Introduction to SQL

Greg Ridgeway (gridge@upenn.edu)
Ruth Moyer (moyruth@upenn.edu)

July 27, 2018

Introduction to SQL

Some datasets are far too large for R to handle by itself. Structured Query Language ("SQL") is a widely used international standard language for managing data stored in databases. There are numerous relational database management systems such as Oracle, Microsoft Access, and MySQL. We are going to use SQLite, which is probably the most widely deployed database system. SQLite is in your phone, car, airplanes, thermostats, and numerous applicances. We are going to hook up SQLite to R so that R can handle large datasets.

These are some basic clauses in a SQL query that we will explore:

SELECT fields or functions of fields INTO results table FROM tables queried WHERE conditions for selecting a record GROUP BY list of fields to group ORDER BY list of fields to sort by

However, before being able to use SQL as a tool in R, it will first be necessary to install the `sqldf` package.

```
library(sqldf)
```

```
Loading required package: gsubfn
```

```
Loading required package: proto
```

```
Loading required package: RSQLite
```

Getting the data into proper form

We will be working with Chicago crime data, which is accessible comma separated value (csv) format. Before we can even being learning SQL, we are going to have to do a fair bit of work to acquire the dataset, format it so that it's ready for SQLite, and then load it into the SQLite database.

Navigate to the Chicago open data website to get the data. Click the "Export" button and select the "CSV" option, or directly download from here

The Chicago crime data is huge, more than 1.5Gb. It contains about 7 million records on all crimes reported to the Chicago police department since 2001. R does not handle really large datasets well. By using SQL, you will learn how to more efficiently work with large datasets and learn a data language that is used absolutely everywhere.

Let's use `scan()` to just peek at the first three rows of the file.

```
scan(what="",file="Crimes_-_2001_to_present.csv",nlines=5,sep="\n")
```

```
[1] "ID,Case Number,Date,Block,IUCR,Primary Type,Description,Location Description,Arrest,Domesti
[2] "10000092,HY189866,03/18/2015 07:44:00 PM,047XX W OHIO ST,041A,BATTERY,AGGRAVATED: HANDGUN,S
[3] "10000094,HY190059,03/18/2015 11:00:00 PM,066XX S MARSHFIELD AVE,4625,OTHER OFFENSE,PAROLE V
[4] "10000095,HY190052,03/18/2015 10:45:00 PM,044XX S LAKE PARK AVE,0486,BATTERY,DOMESTIC BATTER
[5] "10000096,HY190054,03/18/2015 10:30:00 PM,051XX S MICHIGAN AVE,0460,BATTERY,SIMPLE,APARTMENT
```

`scan()` is a very basic R function that reads in plain text files. We've told it to read in text (`what==""`), the name of the file, to only read in 5 lines (`nlines=5`), and to start a new row whenever it reaches a line feed character (`sep="\n"`). Using `scan()` without `nlines=5` would cause R to try to read in the whole dataset and that could take a lot of time and you might run out of memory.

You can see that the first row contains the column names. The second row contains the first reported crime in the file. You can see date and time, address, crime descriptions, longitude and latitude of the crime, and other information.

Importantly, SQLite is very particular about the formatting of a file. It can easily read in a csv file, but this dataset has some commas in places that confuse SQLite.

For example, there is a row in this file that looks like this:

```
[1] "10000153,HY189345,03/18/2015 12:20:00 PM,091XX S UNIVERSITY AVE,0483,BATTERY,AGG PRO.EMP: C
```

You see that the location description for this crime is `"SCHOOL, PUBLIC, BUILDING"`. Those commas inside the quotes are going to cause SQLite problems. SQLite is going to think that `SCHOOL`, `PUBLIC`, and `BUILDING` are all separate columns rather than one columns describing the location.

To resolve this, we're going to change all the commas that separate the columns into something else besides commas, leaving the commas in elements like `"SCHOOL, PUBLIC, BUILDING"` alone. It does not matter what we use to separate the fields, but it should be an unusual character that would not appear anywhere else in the dataset. Popular choices in the vertical bar (`|`) and the semi-colon (`;`). So let's take a slight detour to find out how to convert a comma-separated file into a semi-colon separated file.

You'll know if you need to convert your file if, when you try to set up your SQL database, you receive an error message about an "extra column."

We're going to use a `while` loop to read in 1,000,000 rows of the our data file at a time. R can handle 1,000,000 rows. With the 1,000,000 rows read in, we'll use a regular expression to replace all the commas used for separating columns with semi-colons. Then we'll write out the resulting cleaned up rows into a new file. It is a big file so this code can take a few minutes to run to completion.

```
[1] 1000000
[1] 2000000
[1] 3000000
[1] 4000000
[1] 5000000
[1] 6000000
[1] 6656622
```

Now, let's take a look at the first five lines of the new file we just created.

```
scan(what="",file="Crimes_-_2001_to_present-clean.csv",nlines=5,sep="\n")
```

```
[1] "ID;Case Number;Date;Block;IUCR;Primary Type;Description;Location Description;Arrest;Domesti
```

```
[2] "10000092;HY189866;03/18/2015 07:44:00 PM;047XX W OHIO ST;041A;BATTERY;AGGRAVATED: HANDGUN;S
[3] "10000094;HY190059;03/18/2015 11:00:00 PM;066XX S MARSHFIELD AVE;4625;OTHER OFFENSE;PAROLE V
[4] "10000095;HY190052;03/18/2015 10:45:00 PM;044XX S LAKE PARK AVE;0486;BATTERY;DOMESTIC BATTER
[5] "10000096;HY190054;03/18/2015 10:30:00 PM;051XX S MICHIGAN AVE;0460;BATTERY;SIMPLE;APARTMENT
```

You now see that semi-colons separate the columns rather than commas. That previous record that had the location description "SCHOOL, PUBLIC, BUILDING" now looks like this:

```
[1] "10000153;HY189345;03/18/2015 12:20:00 PM;091XX S UNIVERSITY AVE;0483;BATTERY;AGG PRO.EMP: C
```

Note that the commas are still there inside the quotes. Now we will be able to tell SQLite to look for semi-colons to separate the columns.

Setting up the Database

Now that the file containing the data is ready, we can load it into SQLite. SQLite has its own way of storing and managing data. It can store multiple tables containing data in a single database. First, we'll tell SQLite to create a new database that we will call `chicagocrime`. Then we will tell SQLite to load our data file into a table called `crime`.

