Accessing Big Data

Big Data

In real life, folks often work with data sets that are big.

What do we mean by “big”?

For a single data set (table),

- Many variables (columns)
- Many observations (rows)

Data may also be big because it is comprised of many data sets (tables).

Working with Big Data

We want to avoid reading it all in at once!

This may mean:

- Accessing individual data sets (tables) without reading them all into memory
- Accessing parts (rows and/or columns) of an individual data set without reading it all in
- Generally, learning about a data set without reading any of it in

How Big is Big?

Let's consider a simple example of a matrix with numeric elements.

```r
bigmat <- matrix(rnorm(1000000*100),
                 nrow = 1000000,
                 ncol = 100)
```

We can use the `object.size` function combined with the `print` function to see how much space this matrix takes up in memory.

```r
print(object.size(bigmat), unit = "GB")
```

```
0.7 Gb
```

How Big is Too Big?

```r
bigmat <- matrix(rnorm(1000000*100),
                 nrow = 1000000,
                 ncol = 100)
```

How big can you make the matrix before you get an error?

Try fixing the number of columns and changing the number of rows, and then fixing the number of rows and changing the number of columns.

Ways to Interact with Big Data

Hopefully you're convinced that it's possible to run into data that's too big in real life!

This necessitates different ways of storing and interacting with data that let us store data in smaller pieces.

We'll talk about a two approaches:

- HDF5
- SQL

All require packages, which we may have more or less success installing.

Hierarchical Data Format Version 5

HDF5 stands for Hierarchical Data Format... Version 5.

It is a format used to store collections of files that are hierarchically arranged into groups, and is most appropriate for storing numeric data.

This file type is used for:

- Financial data
- Spatial data
- Imaging data
- Sequencing data

FYI: HDF5 files are sometimes known as netCDF-4 files.

HDF5 in R

Even though HDF5 files are generic and widely used, the preferred package for working with HDF5 files `rhdf5` is not available on `CRAN`.

We need to get it from Bioconductor instead.

```r
install.packages("BiocManager")
BiocManager::install("rhdf5")
```

HDF5 via an Example

NASA uses HDF5 files and makes a lot of their data publicly available: https://www.earthdata.nasa.gov.

We're going to use their VESDR data as an example, you can read about it here.

You can download the file from Canvas, `nasa.h5`.

Learning About The Data

```r
library(rhdf5)
nasa <- H5Fopen("~/Downloads/nasa.h5")
```

Learning About The Data

Printing `nasa` will provide information on the groups of objects that the file contains.

```
nasa
```

```
HDF5 FILE
        name /
    filename

    name     otype dclass dim
0 tile10 H5I_GROUP
1 tile11 H5I_GROUP
2 tile20 H5I_GROUP
3 tile21 H5I_GROUP
4 tile30 H5I_GROUP
5 tile31 H5I_GROUP
```

Learning About The Data

We can learn more about each group of objects by adding `&` and a name in single quotes after `nasa` and entering that into the console.

Type in the following in the console:

- `nasa&'tile10'`
- `nasa&'tile11'`
- `nasa&'tile20'`

Do these create objects in your environment? What do they tell you?

Learning About The Data

We can learn more about each individual object by adding another & and a name in single quotes and entering that into the console.

Type in the following in the console:

- `nasa&'tile10'&'01_LAI'`
- `nasa&'tile10'&'06_QA_VESDR'`
- `nasa&'tile11'&'01_LAI'`

Do these create objects in your environment? What do they tell you?

Loading in a Piece of Data

We can load in data by replacing each `&` with `$`.

Type in the following in the console:

- `lai1001 <- nasa$'tile10'$'01_LAI'`

What is `lai1001`?

We can also subset to specific rows and columns:

- `lai1001 <- nasa$'tile10'$'01_LAI'[1:5, 1:5]`

How is this `lai1001` different from the first?

Making Things More Efficient

Let's carefully examine the following

- `lai1001 <- nasa$'tile10'$'01_LAI'`

This actually loads in more data than we really need, and then subsets it.

This is wasteful and sometimes may not be feasible! Fortunately, we can rewrite things to directly retrieve the subsetted data.

