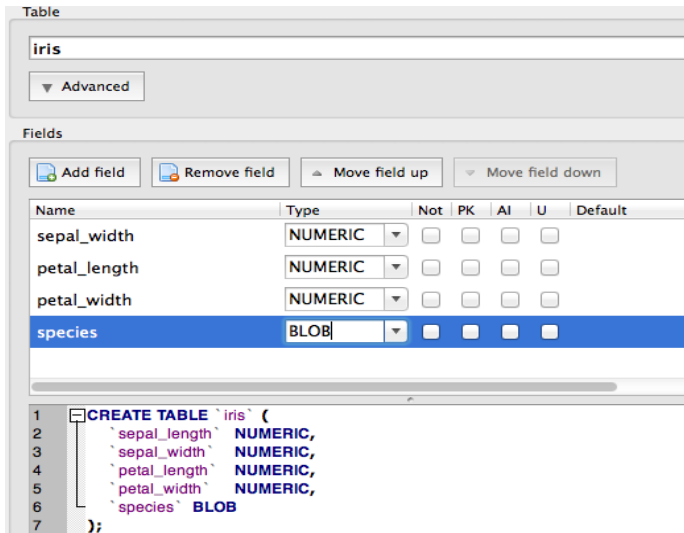
This data actually comes from the Edgar Anderson data set which is famous in statistics.

We will use the GUI to create a database called “Anderson” and then create a single table called “iris” to host the data.



# Create a Database and a Single Table - GUI

After we give the name “Anderson” to the database then we’ll create a table called “iris” to host the information in the .csv file.



The screenshot shows a database GUI with a table named "iris". The table has four fields: "sepal\_width", "petal\_length", "petal\_width", and "species". The "species" field is highlighted in blue. Below the table configuration, the SQL code for creating the table is displayed:

```
1 CREATE TABLE `iris` (  
2   `sepal_length` NUMERIC,  
3   `sepal_width` NUMERIC,  
4   `petal_length` NUMERIC,  
5   `petal_width` NUMERIC,  
6   `species` BLOB  
7 );
```

# Create a Database and a Single Table - GUI

Now we can import the iris.csv file into the table we just created

Table name

Column names in first line ☐

Field separator

Quote character

Encoding

Trim fields? ☒

	1	2	3	4	5
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa

## Creating a Database and a Single Table - Shell

If we wanted to create a database and table using the SQLite command shell we would do it as follows. Here is how:

```
$ sqlite3 iris.db    # Creates a database called "iris"
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

This creates a database called "iris". If the database were pre-existing then SQLite would simply load it in.

## Creating a Database and a Single Table - Shell

Next we create one or more tables to host the data in the iris.csv file. The table is a template describing the column names as well as what type they are (e.g. numeric, double, character, date, etc).

```
sqlite> create table iris (sepal_length numeric(5,1),  
...> sepal_width numeric(5,1),  
...> petal_length numeric(5,1),  
...> petal_width numeric(5,1),  
...> species varchar(10));  
sqlite>
```

We created it on the fly though tables can be created from within a programming language such as Python, R, C++, Java, Perl, etc.

# Creating a Database and a Single Table - Shell

To verify our work let's issue the *schema* command:

```
sqlite> .schema
CREATE TABLE iris (sepal_length numeric(5,1),
sepal_width numeric(5,1),
petal_length numeric(5,1),
petal_width numeric(5,1),
species varchar(10));
```

So now let's import the data:

```
sqlite> .header on
sqlite> .mode csv
sqlite> .import iris.csv iris
```

# Introducing SQL

The SELECT statement is the most frequently used SQL command.

It is basically three clauses: 1) FROM, 2) WHERE, 3) SELECT although they do not necessarily appear in that order

```
SELECT (what column attributes, data to return)
FROM   (relations between database table(s) )
WHERE  (some condition is met)
```



# Basic SQL Examples

So now we can start having fun.

```
sqlite> select count(*) from iris;  
count(*)  
150
```

```
sqlite> select count(*) as total_iris from iris;  
total_iris  
150
```

```
sqlite> .header off  
sqlite> select count(*) from iris where species = "versicolor";  
50
```

```
sqlite> select species, sepal_length from iris  
        where sepal_length >= 7.8;  
virginica|7.9
```

We don't have to know a lot about SQL to leverage its power. Just a few commands will help us.

# SQL Overview

SQL uses the **SELECT** statement to search and extract data. A general format is given here.