The next step is to set up the SQL database. The following lines of code will set up the database for you. Make sure that your path is set correctly so that your database will be stored in the correct folder that you wish to work from. You will know if the database has been successfully set up if the database (stored as a .db file) is greater than 0 KB. there is no reason to run these lines of code again.

```
# create a connection to the database
con <- dbConnect(SQLite(), dbname="chicagocrime.db")

# peek at the first few rows of the dataset
a <- read.table("Crimes_-_2001_to_present-clean.csv",
                sep=";",nrows=5,header=TRUE)
# ask SQLite what data type it plans to use to store each column (eg number, text)
variabletypes <- dbDataType(con, a)
# make sure IUCR and Ward are stored as TEXT
variabletypes["IUCR"] <- "TEXT"
variabletypes["Ward"] <- "TEXT"
variabletypes["District"] <- "TEXT"
variabletypes["Community.Area"] <- "TEXT"

# just in case you've already created a "crime" table, delete it
if(dbExistsTable(con, "crime")) dbRemoveTable(con, "crime")
# import the data file into the database
dbWriteTable(con, "crime",                          # create crime table
             "Crimes_-_2001_to_present-clean.csv", # from our cleaned up file
             row.names=FALSE,
             header=TRUE,                           # first row has column names
             field.types=variabletypes,
             sep=";")                               # columns separated with ;
# does the table exist?
```

```
dbListTables(con)
# a quick check to see if all the columns are there
dbListFields(con,"crime")
# disconnect from the database to finalize
dbDisconnect(con)
```

```
[1] "crime"
 [1] "ID"                  "Case.Number"           "Date"
 [4] "Block"               "IUCR"                  "Primary.Type"
 [7] "Description"         "Location.Description" "Arrest"
[10] "Domestic"            "Beat"                  "District"
[13] "Ward"                "Community.Area"        "FBI.Code"
[16] "X.Coordinate"        "Y.Coordinate"          "Year"
[19] "Updated.On"          "Latitude"              "Longitude"
[22] "Location"
```

Once you've successfully set up your database, there is no reason to run these lines of code again. You should never again need to turn commas into semi-colons or run `dbWriteTable()`. Instead, every time you want to work with your database, you can simply need to reconnect to the database with:

```
con <- dbConnect(SQLite(), dbname="chicagocrime.db")
```

(Note that if you're keeping you are using a cloud-based backup service like iCloud, OneDrive, or Google Drive, you might need to wait until your "db" file has completely "synced" before you can access your database.)

SQL queries

You've now created a database (called "chicagocrime.db") containing a table called "crime" that contains those 7 million crime records.

Two important clauses with an SQL query are `SELECT` and `FROM`. Unlike R, SQL queries are not case-sensitive. Unlike in R, the column names in SQL aren't case-sensitive. So if we were to type "SELECT" as "select" or "Description" as "dEsCrIpTiOn", the SQL query would do the same thing. However, the tradition is to type SQL keywords in all uppercase to make it easier to distinguish them from table and column names.

The `SELECT` clause tells SQL which columns in particular you would like to see. The `FROM` clause simply tells SQL from which table it should pull the data. In this query, we are interested in only the `ID` and `Description` columns.

```
res <- dbSendQuery(con, "
                    SELECT ID, Description
                    FROM crime")
fetch(res, n = 10) # just the first 10 rows
dbClearResult(res)
```

```
         ID                  Description
```

```
1  10000092          AGGRAVATED: HANDGUN
2  10000094             PAROLE VIOLATION
3  10000095      DOMESTIC BATTERY SIMPLE
4  10000096                       SIMPLE
5  10000097               ARMED: HANDGUN
6  10000098                       SIMPLE
7  10000099      DOMESTIC BATTERY SIMPLE
8  10000100      DOMESTIC BATTERY SIMPLE
9  10000101 POSS: CANNABIS 30GMS OR LESS
10 10000104                       SIMPLE
```

Here, we set `n=10` so the first 10 rows are displayed. By convention, setting `n=-1`, will display all your rows. Really large SQL queries can be memory-intensive. So if your dataset is over 25 lines long (which it probably is.....that's why you're using SQL!), you have to make sure that you set the value in the fetch line to something reasonable to display.

However, suppose that your dataset is over 1 million rows, and you want to work with all of them. You can set the `fetch` line to something like `mydata <- fetch(res, n=-1)`.

`dbClearResult(res)` tells SQLite that we are all done with this query. We've retrieved the first 10 lines. SQLite is standing by with another 7 million rows to show us, but `dbClearResult(res)` tells SQLite that we are no longer interested in this query and it can clear out whatever it has stored for us.

In the previous SQL query we just asked for `ID` and `Description`. Typing out all of the column names would be tiresome, so SQL lets you use a `*` to select all the columns. If we want to look at the first 10 rows but all of the columns, we would use this query:

```
res <- dbSendQuery(con, "
                   SELECT *
                   FROM crime")
fetch(res, n = 10) # just the first 10 rows
dbClearResult(res)
```

```
         ID Case.Number                  Date                  Block IUCR
1  10000092    HY189866 03/18/2015 07:44:00 PM        047XX W OHIO ST 041A
2  10000094    HY190059 03/18/2015 11:00:00 PM 066XX S MARSHFIELD AVE 4625
3  10000095    HY190052 03/18/2015 10:45:00 PM  044XX S LAKE PARK AVE 0486
4  10000096    HY190054 03/18/2015 10:30:00 PM   051XX S MICHIGAN AVE 0460
5  10000097    HY189976 03/18/2015 09:00:00 PM       047XX W ADAMS ST 031A
6  10000098    HY190032 03/18/2015 10:00:00 PM    049XX S DREXEL BLVD 0460
7  10000099    HY190047 03/18/2015 11:00:00 PM      070XX S MORGAN ST 0486
8  10000100    HY189988 03/18/2015 09:35:00 PM    042XX S PRAIRIE AVE 0486
9  10000101    HY190020 03/18/2015 10:09:00 PM    036XX S WOLCOTT AVE 1811
10 10000104    HY189964 03/18/2015 09:25:00 PM    097XX S PRAIRIE AVE 0460
    Primary.Type                 Description    Location.Description
1        BATTERY          AGGRAVATED: HANDGUN                  STREET
2  OTHER OFFENSE             PAROLE VIOLATION                  STREET
3        BATTERY      DOMESTIC BATTERY SIMPLE               APARTMENT
4        BATTERY                       SIMPLE               APARTMENT
```

```
5        ROBBERY                ARMED: HANDGUN                SIDEWALK
6        BATTERY                       SIMPLE               APARTMENT
7        BATTERY      DOMESTIC BATTERY SIMPLE               APARTMENT
8        BATTERY      DOMESTIC BATTERY SIMPLE               APARTMENT
9      NARCOTICS POSS: CANNABIS 30GMS OR LESS                  STREET
10       BATTERY                       SIMPLE RESIDENCE PORCH/HALLWAY
   Arrest Domestic Beat District Ward Community.Area FBI.Code X.Coordinate
1   false    false 1111      011   28             25      04B      1144606
2    true    false  725      007   15             67       26      1166468
3   false     true  222      002    4             39      08B      1185075
4   false    false  225      002    3             40      08B      1178033
5   false    false 1113      011   28             25       03      1144920
6   false    false  223      002    4             39      08B      1183018
7   false     true  733      007   17             68      08B      1170859
8   false     true  213      002    3             38      08B      1178746
9    true    false  912      009   11             59       18      1164279
10  false    false  511      005    6             49      08B      1179637
   Y.Coordinate Year            Updated.On Latitude Longitude
1       1903566 2015 02/10/2018 03:50:01 PM 41.89140 -87.74438
2       1860715 2015 02/10/2018 03:50:01 PM 41.77337 -87.66532
3       1875622 2015 02/10/2018 03:50:01 PM 41.81386 -87.59664
4       1870804 2015 02/10/2018 03:50:01 PM 41.80080 -87.62262
5       1898709 2015 02/10/2018 03:50:01 PM 41.87806 -87.74335
6       1872537 2015 02/10/2018 03:50:01 PM 41.80544 -87.60428
7       1858210 2015 02/10/2018 03:50:01 PM 41.76640 -87.64930
8       1876914 2015 02/10/2018 03:50:01 PM 41.81755 -87.61982
9       1880656 2015 02/10/2018 03:50:01 PM 41.82814 -87.67278
10      1840444 2015 02/10/2018 03:50:01 PM 41.71745 -87.61766
                           Location
1  "(41.891398861, -87.744384567)"\r
2  "(41.773371528, -87.665319468)"\r
3   "(41.81386068, -87.596642837)"\r
4  "(41.800802415, -87.622619343)"\r
5  "(41.878064761, -87.743354013)"\r
6  "(41.805443345, -87.604283976)"\r
7  "(41.766402779, -87.649296123)"\r
8  "(41.817552577, -87.619818523)"\r
9  "(41.828138428, -87.672782106)"\r
10  "(41.71745472, -87.617663257)"\r
```