- `lai1001 <- nasa$'tile10/01_LAI'`

Making Things More Efficient - Subsetting

We can also avoid reading an entire object into `R` and rather read in a subset of elements by combining subsetting and the `&` operator.

- `lai1001 <- (nasa&'tile10/01_LAI')[1:5, 1:5]`

Compare to `nasa$'tile10/01_LAI'[1:5, 1:5]`, which is faster?

Closing Things Up

Once we are done working with HDF5 files, we do need to take one additional step to close them.

```
h5closeAll()
```

Why Isn't HDF5 Enough?

- Text data
- Need for multiple users to access concurrently (sometimes)
- Language for manipulation is limited and not very readable

Enter SQL!

What is SQL?

SQL short for “Structured Query Language” and is pronounced “Sequel.”

It is a language that can be used across platforms and within `R` for interacting with databases management systems.

Note: SQL does not describe how the data is stored (that’s determined by the database management system), just how you access it.

We’re going to be agnostic and demonstrate SQL.

Using SQL

Using SQL has roughly two steps:

- Connecting with a database (varies)
- Querying a database (always the same)

We won't really delve into connecting with a database here because it's so heterogeneous.

The database may be stored online, on a server, or elsewhere, and you may need to work with the people who manage the data to figure out how to connect to it.

We'll work through example with an "SQLite" database that we'll download to our computer.

Wait - what is a database?

We've already been using a database when we learned about HFD5.

A database is a collection of tables

Tables are collections of records which share the same fields (variables)

Records are a collection of values for different fields (variables), with one value per field

In our HDF5 example, we had a **hierarchical** database that contained many tables of the same dimensions with the same (numeric) fields.

In general, databases are more flexible, they can contain

- Tables of different dimensions with different fields
- Non-numeric e.g. character fields

Connecting To Our Database

Our database is stored in `baseball.db` on Canvas. Go get it!

Some information about quantities in this database is available here: http://m.mlb.com/glossary/standard-stats.

To connect to this database we'll use the `DBI` and `RSQLite` libraries.

- We need `DBI` or something similar regardless of how our database is stored, to send SQL queries to our database
- We need `RSQLite` because our database is stored as an "SQLite" database

Installing and Loading DBI and RSQLite

Install them!

```r
install.packages("DBI")
install.packages("RSQLite")
```

Load them!

```r
library(DBI)
library(RSQLite)
```

Connecting To Our Database

The first step is to use the dbDriver function in the DBI library to indicate that we will be accessing a “SQLite” database.

```
drv <- dbDriver("SQLite")
```

The next step is to connect to the database by using the dbConnect function in the DBI library.

```
con <- dbConnect(drv, dbname = "~/Downloads/baseball.db")
```

Learning About Our Database

```r
file.info("baseball.db")
```

```
            size isdir mode               mtime               ctime
baseball.db    0 FALSE  644 2025-04-02 09:45:28 2025-04-02 09:45:29
                          atime uid gid     uname grname
baseball.db 2025-04-02 09:45:28 501  20 maryclare  staff
```

Learning About Our Database

```r
dbListTables(con)
```

```
 [1] "AllstarFull"           "Appearances"           "AwardsManagers"
 [4] "AwardsPlayers"         "AwardsShareManagers" "AwardsSharePlayers"
 [7] "Batting"               "BattingPost"           "Fielding"
[10] "FieldingOF"            "FieldingPost"          "HallOfFame"
[13] "Managers"              "ManagersHalf"          "Master"
[16] "Pitching"              "PitchingPost"          "Salaries"
[19] "Schools"               "SchoolsPlayers"        "SeriesPost"
[22] "Teams"                 "TeamsFranchises"       "TeamsHalf"
[25] "sqlite_sequence"       "xref_stats"
```

Learning About Our Database

```r
dbListFields(con, "Master")
```

```
 [1] "lahmanID"     "playerID"     "managerID"    "hofID"        "birth
 [6] "birthMonth"   "birthDay"     "birthCountry" "birthState"   "birth
[11] "deathYear"    "deathMonth"   "deathDay"     "deathCountry" "death
[16] "deathCity"    "nameFirst"    "nameLast"     "nameNote"     "nameG
[21] "nameNick"     "weight"       "height"       "bats"         "throw
[26] "debut"        "finalGame"    "college"      "lahman40ID"   "lahma
[31] "retroID"      "holtzID"      "bbrefID"
```