```
SELECT column or computations
      FROM table
      WHERE condition
      GROUP BY columns
      HAVING conditions
      ORDER BY column (ascending or descending)
      LIMIT offset, count;
```

Not all of these specifiers are necessary.

SQL	Result
SELECT * from iris	Extracts all rows and columns
SELECT sepal_width, petal_width from iris	A two column result

# SQL Overview

```
SELECT * FROM iris WHERE species = 'setosa' LIMIT 5
```

Gets all Setosa records but list only the first 5 records

```
SELECT * FROM iris WHERE species = 'setosa' AND sepal_width >4.0
```

Gets all Setosa or Versicolor records with sepal\_width greater than 4.0

```
SELECT * FROM iris WHERE (species = 'setosa' OR species = 'versicolor')  
AND sepal_width >3.3 LIMIT 5
```

Gets all setosa or versicolor records whose sepal\_width is >than 3.3. Display only 5 records.

```
SELECT species, sepal_width, sepal_length from iris where sepal_width >3.5  
ORDER BY species
```

Gets species, sepal\_width, sepal\_length where sepal\_width is greater than 3.5  
Result is then ordered by species alphabetically.

# SQL Examples

```
sqlite> SELECT * FROM iris WHERE species = 'setosa' LIMIT 5;
```

```
5.1|3.5|1.4|0.2|setosa
```

```
4.9|3|1.4|0.2|setosa
```

```
4.7|3.2|1.3|0.2|setosa
```

```
4.6|3.1|1.5|0.2|setosa
```

```
5|3.6|1.4|0.2|setosa
```

```
sqlite> SELECT * FROM iris WHERE species = 'setosa' AND sepal_width > 4.0 ;
```

```
5.7|4.4|1.5|0.4|setosa
```

```
5.2|4.1|1.5|0.1|setosa
```

```
5.5|4.2|1.4|0.2|setosa
```

```
sqlite> SELECT * FROM iris WHERE (species = 'setosa' OR species = 'versicolor')  
      AND sepal_width > 3.3 LIMIT 5;
```

```
5.1|3.5|1.4|0.2|setosa
```

```
5|3.6|1.4|0.2|setosa
```

```
5.4|3.9|1.7|0.4|setosa
```

```
4.6|3.4|1.4|0.3|setosa
```

```
5|3.4|1.5|0.2|setosa
```

```
sqlite> select * from iris where species like '%osa%' limit 1;
```

```
5.1|3.5|1.4|0.2|setosa
```

# SQL Aggregation Functions

SQL provides some aggregate functions. These are basic though can be used to compute more sophisticated quantities. However do not try to use SQL as a substitute for R, SAS, or SPSS.

TASK	SQL Aggregate Function
Count number of occurrences	COUNT()
Compute the sum	SUM()
Get the mean	AVG()
Get the minimum	MIN()
Get the maximum	MAX()
Get the variance	VAR_SAMP()
Get the standard deviation	STDDEV_SAMP()

# SQL Examples

```
sqlite> select avg(sepal_length), avg(sepal_width) from iris;  
5.84333333333333|3.05733333333333
```

```
sqlite> select round(avg(sepal_length),2), round(avg(sepal_width),2) from iris;  
5.84|3.06
```

```
sqlite> select round(avg(sepal_length),2),round(avg(sepal_width),2)  
        from iris group by species;  
5.01|3.43  
5.94|2.77  
6.59|2.97
```

# But we need the species names here too

```
sqlite> select species, round(avg(sepal_length),2), round(avg(sepal_width),2)  
        from iris group by species;  
setosa|5.01|3.43  
versicolor|5.94|2.77  
virginica|6.59|2.97
```

# SQL Examples

Let's say that we want the average sepal width for each of the three species but only for all records wherein the sepal length is  $> 2.8$

```
sqlite> select species,avg(sepal_width) from iris where sepal_width > 2.8  
        group by species;
```

```
species|avg(sepal_width)  
setosa|3.45102040816327  
versicolor|3.03913043478261  
virginica|3.16451612903226
```

```
sqlite> select species,round(avg(sepal_width),2) from iris where sepal_width > 2.8  
        group by species;
```

```
species|round(avg(sepal_width),2)  
setosa|3.45  
versicolor|3.04  
virginica|3.16
```

# The "as" operator

The "as" operator can cause some initial confusion until you see a few examples. At its most basic we use it to name columns in the output. To see the result you need to turn headers on in sqlite