Just as `SELECT` filters the columns, the `WHERE` clause filters the rows. Note the use of `AND` and `OR` in the `WHERE` clause. As you might intuitively guess, `AND` and `OR` are logical operators that help us further filter our rows. Here we select three columns: `ID`, `Description`, and `Location.Description`. Also, we want only rows where * the value in the `Beat` column is "611" * the value in the `Arrest` column is "true" * the value in the `ICUR` column is either "0486" or "0498"

Importantly, note the use of single (not double) quotation marks in the `WHERE` line. The reason is that if we used double quotes, then R will think that the double quote signals the end of the query. Also note that `Location.Description` has a period in its name. The period has a special meaning

in SQL that we will discuss later. We use the square brakets around the name to "protect" the column name, telling SQL to treat it as a column name.

Also, note how we set the `fetch()` line to the variable `a`.

```
res <- dbSendQuery(con, "
                    SELECT ID, Description, [Location.Description]
                    FROM crime
                    WHERE ((Beat=611) AND
                          (Arrest='true')) AND
                          ((IUCR='0486') OR (IUCR='0498'))")
a <- fetch(res, n = -1) # all the rows
dbClearResult(res)
# show the first few rows of the results
a[1:3,]
```

```
        ID             Description Location.Description
1 10011764 DOMESTIC BATTERY SIMPLE            APARTMENT
2 10019667 DOMESTIC BATTERY SIMPLE               STREET
3 10046813 DOMESTIC BATTERY SIMPLE            APARTMENT
```

Exercises

1. Select records from Beat 234
2. Select Beat, District, Ward, and Community Area for all "ASSAULT"s. Remember that, since `Primary.Type` has a period in its name, you need to use square brackets like `[Primary.Type]`
3. Select records on assaults from Beat 234
4. Make a table in R of the number of assaults (IUCR 0560) by Ward

More SQL clauses

We've already covered SQL clauses `SELECT`, `WHERE`, and `FROM`. The SQL function `COUNT(*)` and `GROUP BY` are also very useful. For example, the following query counts how many assaults (IUCR 0560) occurred by ward. `COUNT()` is a SQL "aggregate" function, a function that performs a calculation on a group of values and returns a single number. Other SQL aggregate functions include `AVG()`, `MIN()`, `MAX()`, and `SUM()`. This query will group all the records by `Ward` and then apply the aggregate function `COUNT()` and report that value in a column called `crimecount`.

```
res <- dbSendQuery(con, "
   SELECT COUNT(*) AS crimecount,
          Ward
   FROM crime
   WHERE IUCR='0560'
   GROUP BY Ward")
a <- fetch(res, n = -1)
```

```
dbClearResult(res)

print(a)
```

```
   crimecount Ward
1       29468
2        3973    1
3        5863   10
4        3886   11
5        3095   12
6        2917   13
7        3004   14
8        7899   15
9        7930   16
10      10152   17
11       4538   18
12       2631   19
13       9424    2
14       9714   20
15       8104   21
16       3156   22
17       3045   23
18       9227   24
19       3508   25
20       4950   26
21       7656   27
22      10688   28
23       6536   29
24       8669    3
25       3439   30
26       3405   31
27       2440   32
28       2173   33
29       8732   34
30       3429   35
31       2515   36
32       6722   37
33       2580   38
34       2085   39
35       5489    4
36       2625   40
37       2290   41
38       6828   42
39       1618   43
40       2140   44
41       2600   45
42       3656   46
```

```
43       1959   47
44       2669   48
45       3688   49
46       6842    5
47       2392   50
48       8907    6
49       8321    7
50       8089    8
51       8343    9
```

The `GROUP BY` clause is critical. If you forget it then the result is not well defined. That is, different implementations of SQL may produce different results. The rule you should remember is that "every non-aggregate column in the `SELECT` clause should appear in the `GROUP BY` clause." Here `Ward` is not part of the aggregate function `COUNT()` so it must appear in the `GROUP BY` clause.

Practice exercises

5. Count the number of crimes by PrimaryType
6. Count the number of crimes resulting in arrest
7. Count the number of crimes by LocationDescription. LocationDescription is the variable that tells us where (e.g., a parking lot, a barbershop, a fire station, a CTA train, or a motel) a crime occurred

More SQL

`MAX`, `MIN`, `SUM`, `AVG` are common (and useful) aggregating functions. The `ORDER BY` clause sorts the results for us. It's the SQL version of the `sort()` command. Here is an illustration that gives the range of beat numbers in each policing district.

```
res <- dbSendQuery(con, "
   SELECT MIN(Beat) AS min_beat,
          MAX(Beat) AS max_beat,
          District
   FROM crime
   GROUP BY District
   ORDER BY District")
fetch(res, n = -1)
dbClearResult(res)
```

```
   min_beat max_beat District
1       124     2535
2       111     2515      001
3       211     2133      002
4       310     2132      003
5       324     2221      004
```

```
6       333     2233      005
7       123     2424      006
8       233     2431      007
9       333     2411      008
10      134     2331      009
11      133     2534      010
12      831     2535      011
13      111     2525      012
14      411     2535      014
15      726     2533      015
16      811     2521      016
17      813     2523      017
18      111     2514      018
19      112     2533      019
20      112     2433      020
21     2112     2112      021
22      214     2234      022
23      123     2433      024
24     1011     2535      025
25      124     2535      031
```

Remember that the `GROUP BY` clause should include every element of the `SELECT` clause that is not involved with an aggregate function. We have `MIN()` and `MAX()` operating on `Beat`, but `District` is on its own and should be placed in the `GROUP BY` clause.