Importing Data From Our Database

To import data from our database, we need to indicate:

- Which table we want data from
- Which fields (variables) we are interested in

We use SQL to communicate what we want

```
db <- dbGetQuery(
  con,
  "SELECT playerID, yearID, AB, H, HR FROM Batting")
```

What Do We Get?

```r
str(db)
```

```
'data.frame':   93955 obs. of  5 variables:
 $ playerID: chr  "aardsda01" "aardsda01" "aardsda01" "aardsda01" ...
 $ yearID  : int  2004 2006 2007 2008 2009 1954 1955 1956 1957 1958 ...
 $ AB      : int  0 2 0 1 0 468 602 609 615 601 ...
 $ H       : int  0 0 0 0 0 131 189 200 198 196 ...
 $ HR      : int  0 0 0 0 0 13 27 26 44 30 ...
```

SQL Is Not Case Sensitive

```r
db <- dbGetQuery(
  con,
  "Select playerID, yearID, AB, h, HR FROM Batting")
```

That said - a common practice for readability is to type SQL commands (here, “Select” and “From”) in all caps like we did originally.

SQL is Order Sensitive

```
db <- dbGetQuery(
  con,
  "SELECT yearID, playerID, AB, H, HR FROM Batting")
```

SQL Allows Line Breaks

Across all coding languages, extremely long lines of code that require scrolling to view are discouraged.

```
db <- dbGetQuery(
  con,
  "SELECT
  playerID, yearID, AB, H, HR
  FROM
  Batting")
```

Getting All of the Fields (Variables)

```r
db <- dbGetQuery(con, "Select * FROM Batting")
```

Getting All of the Fields (Variables)

```r
str(db)
```

```
'data.frame':	93955 obs. of  24 variables:
 $ playerID : chr  "aardsda01" "aardsda01" "aardsda01" "aardsda01" ...
 $ yearID   : int  2004 2006 2007 2008 2009 1954 1955 1956 1957 1958 ..
 $ stint    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ teamID   : chr  "SFN" "CHN" "CHA" "BOS" ...
 $ lgID     : chr  "NL" "NL" "AL" "AL" ...
 $ G        : int  11 45 25 47 73 122 153 153 151 153 ...
 $ G_batting: int  11 43 2 5 3 122 153 153 151 153 ...
 $ AB       : int  0 2 0 1 0 468 602 609 615 601 ...
 $ R        : int  0 0 0 0 0 58 105 106 118 109 ...
 $ H        : int  0 0 0 0 0 131 189 200 198 196 ...
 $ 2B       : int  0 0 0 0 0 27 37 34 27 34 ...
 $ 3B       : int  0 0 0 0 0 6 9 14 6 4 ...
 $ HR       : int  0 0 0 0 0 13 27 26 44 30 ...
 $ RBI      : int  0 0 0 0 0 69 106 92 132 95 ...
 $ SB       : int  0 0 0 0 0 2 3 2 1 4 ...
 $ CS       : int  0 0 0 0 0 2 1 4 1 1 ...
 $ BB       : int  0 0 0 0 0 28 49 37 57 59 ...
 $ SO       : int  0 0 0 1 0 39 61 54 58 49 ...
 $ IBB      : int  0 0 0 0 0 NA 5 6 15 16 ...
 $ HBP      : int  0 0 0 0 0 3 3 2 0 1 ...
```

More Complicated Requests

SQL allows us to easily:

- Reorder the table before importing it into `R`
- Request a subset of records (rows/observations)

Requesting a Sorted Table

```
db <- dbGetQuery(
  con,
  "SELECT * FROM Salaries ORDER BY Salary")
```

Requesting a Sorted Table (Decreasing)

```r
db <- dbGetQuery(
  con,
  "SELECT * FROM Salaries ORDER BY Salary DESC")
```

Subsetting Based on Ordering

```r
db <- dbGetQuery(
  con,
  "SELECT * FROM Salaries
  ORDER BY Salary DESC
  LIMIT 5")
```