```
sqlite> .headers on
sqlite> select species,avg(sepal_length) from iris;
species|avg(sepal_length)
virginica|5.84333333333333
```

```
sqlite> select species,avg(sepal_length) as mean_sepal_length from iris;
species|mean_sepal_length
virginica|5.84333333333333
```

```
sqlite> select species as Iris_Species,avg(sepal_length) as mean_sepal_length
        from iris;
Iris_Species|mean_sepal_length
virginica|5.84333333333333
```



# The "as" operator

Let's say that we want the average sepal width for each of the three species but only where the average for the respective group exceeds 2.8.

This is different than before. We already know the answer from looking at a previous slide but let's work through it.

```
sqlite> select species, avg(sepal_width) from iris where avg(sepal_width) > 2.8;  
Error: misuse of aggregate: avg()
```

# That didn't work but we aren't sure why. Let's try to use "as"

```
sqlite> select species, avg(sepal_width) as mean from iris where mean > 2.8;  
Error: misuse of aggregate: avg()
```

# The "as" operator

We need to extend our knowledge of SQL to solve this problem

```
sqlite> select species, round(avg(sepal_width),2) as mean
        from iris group by species HAVING mean > 2.8;
```

```
species|mean
setosa,3.43
virginica,2.97
```

# If we remove the requirement for the mean to be > 2.8

```
sqlite> select species, round(avg(sepal_width),2) as mean
        from iris group by species;
```

```
species|mean
setosa|3.43
versicolor|2.77
virginica|2.97
```

# Let's sort the result by the mean in ascending order

```
sqlite> select species, round(avg(sepal_width),2) as mean
        from iris group by species order by mean asc;
```

```
species|mean
versicolor|2.77
virginica|2.97
setosa|3.43
```

# Insertions

We can also insert new records into the database. Usually this is done from a program or script which loads many records at once. However, you can load records one at a time if you wish.

```
sqlite> insert into iris values (5.1,3.5,1.4,0.2,"setosa");  
sqlite> insert into iris values (4.9,3.5,1.4,0.2,"setosa");
```

We can also update or delete existing records based on some condition.  
An example:

```
sqlite> delete from iris where species = 'versicolor';  
sqlite> select count(*) from iris where species = 'versicolor';  
0
```

# Access from within R

We can also access databases from within R. There are different ways to do this. Let's say we have a pre-existing database such as the one we just created - iris.db

```
library(DBI)
con <- dbConnect(RSQLite::SQLite(),dbname="iris.db")
dbListTables(con)
[1] "iris"
```

```
(res <- dbGetQuery(con, "select count(*) from iris"))
count(*)
1      150
```

```
(res <- dbGetQuery(con, "select * from iris where sepal_length > 3.0 limit 5"))
sepal_length sepal_width petal_length petal_width species
1           5.1         3.5         1.4         0.2  setosa
2           4.9         3.0         1.4         0.2  setosa
3           4.7         3.2         1.3         0.2  setosa
4           4.6         3.1         1.5         0.2  setosa
5           5.0         3.6         1.4         0.2  setosa
```

# Access from within R

Using DBI allows us to connect to the database and issue SQL to it. The performance on this is good and we get back information in R friendly structures

```
library(DBI)
con <- dbConnect(RSQLite::SQLite(), dbname="iris.db")
dbListTables(con)
[1] "iris"
```

```
(res <- dbGetQuery(con, "select count(*) from iris"))
  count(*)
1      150
```

```
(res <- dbGetQuery(con, "select * from iris where sepal_length > 3.0 limit 5"))
  sepal_length sepal_width petal_length petal_width species
1          5.1          3.5          1.4          0.2  setosa
2          4.9          3.0          1.4          0.2  setosa
3          4.7          3.2          1.3          0.2  setosa
4          4.6          3.1          1.5          0.2  setosa
5          5.0          3.6          1.4          0.2  setosa
```

# Access from within R

There is the “sqldf” package that allows you to query data frames as if they were database tables. The sqldf package creates an SQLite database/table behind the scenes to accommodate.

