Big Data Analytics

Lecture 7: Visualization II | Data Storage, Databases Interaction with R

Prof. Dr. Ulrich Matter
(University of St. Gallen)

22/04/2021

1 Data Visualization Part II

1.1 Time and Space

The previous visualization exercises were focused on visually exploring patterns in the tipping behavior of people taking a NYC yellow cap ride. Based on the same data set, will explore the time dimension and spatial dimension of the TLC Yellow Cap data. That is, we explore where trips tend to start and end, depending on the time of the day.

1.1.1 Preparations

For the visualization of spatial data we first load additional packages that give R some GIS features.

```
# load GIS packages
library(rgdal)
library(rgeos)
```

Moreover, we download and import a so-called 'shape file' (a geospatial data format) of New York City. This will be the basis for our visualization of the spatial dimension of taxi trips. The file is downloaded from New York's Department of City Planning and indicates the city's community district borders.[1]

```
# download the zipped shapefile to a temporary file, unzip
URL <- "https://www1.nyc.gov/assets/planning/download/zip/data-maps/open-data/nycd_19a.zip"
tmp_file <- tempfile()
download.file(URL, tmp_file)
file_path <- unzip(tmp_file, exdir= "../data")
# delete the temporary file
unlink(tmp_file)
```

Now we can import the shape file and have a look at how the GIS data is structured.

```
# read GIS data
nyc_map <- readOGR(file_path[1], verbose = FALSE)

# have a look at the GIS data
summary(nyc_map)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##        min       max
```

[1] Similar files are provided online by most city authorities in developed countries. See, for example, GIS Data for the City and Canton of Zurich: https://maps.zh.ch/.

```
## x 913175.1 1067382.5
## y 120121.9  272844.3
## Is projected: TRUE
## proj4string :
## [+proj=lcc +lat_0=40.1666666666667 +lon_0=-74 +lat_1=41.0333333333333
## +lat_2=40.6666666666667 +x_0=300000 +y_0=0 +datum=NAD83 +units=us-ft +no_defs]
## Data attributes:
##      BoroCD        Shape_Leng       Shape_Area
##  Min.   :101.0   Min.   : 23963   Min.   : 24293239
##  1st Qu.:205.5   1st Qu.: 36611   1st Qu.: 48407357
##  Median :308.0   Median : 52246   Median : 82702417
##  Mean   :297.2   Mean   : 74890   Mean   :118724012
##  3rd Qu.:405.5   3rd Qu.: 85711   3rd Qu.:136615357
##  Max.   :595.0   Max.   :270660   Max.   :599062130
```

Note that the coordinates are not in the usual longitude and latitude units. The original map uses a different projection than the TLC data of cap trips records. Before plotting, we thus have to change the projection to be in line with the TLC data.

```
# transform the projection
nyc_map <- spTransform(nyc_map, CRS("+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"))
# check result
summary(nyc_map)
```

```
## Object of class SpatialPolygonsDataFrame
## Coordinates:
##         min       max
## x -74.25559 -73.70001
## y  40.49612  40.91553
## Is projected: FALSE
## proj4string : [+proj=longlat +datum=WGS84 +no_defs]
## Data attributes:
##      BoroCD        Shape_Leng       Shape_Area
##  Min.   :101.0   Min.   : 23963   Min.   : 24293239
##  1st Qu.:205.5   1st Qu.: 36611   1st Qu.: 48407357
##  Median :308.0   Median : 52246   Median : 82702417
##  Mean   :297.2   Mean   : 74890   Mean   :118724012
##  3rd Qu.:405.5   3rd Qu.: 85711   3rd Qu.:136615357
##  Max.   :595.0   Max.   :270660   Max.   :599062130
```

One last preparatory step is to convert the map data to a `data.frame` for plotting with `ggplot`.

```
nyc_map <- fortify(nyc_map)
```

1.1.2 Pick-up and drop-off locations

Since trips might actually start or end outside of NYC, we first restrict the sample of trips to those within the boundary box of the map. For the sake of the exercise, we only select a random sample of 50000 trips from the remaining trip records.

```
# taxi trips plot data
taxi_trips <- taxi[start_long <= max(nyc_map$long) &
                       start_long >= min(nyc_map$long) &
                       dest_long <= max(nyc_map$long) &
                       dest_long >= min(nyc_map$long) &
                       start_lat <= max(nyc_map$lat) &
                       start_lat >= min(nyc_map$lat) &
```

```
                        dest_lat <= max(nyc_map$lat) &
                        dest_lat >= min(nyc_map$lat)
                        ]
taxi_trips <- taxi_trips[sample(nrow(taxi_trips), 50000)]
```

In order to visualize how the cap traffic is changing over the course of the day, we add an additional variable called `start_time` in which we store the time (hour) of the day a trip started.