Let's look at our `Latitude` and `Longitude` columns (which, as we find in a subsequent section, will be extremely useful for mapping data points). The following query will give unexpected results.

```
res <- dbSendQuery(con, "
   SELECT MIN(Latitude)  AS min_lat,
          MAX(Latitude)  AS max_lat,
          MIN(Longitude) AS min_lon,
          MAX(Longitude) AS max_lon,
          District
   FROM crime
   GROUP BY District
   ORDER BY District")
fetch(res, n = -1)
```

```
Warning in result_fetch(res@ptr, n = n): Column `max_lat`: mixed type,
first seen values of type real, coercing other values of type string
```

```
Warning in result_fetch(res@ptr, n = n): Column `max_lon`: mixed type,
first seen values of type real, coercing other values of type string
```

```
dbClearResult(res)
```

```
    min_lat  max_lat   min_lon   max_lon District
1  41.69991 42.00030 -87.87742 -87.59533
2  36.61945  0.00000 -91.68657   0.00000      001
3  36.61945  0.00000 -91.68657   0.00000      002
```

```
4  36.61945  0.00000 -91.68657   0.00000      003
5  36.61945  0.00000 -91.68657   0.00000      004
6  36.61945  0.00000 -91.68657   0.00000      005
7  36.61945  0.00000 -91.68657   0.00000      006
8  36.61945  0.00000 -91.68657   0.00000      007
9  36.61945  0.00000 -91.68657   0.00000      008
10 36.61945  0.00000 -91.68657   0.00000      009
11 36.61945  0.00000 -91.68657   0.00000      010
12 36.61945  0.00000 -91.68657   0.00000      011
13 36.61945  0.00000 -91.68657   0.00000      012
14 36.61945  0.00000 -91.68657   0.00000      014
15 36.61945  0.00000 -91.68657   0.00000      015
16 36.61945  0.00000 -91.68657   0.00000      016
17 36.61945  0.00000 -91.68657   0.00000      017
18 36.61945  0.00000 -91.68657   0.00000      018
19 41.80933  0.00000 -87.76791   0.00000      019
20 41.79145  0.00000 -87.76303   0.00000      020
21 41.83790 41.83790 -87.62192 -87.62192      021
22 36.61945  0.00000 -91.68657   0.00000      022
23 36.61945  0.00000 -91.68657   0.00000      024
24 36.61945  0.00000 -91.68657   0.00000      025
25 41.69165 42.01939 -87.90647 -87.53528      031
```

The first problem is that we have some rows with blank values in `Longitude` and `Latitude`. Here are some of them.

```
fetch(dbSendQuery(con,"SELECT * FROM crime WHERE Longitude=''"), n=3)
dbClearResult(res)
```

```
Warning: Expired, result set already closed
```

```
        ID Case.Number                   Date                 Block IUCR
1 10581023    HZ329792 09/01/2014 08:00:00 AM       0000X E LAKE ST 1140
2  4755307    HM367521 01/23/2002 12:00:00 AM 077XX S GREENWOOD AVE 0840
3  4757173    HM368193 05/22/2006 02:30:00 PM    074XX N ROGERS AVE 0910
         Primary.Type                  Description
1  DECEPTIVE PRACTICE                  EMBEZZLEMENT
2               THEFT FINANCIAL ID THEFT: OVER $300
3 MOTOR VEHICLE THEFT                   AUTOMOBILE
           Location.Description Arrest Domestic Beat District Ward
1                         OTHER   true    false  111      001   42
2                     APARTMENT  false    false  624      006    8
3 PARKING LOT/GARAGE(NON.RESID.)  false    false 2422      024   49
  Community.Area FBI.Code X.Coordinate Y.Coordinate Year
1             32       12                           2014
2             69       06                           2002
3              1       07                           2006
              Updated.On Latitude Longitude Location
1 03/01/2018 03:52:35 PM                          \r
```

```
2 08/17/2015 03:03:40 PM                                \r
3 08/17/2015 03:03:40 PM                                \r
```

Note the `X.Coordinate` and the `Y.Coordinate` columns. They should give the location per the State Plane Illinois East NAD 1983 projection, but in these rows they are empty.

The second problem is that there are minimum latitudes of 36.61945 and minimum longitudes of -91.68657. That's near the Missouri/Arkansas border 500 miles south of Chicago!

```
fetch(dbSendQuery(con,"SELECT * FROM crime where Latitude<36.61946"), n=3)
```

```
Warning: Closing open result set, pending rows
```

```
dbClearResult(res)
```

```
Warning: Expired, result set already closed
```

```
   ID Case.Number                   Date              Block IUCR
1 757     G209405 04/12/2001 09:32:00 PM  056XX S NORMAL AV 0110
2 808     G259321 05/06/2001 01:30:00 AM      020XX W 55 ST 0110
3 937     G411262 07/15/2001 12:34:00 AM 030XX S HARDING ST 0110
  Primary.Type         Description Location.Description Arrest Domestic
1     HOMICIDE FIRST DEGREE MURDER               STREET  false    false
2     HOMICIDE FIRST DEGREE MURDER                 AUTO   true    false
3     HOMICIDE FIRST DEGREE MURDER               STREET  false    false
  Beat District Ward Community.Area FBI.Code X.Coordinate Y.Coordinate
1  711      007                          01A            0            0
2  915      009                          01A            0            0
3 1031      010                          01A            0            0
  Year             Updated.On Latitude Longitude
1 2001 05/03/2018 03:49:53 PM 36.61945 -91.68657
2 2001 08/17/2015 03:03:40 PM 36.61945 -91.68657
3 2001 05/03/2018 03:49:53 PM 36.61945 -91.68657
                           Location
1 "(36.619446395, -91.686565684)"\r
2 "(36.619446395, -91.686565684)"\r
3 "(36.619446395, -91.686565684)"\r
```

Note that missing `X.Coordinate` and `Y.Coordinate` are getting mapped to some place far from Chicago.