```
> library(sqldf)
> data(mpg, package = "ggplot2") # Get the mpg data frame from ggplot2
> head(mpg,2)
  manufacturer model displ year cyl      trans drv  cty   hwy fl      class
1         audi   a4    1.8 1999   4    auto(l5)  f   18   29 p compact
2         audi   a4    1.8 1999   4 manual(m5)  f   21   29 p compact

> sqldf("select * from mpg where hwy > 35 and cty > 20")
  manufacturer      model displ year cyl      trans drv  cty   hwy fl      class
1         honda     civic   1.8 2008   4    auto(l5)  f   25   36 r subcompact
2         honda     civic   1.8 2008   4    auto(l5)  f   24   36 c subcompact
3        toyota  corolla   1.8 2008   4 manual(m5)  f   28   37 r   compact
4  volkswagen    jetta    1.9 1999   4 manual(m5)  f   33   44 d   compact
5  volkswagen new beetle   1.9 1999   4 manual(m5)  f   35   44 d subcompact
6  volkswagen new beetle   1.9 1999   4    auto(l4)  f   29   41 d subcompact
```

## Access from within R

Note that the above is equivalent to the following which is how one would usually do this within R. The advantage of using the SQL approach is that it generalizes outside of R.

```
mpg %>% filter(hwy > 35 & cty > 20)
```

```
Source: local data frame [6 x 11]
```

	manufacturer (chr)	model (chr)	displ (dbl)	year (int)	cyl (int)	trans (chr)	drv (chr)	cty (int)	hwy (int)	fl (chr)
1	honda	civic	1.8	2008	4	auto(l5)	f	25	36	r subc
2	honda	civic	1.8	2008	4	auto(l5)	f	24	36	c subc
3	toyota	corolla	1.8	2008	4	manual(m5)	f	28	37	r c
4	volkswagen	jetta	1.9	1999	4	manual(m5)	f	33	44	d c
5	volkswagen	new beetle	1.9	1999	4	manual(m5)	f	35	44	d subc
6	volkswagen	new beetle	1.9	1999	4	auto(l4)	f	29	41	d subc

Behind the scenes the sqldf package creates an SQLite version of the data. If you are working with an existing multi-table database then you should use the DBI approach described above because it performs better.

```
# How many cars have a manual transmission ?
```

```
sqldf("select count(*) from mpg where trans like '%man%'")
      count(*)
1           77
```

```
# Here is an equivalent R expression
```

```
filter(mpg, grepl("manual",trans)) %>% nrow()
[1] 77
```



Find the average city MPG by manufacturer. Sort it from highest to lowest

```
> sqldf("select manufacturer, avg(cty) as mean from mpg  
        group by manufacturer order by mean desc")
```

	manufacturer	mean
1	honda	24.4
2	volkswagen	20.9
3	subaru	19.3
4	hyundai	18.6
5	toyota	18.5
6	nissan	18.1
7	audi	17.6
8	pontiac	17.0
9	chevrolet	15.0
10	ford	14.0
11	jeep	13.5
12	mercury	13.2
13	dodge	13.1
14	land rover	11.5
15	lincoln	11.3

Same as

```
mpg %>% group_by(manufacturer) %>% summarize(mean=mean(cty)) %>%  
  arrange(desc(mean))
```

Source: local data frame [15 x 2]

	manufacturer (chr)	mean (dbl)
1	honda	24.44444
2	volkswagen	20.92593
3	subaru	19.28571
4	hyundai	18.64286
5	toyota	18.52941
6	nissan	18.07692
7	audi	17.61111
8	pontiac	17.00000
9	chevrolet	15.00000
10	ford	14.00000
11	jeep	13.50000
12	mercury	13.25000
13	dodge	13.13514
14	land rover	11.50000
15	lincoln	11.33333

Use the `read.csv.sql` function to read in parts of really large .csv files. Here is an example using the internal `mtcars` dataset. We'll write it out and read it back in.

It's not a big file but provides a prototype example that could be used for huge files. We'll read in only those records corresponding to 4 cylinder cars.