Peeking At the Data

We can also use LIMIT to just peek at the data if we don't assign values.

```
dbGetQuery(con, "SELECT * FROM Salaries LIMIT 5")
```

```
  yearID teamID lgID  playerID salary
1   1980    TOR   AL stiebda01  55000
2   1981    NYA   AL jacksre01 588000
3   1981    TOR   AL stiebda01  85000
4   1982    TOR   AL stiebda01 250000
5   1983    TOR   AL stiebda01 450000
```

Subsetting Based on Conditions

```r
db <- dbGetQuery(
  con,
  "SELECT PlayerID, yearID, AB, H
  FROM Batting
  WHERE AB > 100 AND H > 0")
```

Subsetting Based on Conditions

```
db <- dbGetQuery(
  con,
  "SELECT *
  FROM Master
  WHERE nameLast IN (\"Alou\", \"Griffey\")")
```

Subsetting Based on Conditions

```
db <- dbGetQuery(
  con,
  "SELECT *
  FROM Master
  WHERE nameLast LIKE '%riff%'")
```

What is LIKE doing?

- ‘%’ Matches everything before the %
- ’_’ Matches before and after the _, allows an arbitrary character in between
- ‘[]’ Matches before and after the [] sign, allows any character in the brackets in between
- ‘[^]’ Matches before and after the [] sign, allows any character expect those after ^ in the brackets in between
- ‘[-]’ Matches before and after the [] sign, allows any character between the characters separated by - in the brackets in between

Subsetting Based on Conditions

```
db <- dbGetQuery(
  con,
  "SELECT *
  FROM Master
  WHERE birthCountry == 'P.R.'
  AND
  birthYear LIKE '198%'")
```

Summarizing Data

```r
dbGetQuery(
  con,
  "SELECT MIN(AB), AVG(AB), MAX(AB) FROM Batting")
```

```
  MIN(AB)  AVG(AB) MAX(AB)
1       0 155.0852     716
```

Summarizing Data

```r
dbGetQuery(
  con,
  "SELECT COUNT(*) FROM Batting")
```

```
  COUNT(*)
1    93955
```

Does it matter if there is an asterisk or one or more variable names between the parentheses?

Transforming Variables with Arithmetic

SQL can create new variables from existing variables using basic elementwise arithmetic.

```
db <- dbGetQuery(
  con,
  "SELECT AB, H, H*AB
  FROM Batting")
```

The name of the new variable will be the text describing the operation you used to create it. This is not great, let's see how to fix this.

```
names(db)
```

```
[1] "AB"   "H"    "H*AB"
```

Renaming Fields (Variables)

```
db <- dbGetQuery(
  con,
  "SELECT AB, H, H*AB AS HAB
  FROM Batting")
```

```
names(db)
```

```
[1] "AB"  "H"   "HAB"
```

Be Careful!

```r
dbGetQuery(con, "SELECT AB, H, H/AB
           FROM Batting LIMIT 7")
```

```
   AB   H H/AB
1   0   0   NA
2   2   0    0
3   0   0   NA
4   1   0    0
5   0   0   NA
6 468 131    0
7 602 189    0
```

Why?!?

```
str(dbGetQuery(con, "SELECT AB, H, H/AB
               FROM Batting"))
```

```
'data.frame':	93955 obs. of  3 variables:
 $ AB  : int  0 2 0 1 0 468 602 609 615 601 ...
 $ H   : int  0 0 0 0 0 131 189 200 198 196 ...
 $ H/AB: int  NA 0 NA 0 NA 0 0 0 0 0 ...
```

Transforming Then Summarizing

```r
dbGetQuery(con,
"SELECT MIN(H/CAST(AB AS REAL)) FROM Batting")
```

```
  MIN(H/CAST(AB AS REAL))
1                       0
```

Summarizing After Grouping

```r
db <- dbGetQuery(con,
                 "SELECT playerID, SUM(salary)
                 FROM Salaries GROUP BY playerID")
```

```r
str(db)
```

```
'data.frame':	4196 obs. of  2 variables:
 $ playerID   : chr  "aardsda01" "aasedo01" "abadan01" "abb
 $ SUM(salary): num  4259750 2300000 327000 985000 1296050C
```