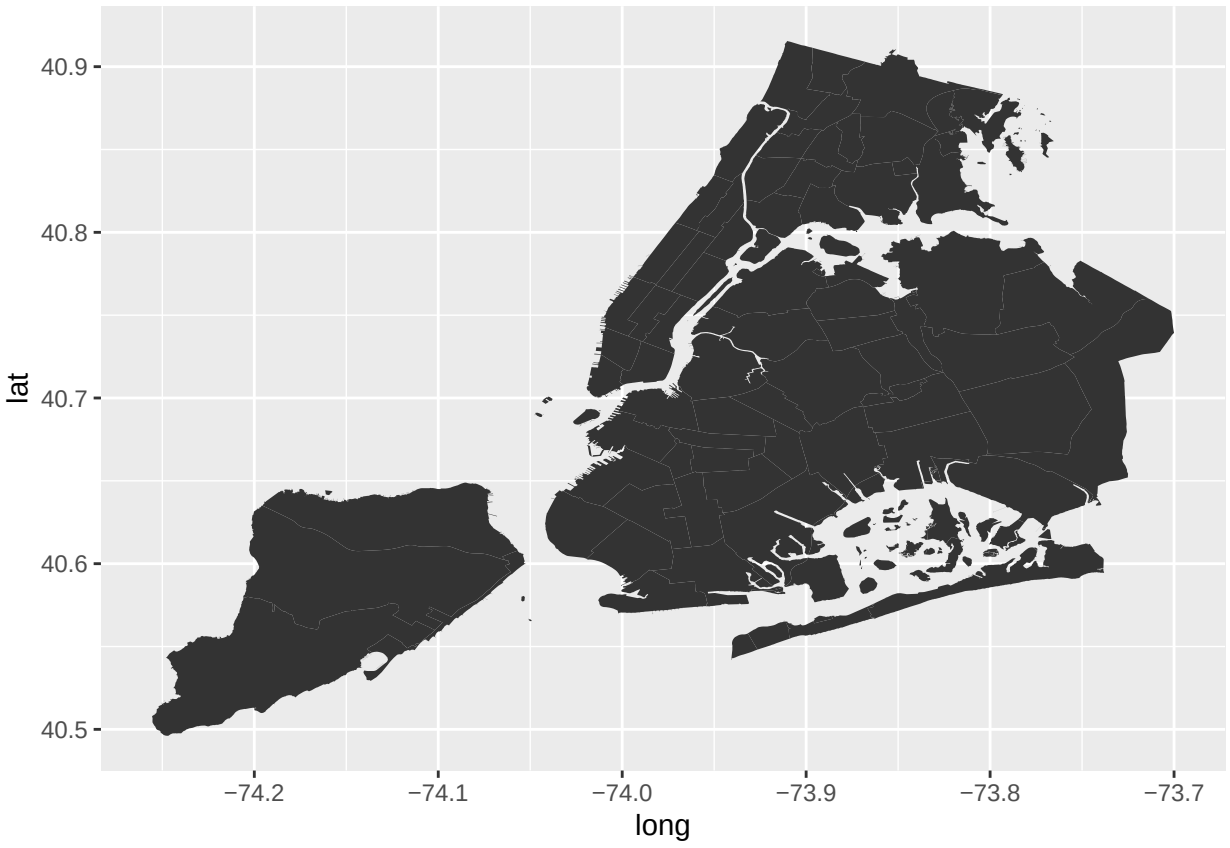
```
taxi_trips$start_time <- hour(taxi_trips$pickup_time)
```

Particularly, we want to look at differences between, morning, afternoon, and evening/night.

```
# define new variable for facets
taxi_trips$time_of_day <- "Morning"
taxi_trips[start_time > 12 & start_time < 17]$time_of_day <- "Afternoon"
taxi_trips[start_time %in% c(17:24, 0:5)]$time_of_day <- "Evening/Night"
taxi_trips$time_of_day  <- factor(taxi_trips$time_of_day, levels = c("Morning", "Afternoon", "Evening/N
```