```
> library(sqldf)
> write.csv(mtcars,"mtcars.csv",quote=F,row.names=F)
> mtcars2 <- read.csv.sql("mtcars.csv",sql = "select * from file where cyl = 4")
> head(mtcars2)
```

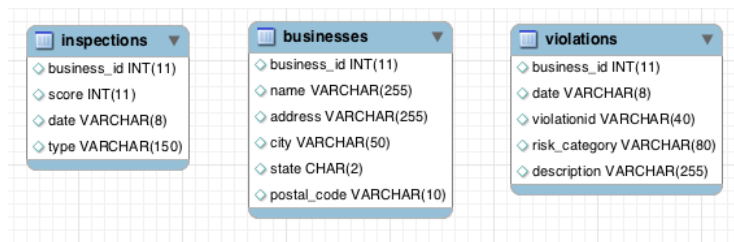
	mpg	cyl	displacement	horsepower	drat	wt	qsec	vs	am	gear	carb
1	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
2	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
3	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
4	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
5	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
6	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1

# Multiple Tables

- Up until now the database has been very simple. It contains only one table. In reality most databases have multiple tables with an identifier or key in each table that can be referenced when writing SQL statements
- When we query multiple tables using SQL we call this a “join” although one could do a “self join” on a single table.
- Joins and keys can be complicated so here we will just outline some of the basics.
- Let’s use an example where we have a database with three tables. This data relates to restaurant inspection information in San Francisco. For more info see my blog posting at <http://tinyurl.com/14qnjbu>

# Multiple Tables

When doing database design or trying to reverse engineer a database structure we frequently look at Entity Relationship diagrams or perhaps UML diagrams that provide a graphical view of the tables and the possible relationships between the tables.



# Multiple Tables

SQLite databases can be dumped into a single file that can later be read by others. The dump file contains the database and schema table for all tables.

Download this file using your browser, wget, curl, or whatever command you usually use to do download files. Make note of where you download it and use the sqlite shell to read it in:

```
https://github.com/pittardsp/info550\_spring\_2018/blob/master/SUPPORT/restaurants.db
```

```
# Read the database
$ sqlite3 restaurants.db
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

You now have a working copy of the database and the three supporting tables as well as the data.

# Multiple Tables

Let's do some exploration.

```
sqlite> select count(score) as 'number_of_inspections' from inspections;  
number_of_inspections  
40936
```

# How many scores were below 70 ?

```
sqlite> select count(score) as score70 from inspections where score < 70;  
score70  
373
```

# Below 80 ?

```
sqlite> select count(score) as score80 from inspections where score < 80;  
score80  
1793
```

# Multiple Tables

Okay well WHO has the LOWEST numeric score overall ?


```
sqlite> select name, score from businesses, inspections
        where businesses.business_id = inspections.business_id
        order by score limit 15;
```

```
PACIFIC SUPER MARKET|42
DICK LEE PASTRY|42
MANILA MARKET & GROCERIES|44
KL1 RESTAURANT|45
NEW ASIA RESTAURANT|47
ALBORZ|47
MANILA MARKET & GROCERIES|48
PUNJAB KABAB HOUSE|49
MEE HEONG BAKERY|49
HAPPY CHINESE RESTAURANT|50
KUSINA NI TESS|50
BANGKOK NOODLES & THAI BBQ|50
BROADWAY DIM SUM|51
NEW ASIA RESTAURANT|51
HONG KEE & KIM|51
```



# Multiple Tables

Okay well WHO has the LOWEST numeric score overall ?



```
SELECT name, score from businesses, inspections
WHERE businesses.business_id = inspections.business_id
ORDER BY score limit 15;
```

# Multiple Tables

How many violations exist for each restaurant ?

```
sqlite> select name, count(violationid) from businesses, violations
        where businesses.business_id = violations.business_id
        group by name limit 15;
```

```
HOL N JAM LEMONADE STAND #2|2
HOL N JAM LEMONADE STAND #1|6
HOL N JAM LEMONADE STAND #3|4
NORDSTROM CAFE BISTRO|19
1-CREDE|7
100% DESSERT CAFE|11
1058 HOAGIE|5
123 DELI - LEE'S|3
1300 ON FILLMORE|1
1760|1
17TH & NOE MARKET|6
18 REASONS|4
18TH STREET COMMISSARY|3
19TH AVE SHELL|9
20 SPOT MISSION, LLC|1
```

# Multiple Tables

Well that wasn't sorted. I want to know the place with the most violations

```
sqlite> select name, count(violationid) as volcnt from businesses, violations
        where businesses.business_id = violations.business_id
        group by name order by volcnt desc limit 15;
```

```
STARBUCKS COFFEE|327
PEET'S COFFEE & TEA|189
MCDONALDS|121
QUICKLY|90
KENTUCKY FRIED CHICKEN|85
ROUND TABLE PIZZA|70
SPECIALTY'S CAFE & BAKERY|68
SAN FRANCISCO SOUP COMPANY|59
BURGERMEISTER|58
TEAWAY|58
HO'S BAR & RESTAURANT INC|56
KING OF THAI NOODLE HOUSE|54
MARNEE THAI RESTAURANT|54
YOPPI YOGURT|54
NORTH BEACH PIZZA|53
```

# Multiple Tables

Wait a minute. So was it one Starbuck's location that got 327 violations ? Probably not. There are multiple Starbucks. How could we find out ?