Summarizing After Grouping and Aggregating

```r
db <- dbGetQuery(con,
"SELECT playerID
FROM batting
WHERE yearID >= 2005
GROUP BY playerID
HAVING SUM(RBI) >= 500")
```

```r
str(db)
```

```
'data.frame':   33 obs. of  1 variable:
 $ playerID: chr  "abreubo01" "bayja01" "berkmla01" "burrep
```

Ordering After Grouping and Aggregating

```
db <- dbGetQuery(con,
"SELECT playerID
FROM batting
WHERE yearID >= 2005
GROUP BY playerID
HAVING SUM(RBI) >= 500
ORDER BY SUM(RBI) DESC")
```

```
str(db)
```

```
'data.frame':   33 obs. of  1 variable:
 $ playerID: chr  "howarry01" "rodrial01" "pujolal01" "teix
```

Joins (Merging Multiple Data Sets)

An underratedly tricky but important task in real life is merging or joining data sets by one or more common records that are shared between them.

SQL can help us do this!

We'll learn how to do this in `R` later.

There are three types of joins:

- Inner joins (keep rows common to both datasets)
- Left joins (keep rows in the dataset on the left)
- Right joins (keep rows in the dataset on the right)

Left and right refer to where in the SQL command the dataset appears.

An Inner Join Example

```r
dbListFields(con, "Batting")
```

```
 [1] "playerID"  "yearID"    "stint"     "teamID"    "lgID'
 [7] "G_batting" "AB"        "R"         "H"         "2B"
[13] "HR"        "RBI"       "SB"        "CS"        "BB"
[19] "IBB"       "HBP"       "SH"        "SF"        "GIDP'
```

```r
dbListFields(con, "Master")
```

```
 [1] "lahmanID"     "playerID"     "managerID"    "hofID"
 [6] "birthMonth"   "birthDay"     "birthCountry" "birthSta
[11] "deathYear"    "deathMonth"   "deathDay"     "deathCou
[16] "deathCity"    "nameFirst"    "nameLast"     "nameNote
[21] "nameNick"     "weight"       "height"       "bats"
[26] "debut"        "finalGame"    "college"      "lahman4(
[31] "retroID"      "holtzID"      "bbrefID"
```

How do we do an inner join?

```
db <- dbGetQuery(con,
"SELECT m.nameFirst First, m.nameLast Last,
sum(RBI) as RBI_TOTAL
FROM batting b
INNER JOIN master m ON b.playerID == m.playerID
WHERE yearID >= 2005
GROUP BY b.playerID
HAVING RBI_total >= 500
ORDER BY -RBI_total")
```

Left Join Example

```r
dbGetQuery(con, "SELECT * FROM SchoolsPlayers")
```

```
      playerID    schoolID yearMin yearMax
1    aardsda01        rice    2002    2003
2    aardsda01      pennst    2001    2001
3    abbeybe01     vermont    1888    1892
4    abbotgl01   carkansas    1970    1970
5    abbotje01    kentucky    1991    1992
6    abbotji01    michigan    1986    1988
7    abbotky01   longbeach    1989    1989
8    abbotky01        ucsd    1987    1988
9    abbotod01  washjeffpa    1906    1910
10   abernte01        elon    1939    1941
11   ablesha01  swesterntx    1903    1904
12   accarje01  illinoisst    2001    2003
13   ackerji01       texas    1978    1980
14    acrema01     nmstate    1989    1991
15   adairje01     okstate    1957    1958
```

How do we do the left join?

```
db <- dbGetQuery(con,
"SELECT roy.playerID playerID, roy.yearID year, lgID league
FROM AwardsPlayers roy
LEFT JOIN
(SELECT * FROM SchoolsPlayers) c
ON c.playerID == roy.playerID
WHERE awardID LIKE \"Rookie%\"")
```

Closing Things Up

```
dbDisconnect(con)
dbUnloadDriver(drv)
```

More Information

This is only the tip of the iceberg in terms of what you can do with SQL.

Here's one handy reference for learning more: https://sqlzoo.net/wiki/SQL_Tutorial