We can tell SQLite to make the empty or missing values `NULL`, a more proper way to encode that these rows have missing coordinates. The `UPDATE` clause edits our table. R will read in `NULL` values as `NA`. After we do the update, we can rerun the `MIN()`, `MAX()` query. The dataset also has some latitudes and longitudes that are very close to 0 (and Chicago is quite far from the equator), but not exactly 0. We can make those `NULL` as well. We're also going to fix the `X.Coordinate` and `Y.Coordinate` columns.

```
res <- dbSendQuery(con, "
   UPDATE crime SET Latitude=NULL
   WHERE (Latitude='') OR (ABS(Latitude-0.0) < 0.01) OR (Latitude < 36.7)")
```

```
Warning: Closing open result set, pending rows
```

```
dbClearResult(res)
res <- dbSendQuery(con, "
   UPDATE crime SET Longitude=NULL
   WHERE (Longitude='') OR (ABS(Longitude-0.0) < 0.01) OR (Longitude < -91.6)")
dbClearResult(res)
res <- dbSendQuery(con, "
   UPDATE crime SET [X.Coordinate]=NULL
   WHERE ([X.Coordinate]='') OR ([X.Coordinate]=0)")
dbClearResult(res)
res <- dbSendQuery(con, "
   UPDATE crime SET [Y.Coordinate]=NULL
   WHERE ([Y.Coordinate]='') OR ([Y.Coordinate]=0)")
dbClearResult(res)
```

Let's rerun that query and check that we get more sensible results.

```
res <- dbSendQuery(con, "
   SELECT MIN(Latitude)  AS min_lat,
          MAX(Latitude)  AS max_lat,
          MIN(Longitude) AS min_lon,
          MAX(Longitude) AS max_lon,
          District
   FROM crime
   GROUP BY District
   ORDER BY District")
fetch(res, n = -1)
dbClearResult(res)
```

```
    min_lat  max_lat   min_lon   max_lon District
1  41.69991 42.00030 -87.87742 -87.59533
2  41.72827 41.98740 -87.84349 -87.54925      001
3  41.73298 41.97608 -87.70277 -87.56954      002
4  41.71424 41.79946 -87.73941 -87.55258      003
5  41.64467 41.79220 -87.72436 -87.52453      004
6  41.64459 41.88693 -87.73145 -87.54348      005
7  41.69249 42.01876 -87.77138 -87.55810      006
8  41.66806 42.01369 -87.68723 -87.57906      007
9  41.73453 42.01765 -87.80161 -87.55239      008
10 41.79018 41.97645 -87.71397 -87.58822      009
11 41.68357 41.94304 -87.74364 -87.61895      010
12 41.77163 41.90624 -87.76332 -87.62328      011
13 41.68544 41.96539 -87.76321 -87.60502      012
14 41.77688 42.01938 -87.80222 -87.65657      014
15 41.76641 41.94234 -87.77534 -87.63087      015
16 41.78464 42.01938 -87.93432 -87.58256      016
17 41.77950 42.01390 -87.75780 -87.66131      017
18 41.85926 41.96879 -87.76313 -87.60136      018
19 41.80933 41.98397 -87.76791 -87.58775      019
```

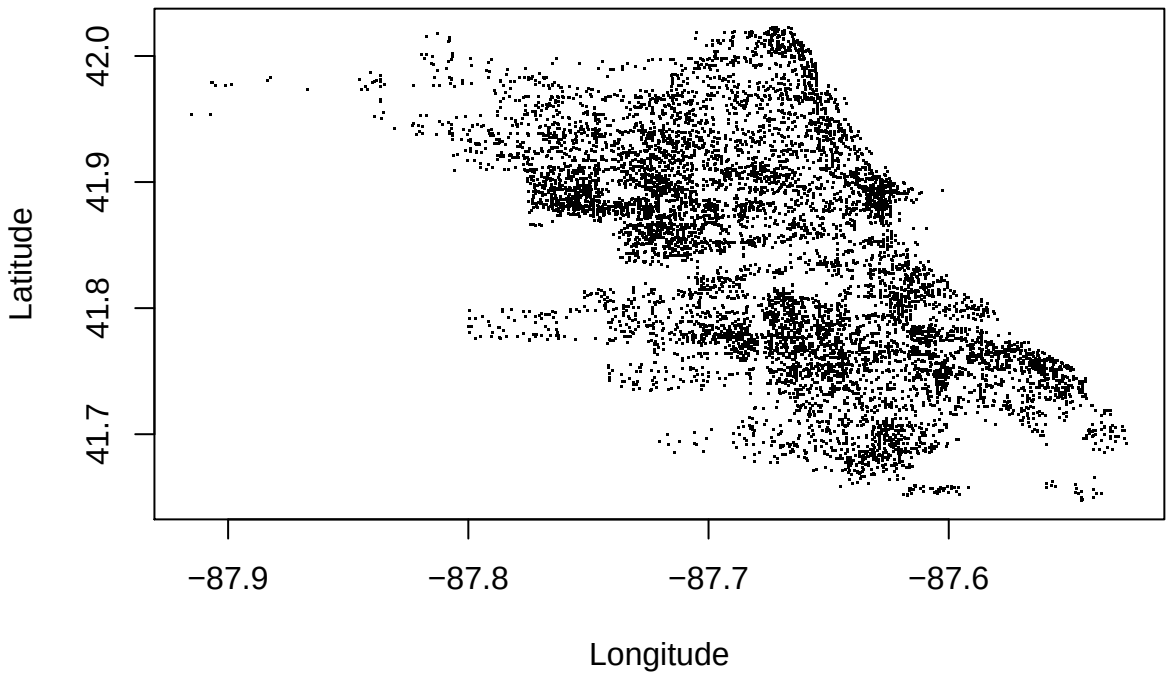
```
20 41.79145 42.00458 -87.76303 -87.62992      020
21 41.83790 41.83790 -87.62192 -87.62192      021
22 41.67709 41.85572 -87.74328 -87.58965      022
23 41.75884 42.02291 -87.79757 -87.62545      024
24 41.83930 41.94586 -87.81648 -87.64093      025
25 41.69165 42.01939 -87.90647 -87.53528      031
```

Now we have results that are more in line with where Chicago actually is. Make it a habit to do some checks of your data before doing too much analysis.

And what city does the following plot have the shape of? Let's plot the location of these crimes. Plotting all 7 million would be overkill, so let's take a random sample of 10,000 crimes. Here's a SQL query that will do that. It uses some tricks we'll learn more about later including the use of `IN`, the use of subqueries (a query within a query), and `LIMIT`. Does the shape of the plot look right?

```
res <- dbSendQuery(con, "
   SELECT Longitude, Latitude
   FROM crime
   WHERE id IN (SELECT id FROM crime ORDER BY RANDOM() LIMIT 10000)")
a <- fetch(res, n = -1)
dbClearResult(res)

plot(Latitude~Longitude, data=a, pch=".", xlab="Longitude", ylab="Latitude")
```



Practice exercises

8. Plot the location of all "ASSAULT"s for Ward 22
9. What is the most common (Lat,Long) for assaults in Ward 22? Add the point to your plot using the points() command. The points() command is new. It is simply a function that draws a point (or sequence of points) at the specified coordinates

Solutions to the exercises

1. Select records from Beat 234

```
res <- dbSendQuery(con, "
          SELECT *
          FROM crime
          WHERE ((Beat=234))")
a <- fetch(res, n = -1)
dbClearResult(res)
```

2. Select Beat, District, Ward, and Community Area for all "ASSAULT"s

```
res <- dbSendQuery(con, "
                  SELECT Beat, District, Ward, [Community.Area], [Primary.Type]
                  FROM crime
                  WHERE (([Primary.Type]='ASSAULT'))")
a <- fetch(res, n = -1)
dbClearResult(res)
```

3. Select records on assaults from Beat 234

```
res <- dbSendQuery(con, "
                    SELECT *
                    FROM crime
                    WHERE ((Beat=234) AND ([Primary.Type]='ASSAULT'))")
a <- fetch(res, n = -1)
dbClearResult(res)
```