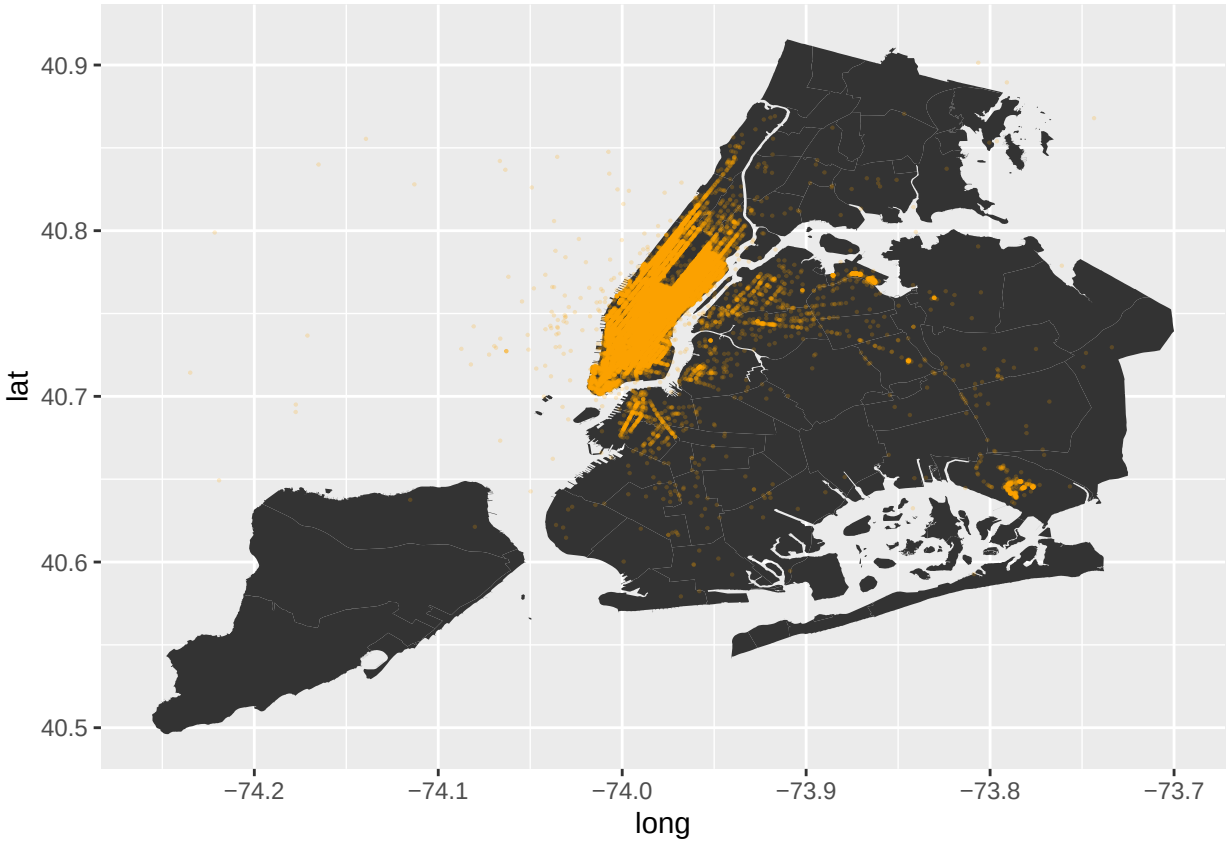
We initiate the plot by first setting up the canvas with our taxi trips data. Then, we add the map as a first layer.

```
# set up the canvas
locations <- ggplot(taxi_trips, aes(x=long, y=lat))
# add the map geometry
locations <- locations + geom_map(data = nyc_map,
                                  map = nyc_map,
                                  aes(map_id = id))
locations
```