The DISTINCT function in SQL will show us how many distinct business ids there are associated with any business containing "STARBUCK" in the name.

```
sqlite> select count(distinct(business_id)) from businesses  
        where name like '%STARBUCK%';
```

71

# Multiple Tables

So there are 71 Starbucks in the area. The 327 violations are spread over them. But how many per each

```
sqlite> select count(distinct(business_id)) from businesses
        where name like '%STARBUCK%';
```

71

```
sqlite> select name, businesses.business_id, count(violationid) from
        businesses, violations where name like '%STARBUCK%' and
        businesses.business_id = violations.business_id group by
        businesses.business_id order by count(violationid) desc limit 10;
```

```
STARBUCKS COFFEE|2072|18
STARBUCKS COFFEE|4425|13
STARBUCKS COFFEE|19416|13
STARBUCKS COFFEE|1505|12
STARBUCKS COFFEE|2633|12
STARBUCKS COFFEE|2799|12
STARBUCKS COFFEE|2805|12
STARBUCKS COFFEE|35771|12
STARBUCKS COFFEE|2218|11
STARBUCKS COFFEE|4656|11
```

# Dates

We can use SQL to extract query and data between given date ranges. We need to use a helper function that tells SQLite that we are processing a true date as opposed to a mere character string.

See [https://www.sqlite.org/lang\\_datefunc.html](https://www.sqlite.org/lang_datefunc.html) for more info on dates in SQLite.

Let's determine how many inspections took place between March 27, 2014 and April 10, 2014 ?

```
sqlite> select count(*) from inspections where strftime(date) >
        strftime('20140327') and strftime(date) < strftime('20140410');
563
```

# Missing Values

Frequently we have missing values in data. It might be blank or have something like "NA". We have to be on the lookout for this.

```
sqlite> select avg(score) from inspections limit 5;  
51.1987736955247
```

```
sqlite> select avg(score) from inspections where score not like '%NA%';  
91.952485412188
```

```
sqlite> select postal_code, avg(score) as mean from businesses,  
        inspections where businesses.business_id = inspections.business_id  
        and score not like '%NA%' group by postal_code order by mean  
        asc limit 5;
```

```
postal_code|0.0  
94014|85.66666666666667  
94609|86.375  
94133|87.6198514517218  
94122|87.6516976998905  
sqlite>
```

## How would we handle this using DBI ? Easy

```
library(DBI)

con <- dbConnect(RSQLite::SQLite(),dbname="restaurants.db")

dbListTables(con)
[1] "businesses" "inspections" "violations"

(res <- dbGetQuery(con, "select postal_code, avg(score) as mean from
                           businesses,inspections where
                           businesses.business_id = inspections.business_id
                           and score not like '%NA%'group by postal_code
                           order by mean asc limit 5"))
```

	postal_code	mean
1	postal_code	0.0
2	94014	85.7
3	94609	86.4
4	94133	87.6
5	94122	87.7



# Summary

Use relational databases when:

- There is lots and lots of data
- When the data is complex and/or has many interrelationships
- Other people might want to work with the data but might want different subsets
- When you need to put up a website that is a front end for the data

**Don't attempt to use SQL for in depth statistical analysis.** In such cases use SQL to get data and then import it into SAS, Python, SPSS, R, MatLab, Java or whatever language you want to use.

Most languages have methods that allow you to query databases directly from the language if you want to do that.

# Summary

SQL is powerful. With it we query databases.

- Very useful for extracting data and subsets thereof
- Can be called from most languages R, SAS, Java, Python, SPSS
- Mastering SQL, like any other language, takes effort
- Can be extended to include GIS capabilities (see GIS mods for MySQL)

# Do you Really Need SQL ?

## The rise of the ORMs ! Object Relational Mappers

- These are packages that allow you to communicate to databases from your language of choice (e.g. Python, Java, etc)
- You don't have to know SQL to use ORMs
- Using ORMs results in much less code than when trying to use native SQL directly
- ORMs are good for when you don't care about the structure of the database - you just want to access it somehow

# Do you Really Need SQL ?

R has some great packages that let you do “SQL-like” things without leaving R or learning SQL explicitly.