4. Make a table in R of the number of assaults (IUCR 0560) by Ward

```
# system.time() reports how long it takes to run the SQL query
system.time(
{
   res <- dbSendQuery(con, "
                      SELECT *
                      FROM crime
                      WHERE ((IUCR='0560') AND ([Primary.Type]='ASSAULT'))")
   a <- fetch(res, n = -1)
})
```

```
dbClearResult(res)
table(a$Ward)
```

```
   user  system elapsed
   2.58    4.00    6.58

          1    10    11    12    13    14    15    16    17    18    19
29468  3973  5863  3886  3095  2917  3004  7899  7930 10152  4538  2631
    2    20    21    22    23    24    25    26    27    28    29     3
 9424  9714  8104  3156  3045  9227  3508  4950  7656 10688  6536  8669
   30    31    32    33    34    35    36    37    38    39     4    40
 3439  3405  2440  2173  8732  3429  2515  6722  2580  2085  5489  2625
   41    42    43    44    45    46    47    48    49     5    50     6
 2290  6828  1618  2140  2600  3656  1959  2669  3688  6842  2392  8907
    7     8     9
 8321  8089  8343
```

Or, we could also try selecting all the IUCR codes and ward and then subsetting the data through R.

```
system.time(
{
   res <- dbSendQuery(con, "
                      SELECT IUCR, Ward, [Primary.Type]
                      FROM crime")
   data <- fetch(res, n = -1)
   data <- subset(data, Primary.Type=="ASSAULT" & IUCR=="0560")
})
dbClearResult(res)
```

```
   user  system elapsed
   5.20    4.19    9.39
```

5. Count the number of crimes by PrimaryType

```
res <- dbSendQuery(con, "
              SELECT COUNT(*) AS count, [Primary.Type]
              FROM crime
              GROUP BY [Primary.Type]")
fetch(res, n = -1)
dbClearResult(res)
```

```
     count                       Primary.Type
1    11019                             ARSON
2   411720                            ASSAULT
3  1215592                            BATTERY
4   383891                           BURGLARY
5      222 CONCEALED CARRY LICENSE VIOLATION
6    26455               CRIM SEXUAL ASSAULT
7   762125                   CRIMINAL DAMAGE
```

```
8   191154                 CRIMINAL TRESPASS
9   255797               DECEPTIVE PRACTICE
10       1                DOMESTIC VIOLENCE
11   14337                         GAMBLING
12    9207                         HOMICIDE
13      42                HUMAN TRAFFICKING
14   14711 INTERFERENCE WITH PUBLIC OFFICER
15    3862                      INTIMIDATION
16    6616                        KIDNAPPING
17   13967             LIQUOR LAW VIOLATION
18  310921              MOTOR VEHICLE THEFT
19  706464                         NARCOTICS
20      38                   NON - CRIMINAL
21     152                     NON-CRIMINAL
22       8 NON-CRIMINAL (SUBJECT SPECIFIED)
23     547                         OBSCENITY
24   44552       OFFENSE INVOLVING CHILDREN
25     123         OTHER NARCOTIC VIOLATION
26  413056                     OTHER OFFENSE
27   68065                      PROSTITUTION
28     158                  PUBLIC INDECENCY
29   47349           PUBLIC PEACE VIOLATION
30      23                         RITUALISM
31  252413                           ROBBERY
32   24620                       SEX OFFENSE
33    3310                          STALKING
34 1395322                             THEFT
35   68782                 WEAPONS VIOLATION
```

6. Count the number of crimes resulting in arrest

```
res <- dbSendQuery(con, "
                   SELECT COUNT(*) AS count, [Primary.Type]
                   FROM crime
                   WHERE Arrest='true'
                   GROUP BY [Primary.Type]")
fetch(res, n = -1)
dbClearResult(res)
```

```
    count                      Primary.Type
1    1441                             ARSON
2   95637                           ASSAULT
3  277322                           BATTERY
4   21957                          BURGLARY
5     205 CONCEALED CARRY LICENSE VIOLATION
6    4166              CRIM SEXUAL ASSAULT
7   53857                  CRIMINAL DAMAGE
8  140515                CRIMINAL TRESPASS
9   43741              DECEPTIVE PRACTICE
```

```
10      1                 DOMESTIC VIOLENCE
11  14233                          GAMBLING
12   4343                          HOMICIDE
13      6                 HUMAN TRAFFICKING
14  13476  INTERFERENCE WITH PUBLIC OFFICER
15    690                       INTIMIDATION
16    739                         KIDNAPPING
17  13842              LIQUOR LAW VIOLATION
18  28496               MOTOR VEHICLE THEFT
19 702233                          NARCOTICS
20      6                   NON - CRIMINAL
21      8                     NON-CRIMINAL
22      3  NON-CRIMINAL (SUBJECT SPECIFIED)
23    453                          OBSCENITY
24   9488        OFFENSE INVOLVING CHILDREN
25     88          OTHER NARCOTIC VIOLATION
26  73400                     OTHER OFFENSE
27  67798                      PROSTITUTION
28    157                  PUBLIC INDECENCY
29  30434            PUBLIC PEACE VIOLATION
30      3                          RITUALISM
31  24491                            ROBBERY
32   7617                       SEX OFFENSE
33    536                          STALKING
34 167223                             THEFT
35  54982                 WEAPONS VIOLATION
```

Or, if we weren't interested in differentiating based on the `Primary.Type`, we could simply do the following:

```
res <- dbSendQuery(con, "
                   SELECT COUNT(*) AS count
                   FROM crime
                   WHERE Arrest='true'")
fetch(res, n = -1)
dbClearResult(res)
```

```
    count
1 1853587
```

7. Count the number of crimes by LocationDescription. LocationDescription is the variable that tells us where (e.g., a parking lot, a barbershop, a fire station, a CTA train, or a motel) a crime occurred

```
res <- dbSendQuery(con, "
SELECT COUNT(*) AS count,
                   [Location.Description]
                   FROM crime
                   GROUP BY [Location.Description]")
fetch(res, n = -1)
```

```
dbClearResult(res)
```