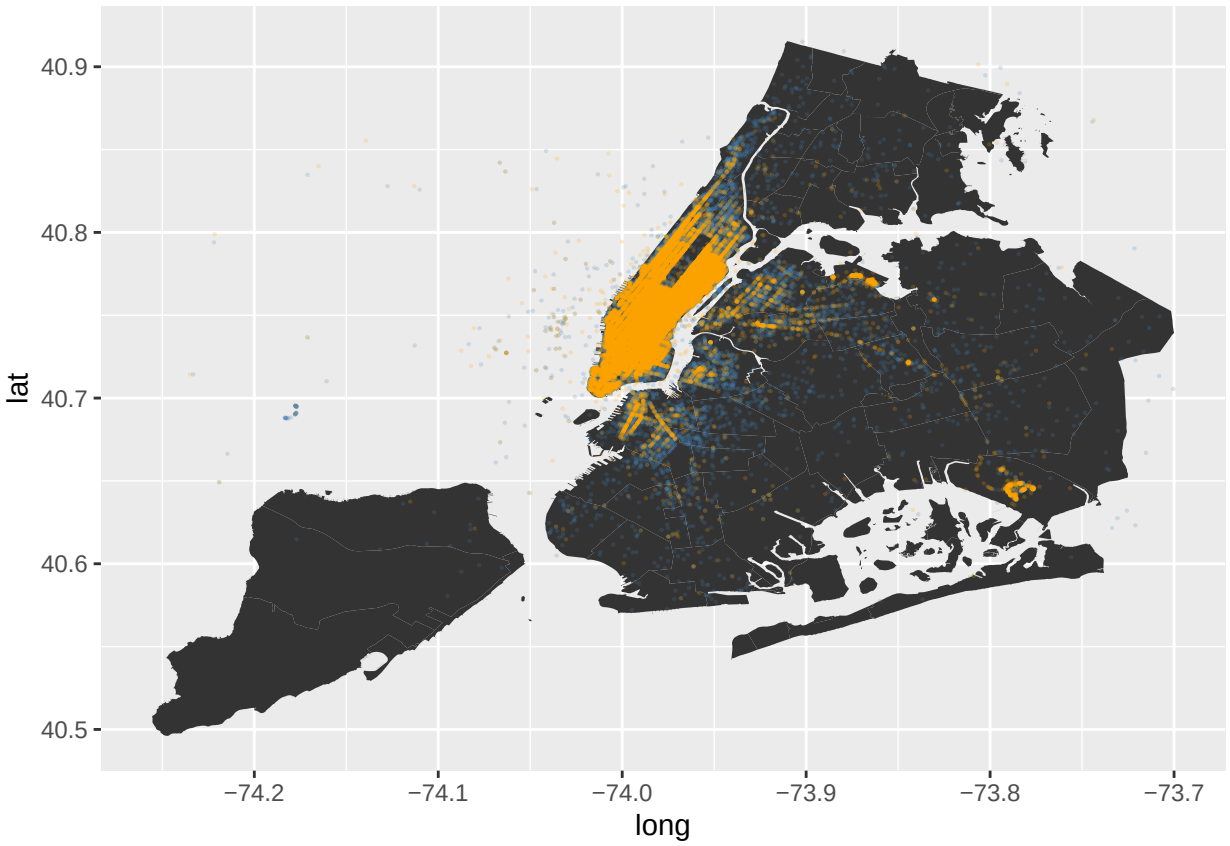
Now we can start adding the pick-up and drop-off locations of cap trips.

```
# add pick-up locations to plot
locations +
     geom_point(aes(x=start_long, y=start_lat),
                color="orange",
                size = 0.1,
                alpha = 0.2)
```



As to be expected, most of the trips start in Manhattan. Now let's look at where trips end.

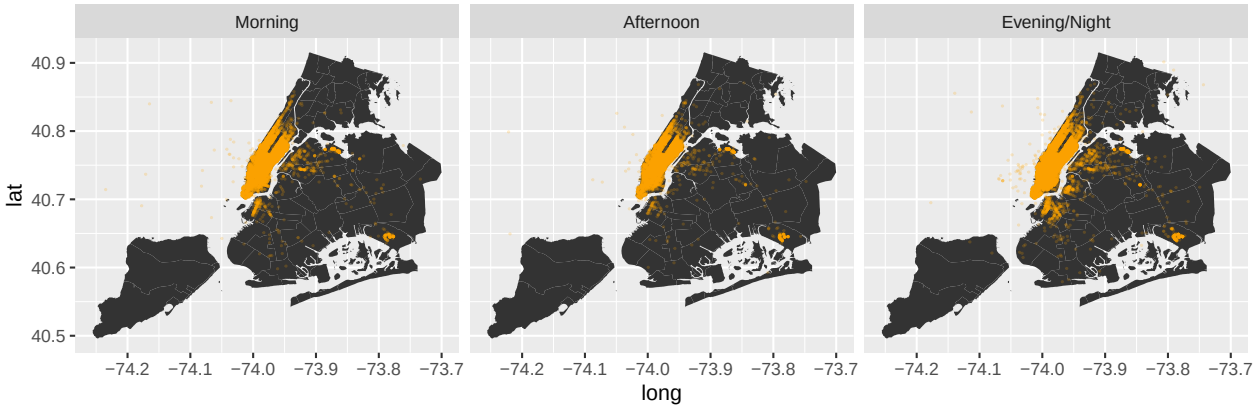
```
# add pick-up locations to plot
locations +
     geom_point(aes(x=dest_long, y=dest_lat),
                color="steelblue",
                size = 0.1,
                alpha = 0.2) +
     geom_point(aes(x=start_long, y=start_lat),
                color="orange",
                size = 0.1,
                alpha = 0.2)
```



Incidentally, more trips tend to end outside of Manhattan. And the destinations seem to be broader spread across the city then the pick-up locations. Most destinations are still in Manhattan, though.