- The **data.table** and **dplyr** package are awesome !
- **data.table** is great for reading in huge files
- Both **data.table** and **dplyr** have cool ways of implementing SAC
- SAC stands for **Split-Apply-Combine**
- We **Split** data on some factor(s), **Apply** some function (e.g. mean, sd, etc) to the split data, and then combine it into a summarized/aggregated form
- I prefer **dplyr** but frequently use **data.table** to read in large files
- You can mix both packages

I will use dplyr as a “training wheels” approach to learning SQL.

## Do you Really Need SQL ?

Check this out. If you have a SQLite database such as restaurants.db you can work with it using dplyr commands !

```
mydb <- src_sqlite(path="restaurants.db")
```

```
src:  sqlite 3.8.6 [restaurants.db]
```

```
tbls: businesses, inspections, violations
```

```
bus <- tbl(mydb,"businesses")
```

```
ins <- tbl(mydb,"inspections")
```

```
vio <- tbl(mydb,"violations")
```

# Do you Really Need SQL ?

So we can join the inspections with the business ids to find out who some of the worst restaurants are in terms of score

```
head(ins)
```

	business_id	score	date	type
1	business_id	Score	date	type
2	10	NA	20140124	Reinspection/Followup
3	10	92	20140114	Routine - Unscheduled
4	10	98	20121114	Routine - Unscheduled
5	10	NA	20120920	Reinspection/Followup
6	10	98	20120403	Routine - Unscheduled

```
head(bus)
```

	business_id	name	address	city	state	postal_code
1	business_id	name	address	city	state	postal_code
2	10	TIRAMISU KITCHEN	033 BELDEN PL	San Francisco	CA	9410
3	17	GEORGE'S COFFEE SHOP	2200 OAKDALE AVE	San Francisco	CA	9412
4	19	NRGIZE LIFESTYLE CAFE	1200 VAN NESS AVE, 3RD FLOOR	San Francisco		
5	24	OMNI S.F. HOTEL - 2ND FLOOR PANTRY	500 CALIFORNIA ST, 2NDFLOOR	San Francisco		
6	29	CHICO'S PIZZA	131 06TH ST	San Francisco	CA	9410

# Do you Really Need SQL ?

So we can join the inspections with the business ids to find out who some of the worst restaurants are in terms of score

```
inner_join(ins,bus) %>% arrange(score) %>% head(.,15) %>% select(score,name,address)
```

Joining by: "business\_id"

	score	name	address
1	42	PACIFIC SUPER MARKET	2900 ALEMANY BLVD
2	42	DICK LEE PASTRY	716 JACKSON ST
3	44	MANILA MARKET & GROCERIES	987 MISSION ST
4	45	KL1 RESTAURANT	4401 BALBOA ST
5	47	NEW ASIA RESTAURANT	772 PACIFIC AVE
6	47	ALBORZ	1245 VAN NESS AVE
7	48	MANILA MARKET & GROCERIES	987 MISSION ST
8	49	PUNJAB KABAB HOUSE	101 EDDY ST
9	49	MEE HEONG BAKERY	1343 POWELL ST
10	50	HAPPY CHINESE RESTAURANT	1324 POWELL ST
11	50	KUSINA NI TESS	237 ELLIS ST
12	50	BANGKOK NOODLES & THAI BBQ	110 POWELL ST
13	51	BROADWAY DIM SUM	684 BROADWAY ST
14	51	NEW ASIA RESTAURANT	772 PACIFIC AVE
15	51	HONG KEE & KIM	91 DRUMM ST

# Do you Really Need SQL ?

So we can join the violations and business tables to see the breakdown of violations across all Starbucks shops

```
inner_join(vio,bus) %>%  
  filter(name=="STARBUCKS COFFEE") %>%  
  group_by(business_id,violationid) %>%  
  count() %>% arrange(desc(n)) %>% head(5)
```

Joining, by = "business\_id"

```
# Source:      lazy query [?? x 3]  
# Database:    sqlite 3.29.0 [/Users/esteban/Downloads/restaurants.db]  
# Groups:      business_id, violationid  
# Ordered by:  desc(n)
```

	business_id	violationid	n
	<int>	<chr>	<int>
1	2799	103154	4
2	2805	103154	4
3	1085	103119	3
4	1359	103154	3
5	2072	103154	3