	count	Location.Description
1	3676	
2	4	"CTA ""L"" PLATFORM"
3	2	"CTA ""L"" TRAIN"
4	13218	"SCHOOL, PRIVATE, BUILDING"
5	4026	"SCHOOL, PRIVATE, GROUNDS"
6	141371	"SCHOOL, PUBLIC, BUILDING"
7	28703	"SCHOOL, PUBLIC, GROUNDS"
8	128	"VEHICLE - OTHER RIDE SHARE SERVICE (E.G., UBER, LYFT)"
9	11067	ABANDONED BUILDING
10	593	AIRCRAFT
11	731	AIRPORT BUILDING NON-TERMINAL - NON-SECURE AREA
12	505	AIRPORT BUILDING NON-TERMINAL - SECURE AREA
13	674	AIRPORT EXTERIOR - NON-SECURE AREA
14	250	AIRPORT EXTERIOR - SECURE AREA
15	663	AIRPORT PARKING LOT
16	1545	AIRPORT TERMINAL LOWER LEVEL - NON-SECURE AREA
17	571	AIRPORT TERMINAL LOWER LEVEL - SECURE AREA
18	75	AIRPORT TERMINAL MEZZANINE - NON-SECURE AREA
19	610	AIRPORT TERMINAL UPPER LEVEL - NON-SECURE AREA
20	3746	AIRPORT TERMINAL UPPER LEVEL - SECURE AREA
21	84	AIRPORT TRANSPORTATION SYSTEM (ATS)
22	856	AIRPORT VENDING ESTABLISHMENT
23	16091	AIRPORT/AIRCRAFT
24	149019	ALLEY
25	748	ANIMAL HOSPITAL
26	686185	APARTMENT
27	1765	APPLIANCE STORE
28	7885	ATHLETIC CLUB
29	7465	ATM (AUTOMATIC TELLER MACHINE)
30	1097	AUTO
31	147	AUTO / BOAT / RV DEALERSHIP
32	27116	BANK
33	1	BANQUET HALL
34	35239	BAR OR TAVERN
35	12	BARBER SHOP/BEAUTY SALON
36	7624	BARBERSHOP
37	30	BASEMENT
38	650	BOAT/WATERCRAFT
39	647	BOWLING ALLEY
40	371	BRIDGE
41	2728	CAR WASH
42	345	CEMETARY
43	35782	CHA APARTMENT
44	3	CHA BREEZEWAY

45	2	CHA ELEVATOR
46	39	CHA GROUNDS
47	35	CHA HALLWAY
48	24782	CHA HALLWAY/STAIRWELL/ELEVATOR
49	6	CHA LOBBY
50	40	CHA PARKING LOT
51	55270	CHA PARKING LOT/GROUNDS
52	3	CHA PLAY LOT
53	8	CHA STAIRWELL
54	6	CHURCH
55	2	CHURCH PROPERTY
56	14681	CHURCH/SYNAGOGUE/PLACE OF WORSHIP
57	1	CLEANERS/LAUNDROMAT
58	4705	CLEANING STORE
59	15	CLUB
60	3	COACH HOUSE
61	1026	COIN OPERATED MACHINE
62	5561	COLLEGE/UNIVERSITY GROUNDS
63	1340	COLLEGE/UNIVERSITY RESIDENCE HALL
64	48546	COMMERCIAL / BUSINESS OFFICE
65	12753	CONSTRUCTION SITE
66	15302	CONVENIENCE STORE
67	2	COUNTY JAIL
68	476	CREDIT UNION
69	21298	CTA BUS
70	5893	CTA BUS STOP
71	9826	CTA GARAGE / OTHER PROPERTY
72	35934	CTA PLATFORM
73	4	CTA PROPERTY
74	3508	CTA STATION
75	90	CTA TRACKS - RIGHT OF WAY
76	23252	CTA TRAIN
77	10626	CURRENCY EXCHANGE
78	2684	DAY CARE CENTER
79	962	DELIVERY TRUCK
80	82146	DEPARTMENT STORE
81	17	DRIVEWAY
82	19321	DRIVEWAY - RESIDENTIAL
83	30228	DRUG STORE
84	7	DUMPSTER
85	1	ELEVATOR
86	1	EXPRESSWAY EMBANKMENT
87	2	FACTORY
88	6713	FACTORY/MANUFACTURING BUILDING
89	2	FARM
90	761	FEDERAL BUILDING
91	996	FIRE STATION
92	379	FOREST PRESERVE

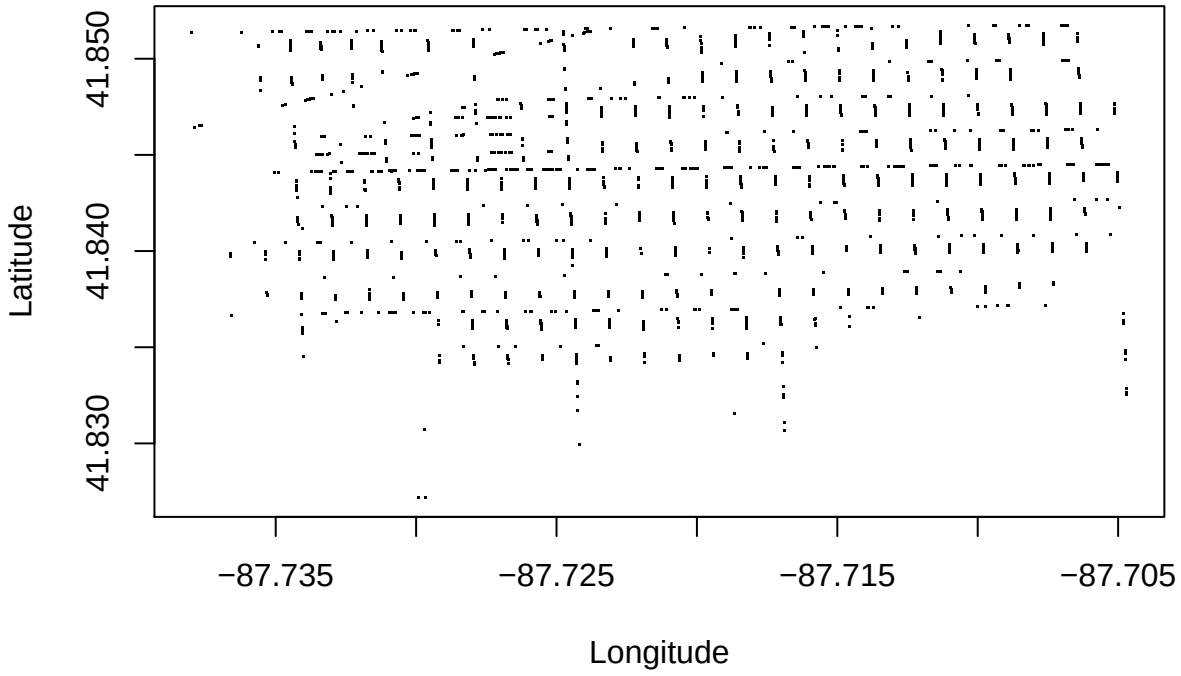
93	1	FUNERAL PARLOR
94	64	GANGWAY
95	52	GARAGE
96	11	GARAGE/AUTO REPAIR
97	70831	GAS STATION
98	43	GAS STATION DRIVE/PROP.
99	2	GOVERNMENT BUILDING
100	13929	GOVERNMENT BUILDING/PROPERTY
101	86195	GROCERY FOOD STORE
102	84	HALLWAY
103	993	HIGHWAY/EXPRESSWAY
104	1	HORSE STABLE
105	5	HOSPITAL
106	19998	HOSPITAL BUILDING/GROUNDS
107	18	HOTEL
108	27530	HOTEL/MOTEL
109	514	HOUSE
110	1024	JAIL / LOCK-UP FACILITY
111	1	JUNK YARD/GARBAGE DUMP
112	1	KENNEL
113	1	LAGOON
114	3	LAKE
115	1085	LAKEFRONT/WATERFRONT/RIVERBANK
116	2	LAUNDRY ROOM
117	5875	LIBRARY
118	7	LIQUOR STORE
119	1	LIVERY AUTO
120	2	LIVERY STAND OFFICE
121	1	LOADING DOCK
122	7000	MEDICAL/DENTAL OFFICE
123	5	MOTEL
124	2507	MOVIE HOUSE/THEATER
125	228	NEWSSTAND
126	5	NURSING HOME
127	13164	NURSING HOME/RETIREMENT HOME
128	14	OFFICE
129	252768	OTHER
130	2871	OTHER COMMERCIAL TRANSPORTATION
131	5677	OTHER RAILROAD PROP / TRAIN DEPOT
132	51555	PARK PROPERTY
133	152	PARKING LOT
134	191224	PARKING LOT/GARAGE(NON.RESID.)
135	509	PAWN SHOP
136	17171	POLICE FACILITY/VEH PARKING LOT
137	855	POOL ROOM
138	1	POOLROOM
139	281	PORCH
140	2	PRAIRIE

```
141       1                                  PUBLIC GRAMMAR SCHOOL
142       2                                     PUBLIC HIGH SCHOOL
143      13                                      RAILROAD PROPERTY
144 1127678                                               RESIDENCE
145  116365                                  RESIDENCE PORCH/HALLWAY
146  130214                                         RESIDENCE-GARAGE
147   67547                           RESIDENTIAL YARD (FRONT/BACK)
148  103254                                           RESTAURANT
149      70                                        RETAIL STORE
150       4                                               RIVER
151       4                                          RIVER BANK
152       2                                       ROOMING HOUSE
153     346                                    SAVINGS AND LOAN
154      11                                         SCHOOL YARD
155       3                                               SEWER
156  657450                                           SIDEWALK
157  116890                                   SMALL RETAIL STORE
158    4873                                 SPORTS ARENA/STADIUM
159      17                                           STAIRWELL
160 1750257                                              STREET
161      34                                              TAVERN
162   21639                                  TAVERN/LIQUOR STORE
163       6                                           TAXI CAB
164    7120                                            TAXICAB
165       3                                            TRAILER
166       8                                              TRUCK
167       1                                  TRUCKING TERMINAL
168     100                                         VACANT LOT
169   23676                                      VACANT LOT/LAND
170      83                                 VEHICLE - DELIVERY TRUCK
171     335                               VEHICLE - OTHER RIDE SERVICE
172  106422                                   VEHICLE NON-COMMERCIAL
173    5288                                       VEHICLE-COMMERCIAL
174       3            VEHICLE-COMMERCIAL - ENTERTAINMENT/PARTY BUS
175       4                        VEHICLE-COMMERCIAL - TROLLEY BUS
176      17                                           VESTIBULE
177    9112                                          WAREHOUSE
178       5                                        WOODED AREA
179     195                                               YARD
180       3                                               YMCA
```