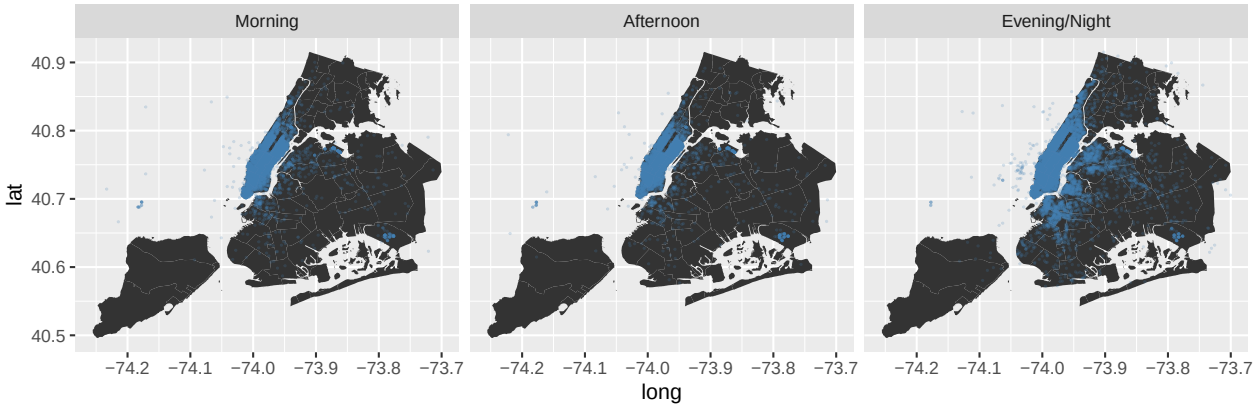
Now let's have a look at how this picture changes depending on the time of the day.

```
# pick-up locations
locations +
     geom_point(aes(x=start_long, y=start_lat),
                color="orange",
                size = 0.1,
                alpha = 0.2) +
     facet_wrap(vars(time_of_day))
```



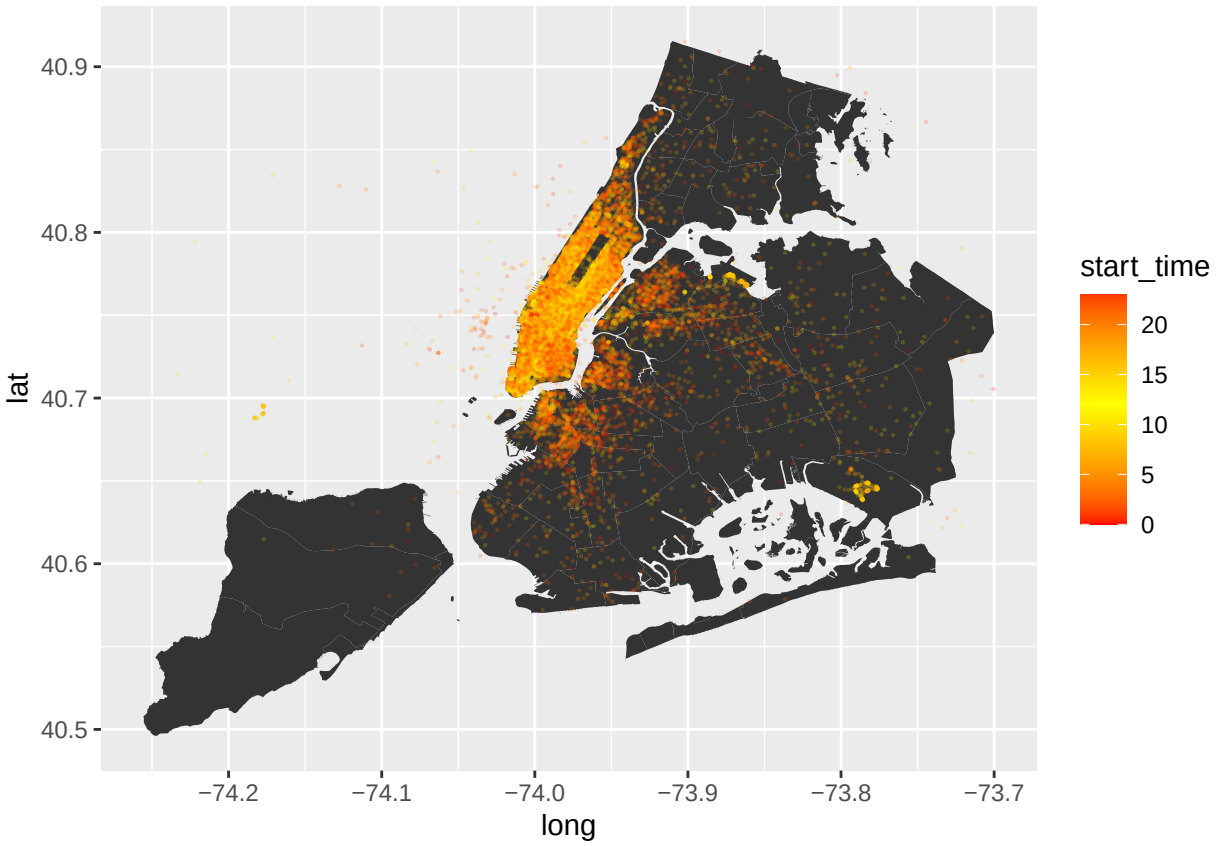

```
# drop-off locations
locations +
```

```
    geom_point(aes(x=dest_long, y=dest_lat),
               color="steelblue",
               size = 0.1,
               alpha = 0.2) +
    facet_wrap(vars(time_of_day))
```



Alternatively, we can plot the hours on a continuous scale.

```
# drop-off locations
locations +
    geom_point(aes(x=dest_long, y=dest_lat, color = start_time ),
               size = 0.1,
               alpha = 0.2) +
    scale_colour_gradient2( low = "red", mid = "yellow", high = "red",
                            midpoint = 12)
```



2 Data Storage and Databases

2.1 (Big) Data Storage

So far, we have primarily been concerned with situations in which a data set is too large to fit into RAM, making an analysis of these data either impossible or very slow (inefficient) when using standard tools. Thus, we have explored concepts and tools that help us use the available RAM (and virtual memory) most efficiently for data analysis tasks.

In this lecture, we are concerned with (I) how we can store large data sets permanently on a mass storage device in an efficient way (here, efficient can be understood as 'not taking up too much space') and (II) how we can load (parts of) this data set in an efficient way (here, efficient~fast) for analysis.

We look at this problem in two situations:

- The data needs to be stored locally (e.g., on the hard disk of our laptop).
- The data can be stored on a server 'in the cloud'.

Various tools have been developed over the last few years to improve the efficiency of storing and accessing large amounts of data (see Walkowiak (2016), chapters 5 and 6 for an overview). Here, we focus on the basic concept of Relational Database Systems (RDBMS) and a well-known tool based on this concept, the Structured Query Language (SQL; more specifically, SQLite)

2.2 RDBMS basics

RDBMSs have two key features that tackle the two efficiency concerns mentioned above:

- The *relational data model*: The overall data set is split by columns (covariates) into tables in order to reduce the storage of redundant variable-value repetitions. The resulting database tables are then linked

via key-variables (unique identifiers). Thus (simply put), each type of entity on which observations exist resides in its own database table. Within this table, each observation has it's unique id. Keeping the data in such a structure is very efficient in terms of storage space used.

- *Indexing*: The key-columns of the database tables are indexed, meaning (in simple terms) ordered on disk. Indexing a table takes time but it has to be performed only once (unless the content of the table changes). The resulting index is then stored on disk as part of the database. These indices substantially reduce the number of disk accesses required to query/find specific observations. Thus, they make the loading of specific parts of the data for analysis much more efficient.

The loading/querying of data from an RDBMS typically involves the selection of specific observations (rows) and covariates (columns) from different tables. Due to the indexing, observations are selected efficiently, and the defined relations between tables (via keys) facilitate the joining of columns to a new table (the queried data).

2.3 Getting started with (R)SQLite

SQLite is a free full-featured SQL database engine and widely used across platforms and is typically pre-installed with Windows and OSX distributions. It is perfect to learn how to use RDBMSs/SQL. The R-package `RSQLite` embeds SQLite in R. That is, it provides functions that allow us to us SQLite directly from within R.