8. Plot the location of all "ASSAULT"s for Ward 22

```
res <- dbSendQuery(con, "
                   SELECT Latitude, Longitude
                   FROM crime
                   WHERE [Primary.Type]='ASSAULT' AND Ward='22'")
a <- fetch(res, n = -1)
```

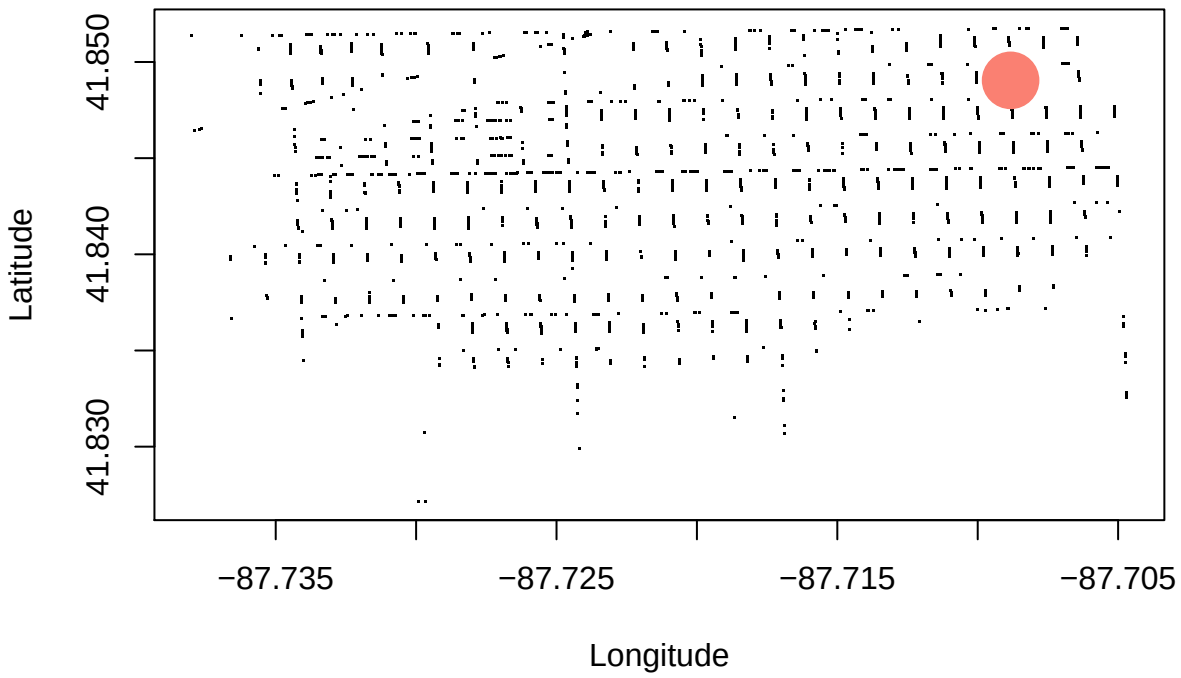
```
dbClearResult(res)
plot(Latitude~Longitude, data=a, pch=".")
```



9. What is the most common (Lat,Long) for assaults in Ward 22? Add the point to your plot using the points() command. The points() command is new. It is simply a function that draws a point (or sequence of points) at the specified coordinates

```
res <- dbSendQuery(con, "
        SELECT COUNT(*) as crimecount,
               Longitude, Latitude
        FROM crime
        WHERE [Primary.Type]='ASSAULT' and Ward='22'
        GROUP BY Longitude, Latitude")
b <- fetch(res, n=-1)
dbClearResult(res)

plot(Latitude~Longitude, data=a, pch=".")
points(b[which.max(b$crimecount), 2:3],
       pch=16,
       col="salmon",
       cex=4)
```




```
b[which.max(b$crimecount), 2:3]
```

```
     Longitude Latitude
2031 -87.70883 41.84905
```