2.3.1 First steps in SQLite

In the terminal, we can directly call SQLite as a command-line tool (on most modern computers this is now `sqlite3`). In this short example, we set up our first SQLite database, using the command line. In the file structure of the course repository, we first switch to the data directory.

```
cd materials/data
```

With one simple command, we both create a new SQLite database called `mydb.sqlite` and connect/run SQLite.

```
sqlite3 mydb.sqlite
```

This created a new file `mydb.sqlite` in our `data` directory. And we are now running `sqlite` in our terminal. But, the database is still empty. There are no tables in it. We can check the tables of a database with the following SQL command `.tables`.

```
.tables
```

As expected, nothing is returned. Now, let's create our first table and import the `economics.csv` data set to it. In SQLite, it makes sense to first set up an empty table in which all column data types are defined before importing data from a CSV-file to it. If a CSV is directly imported to a new table (without type definitions), all columns will be set to `TEXT` by default.

```
-- Create the new table
CREATE TABLE econ(
"date" DATE,
"pce" REAL,
"pop" INTEGER,
"psavert" REAL,
"uempmed" REAL,
"unemploy" INTEGER
);

-- prepare import
.mode csv
```

Table 1: 1 records

date	pce	pop	psavert	uempmed	unemploy
1968-01-01	531.5	199808	11.7	5.1	2878

```
-- import data from csv
.import economics.csv econ
```

Now we can have a look at the new database table in SQLite. `.tables` shows that we now hove one table called `econ` in our database and `.schema` displays the structure of the new `econ` table.

```
.tables
```

```
# econ
```

```
.schema econ
```

```
# CREATE TABLE econ(
# "date" DATE,
# "pce" REAL,
# "pop" INTEGER,
# "psavert" REAL,
# "uempmed" REAL,
# "unemploy" INTEGER
# );
```

With this, we can actually start querying data with SQLite. In order to make the query results easier to read, we first set two options regarding how query results are displayed in the terminal. `.header on` enables the display of the column names in the returned query results. And `.mode columns` arranges the query results in columns.

```
.header on
```

```
.mode columns
```

In our first query, we select all (`*`) variable values of the observation of January 1968.

```
select * from econ where date = '1968-01-01';
```

Now let's select all year/months in which there were more than 15 million unemployed, ordered by date.

```
select date from econ
where unemploy > 15000
order by date;
```

When done working with the database, we can exit SQLite with the `.quit` command.

2.3.2 Indices and joins

So far we have only had a look at the basic functionality of SQLite. But we have not really looked at the key features of an RDBMS (indexing and relations between tables).

We set up a new database called `air.sqlite` and import the csv-file `flights.csv` (used in previous lectures) as a first table.

```
# create database and run sqlite
sqlite3 air.sqlite
```

Table 2: Displaying records 1 - 10

date
2009-09-01
2009-10-01
2009-11-01
2009-12-01
2010-01-01
2010-02-01
2010-03-01
2010-04-01
2010-11-01
date

```
-- import csvs
.mode csv
.import flights.csv flights
```

Again, we can check if everything worked out well with `.tables` and `.schema`.

```
.tables
.schema flights
```

In `flights`, each row describes a flight (the day it took place, its origin, its destination etc.). It contains a covariate `carrier` containing the unique ID of the respective airline/carrier carrying out the flight as well as the covariates `origin` and `dest`. The latter two variables contain the unique IATA-codes of the airports from which the flights departed and where they arrived, respectively. In `flights` we thus have observations at the level of individual flights.

Now we extend our database in a meaningful way, following the relational data model idea. From the [ASA's website], we download two additional csv files containing data that relate to the flights table:

- `airports.csv`: Describes the locations of US Airports (relates to `origin` and `dest`).
- `carriers.csv`: A listing of carrier codes with full names (relates to the `carrier`-column in `flights`.

In this code example, the two csvs have already been downloaded to the `materials/data`-folder.

```
-- import airport data
.mode csv
.import airports.csv airports
.import carriers.csv carriers

-- inspect the result
.tables
.schema airports
.schema carriers
```

Now we can run our first query involving the relation between tables. The aim of the exercise is to query flights data (information on departure delays per flight number and date; from the `flights`-table) for all `United Air Lines Inc.`-flights (information from the `carriers` table) departing from `Newark Intl` airport (information from the `airports`-table). In addition, we want the resulting table ordered by flight number. For the sake of the exercise, we only show the first 10 results of this query (`LIMIT 10`).

```
SELECT
year,
month,
```

Table 3: Displaying records 1 - 10

year	month	day	dep_delay	flight
2013	1	4	0	1
2013	1	5	-2	1
2013	3	6	1	1
2013	2	13	-2	3
2013	2	16	-9	3
2013	2	20	3	3
2013	2	23	-5	3
2013	2	26	24	3
2013	2	27	10	3
2013	1	5	3	10

```
day,
dep_delay,
flight
FROM (flights INNER JOIN airports ON flights.origin=airports.iata)
INNER JOIN carriers ON flights.carrier = carriers.Code
WHERE carriers.Description = 'United Air Lines Inc.'
AND airports.airport = 'Newark Intl'
ORDER BY flight
LIMIT 10;
```

Note that this query has been executed without indexing any of the tables first. Thus SQLite could not take any 'shortcuts' when matching the ID columns in order to join the tables for the query output. That is, SQLite had to scan the entire columns to find the matches. Now we index the respective id columns and re-run the query

```
CREATE INDEX iata_airports ON airports (iata);
CREATE INDEX origin_flights ON flights (origin);
CREATE INDEX carrier_flights ON flights (carrier);
CREATE INDEX code_carriers ON carriers (code);
```

Now we can re-run the query from above. Note that SQLite optimizes the efficiency of the query without our explicit instructions. If there are indices it can use to speed up the query, it will do so.

```
SELECT
year,
month,
day,
dep_delay,
flight
FROM (flights INNER JOIN airports ON flights.origin=airports.iata)
INNER JOIN carriers ON flights.carrier = carriers.Code
WHERE carriers.Description = 'United Air Lines Inc.'
AND airports.airport = 'Newark Intl'
ORDER BY flight
LIMIT 10;
```

You find the final `air.sqlite`, including all the indices and tables as `materials/data/air_final.sqlite` in the course's code repository.

Table 4: Displaying records 1 - 10

year	month	day	dep_delay	flight
2013	1	4	0	1
2013	1	5	-2	1
2013	3	6	1	1
2013	2	13	-2	3
2013	2	16	-9	3
2013	2	20	3	3
2013	2	23	-5	3
2013	2	26	24	3
2013	2	27	10	3
2013	1	5	3	10

2.4 SQLite from within R

The `RSQLite`-package provides various R functions to control basically all of SQLite's functionalities from within R. In the following example, we explore how `RSQLite` can be used to set up and query the `air.sqlite` shown in the example above.

2.4.1 Creating a new database with `RSQLite`

Similarly to the raw SQLite-syntax, connecting to a database that does not exist yet, actually creates this (empty database). Note that for all interactions with the database from within R, we need to refer to the connection (here: `con_air`).

```
# load packages
library(RSQLite)

# initiate the database
con_air <- dbConnect(SQLite(), "../data/air.sqlite")
```

2.4.2 Importing data

With `RSQLite` we can easily add `data.frames` as SQLite tables to the database.

```
# import data into current R sesssion
flights <- fread("../data/flights.csv")
airports <- fread("../data/airports.csv")
carriers <- fread("../data/carriers.csv")

# add tables to database
dbWriteTable(con_air, "flights", flights)
dbWriteTable(con_air, "airports", airports)
dbWriteTable(con_air, "carriers", carriers)
```

2.4.3 Issue queries

Now we can query the database from within R. By default, `RSQLite` returns the query results as `data.frames`. Queries are simply character strings written in SQLite.

```
# define query
delay_query <-
"SELECT
year,
```

```
month,
day,
dep_delay,
flight
FROM (flights INNER JOIN airports ON flights.origin=airports.iata)
INNER JOIN carriers ON flights.carrier = carriers.Code
WHERE carriers.Description = 'United Air Lines Inc.'
AND airports.airport = 'Newark Intl'
ORDER BY flight
LIMIT 10;
"

# issue query
delays_df <- dbGetQuery(con_air, delay_query)
delays_df
```

```
##    year month day dep_delay flight
## 1  2013     1   4         0      1
## 2  2013     1   5        -2      1
## 3  2013     3   6         1      1
## 4  2013     2  13        -2      3
## 5  2013     2  16        -9      3
## 6  2013     2  20         3      3
## 7  2013     2  23        -5      3
## 8  2013     2  26        24      3
## 9  2013     2  27        10      3
## 10 2013     1   5         3     10
```

References

Walkowiak, Simkon. 2016. *Big Data Analytics with R*. Birmingham, UK: PACKT Publishing.