In modern economics, we often deal with large and complex sets of data (big data).

Some data are “conventional” (national accounting, micro panels, industry surveys, census data, international trade flows, ...).

Some data come in “non-conventional” forms (plain text, library records, parish and probate records, GIS data, electricity consumption, satellite imagery, web scraping, network structure, social media, ...).

Some data are old, but now easily available. Check the amazing dataset at https://www.ucl.ac.uk/lbs/.

This trend will increase over time as more archives get digitalized.

These large datasets create their own challenges in terms of data wrangling, storage, management, visualization, and processing.
Parish and probate data

Male labour force shares in England and Wales 1381-1911

Chart key:
- Trendline
- Calculations
- Error margin

Agriculture
Secondary sector
Tertiary sector
Mining

Share of the adult male labour force

1350 1400 1450 1500 1550 1600 1650 1700 1750 1800 1850 1900

1381 poll tax
1522 muster list + coroners’ reports
probate
probate + parish reg.
censuses
parish reg.
Satellite imagery

Estimated daily per capita expenditure, 2012-2015

Nigeria

Uganda

Tanzania

Malawi

Average daily per capita consumption expenditure ($)
Cell phone usage

Fig. 2. Construction of high-resolution maps of poverty and wealth from call records. Information derived from the call records of 1.5 million subscribers is overlaid on a map of Rwanda. The northern and western provinces are divided into cells (the smallest administrative unit of the country), and the cell is shaded according to the average (predicted) wealth of all mobile subscribers in that cell. The southern province is overlaid with a Voronoi division that uses geographic identifiers in the call data to segment the region into several hundred thousand small partitions. (Bottom right inset) Enlargement of a 1 km² region near Kiyonza, with Voronoi cells shaded by the predicted wealth of small groups (5 to 15 subscribers) who live in each region.

Blumenstock et al. (2015)
Handling data II

• This will become more salient over time: watch the lectures at http://www.equality-of-opportunity.org/bigdatacourse/.

• Why?

  1. Explosion of data sources.
  2. Computational power.

• This topic will require a whole course on its own, so I will only introduce fundamental ideas.

• Also, this lecture should motivate you to further understand the data structures of your favorite programming language (e.g., in R, the dataframe; in Python, the pandas).
Some basic references:


A good way to start thinking about how to handle data efficiently is to distinguish between the data and its metadata.

Data: ultimate information of interest.

Metadata: data about the data.

Tye Rattenbury et al. subdivide metadata in five aspects:

1. Structure: format and encoding of its records and fields.
2. Granularity: kinds of entities that each data record contains information about.
3. Accuracy: quality of the data.
4. Temporality: temporal structure of the representation of the data.
5. Scope: number of distinct attributes represented and the population coverage.
Data vs. metadata II

• For simple projects, the metadata will be trivial and you do not need to spend much time thinking about it.

• But for complex, large projects, spending some time “getting” the metadata right will be crucial:

  1. Assess how much effort you want to spend in wrangling the data (e.g., manual vs. automatization).

  2. Assess how much effort you want to spend auditing the data.

  3. Assess how much effort you want to spend in storing the data efficiently.

  4. Assess how early decisions regarding the metadata might limit your future analysis.
The Quartz guide to bad data

I once acquired the complete dog licensing database for Cook County, Illinois. Instead of requiring the person registering their dog to choose a breed from a list, the creators of the system had simply given them a text field to type into. As a result this database contained at least 250 spellings of Chihuahua.

• Issues:

  1. Inconsistent spelling and/or historical changes.
  2. N/A, blank, or null values.
  3. 0 values (or −1 or dates 1900, 1904, 1969, or 1970).
  4. Text is garbled.
  5. Lines ends are garbled.
  6. Text comes from optical-character recognition (OCR).
Alternative data file formats: pdf files

- Many documents follow pdf format.
- Package in R: Pdftools 2.0.
- An example:

```r
pdf_data(pdf, opw = "", upw = "")
```
You need to learn a programming language that manipulates regular expressions efficiently.

Tye Rattenbury et al. claim that between 50% and 80% of real-life data analysis is spent with data wrangling.

About regular expressions in general:


Modern programming languages have powerful regular expressions capabilities.

Regular expressions and R


- Two key packages: `dplyr` and `tidyr` part of `tidyverse`:
  
  ```r
  install.packages("tidyverse")
  ```

- In particular, learn to use the piping command from `dplyr` to make code more readable:
  
  ```r
  x %>% f(y)
  f(x, y)
  ```

- A real example we will discuss below
  
  ```r
  mySelection %>%
    filter(weight < 5) %>%
    select(species_id, sex, weight)
  ```

- Look also at [https://www.tidytextmining.com/](https://www.tidytextmining.com/) for text mining.
A graph we already saw
Alternative data file formats: JSON

- JSON: JavaScript Object Notation, https://www.json.org/:

- Hierarchical data format:
  1. A collection of key-value pairs.
  2. An ordered list (array) of values. The values can be themselves either data or another nested structure.

- Very efficient for the storage, transmitting, and parsing of data.

- It has gained much popularity with respect to XML.

- Important for modern databases (more on this later).

- At the core of Jupyter.

- UBSON: Universal Binary JSON.
Example of JSON data, myObj:

```json
{
    "name":"Adam Smith",
    "age":30,
    "universities":[ "Princeton", "Penn", "Minnesota" ]
}
```

Accessing the data:

```python
x = myObj.universities[0];
```
In Rm we can install the rjson package.

```r
install.packages("rjson")
library (rjson)
```

And use its capabilities to read a JSON object:

```r
mydata <- fromJSON (myObj)
mydata_df <- data.frame (NULL)
for(i in seq_along (mydata$universities)) {
  df <- data.frame (mydata$universities)
  layoff_df <- rbind (layoff_df, df)
}
```
More alternative data file formats

- HTML and XML.

- Plenty of alternative proprietary data formats:
  1. Microsoft office.
  2. Stata files.
  3. pdf files.
  4. ...

- Usually a bad idea to rely on them...

- ...but sometimes they are the only alternative. Resort to tool such as Tabula (https://tabula.technology) and WebPlotDigitizer.
• For datasets of moderate size, spreadsheets are a conventional choice.

• However, you should be careful while using them:
  1. Do not employ their proprietary data formats (i.e., xlsx).
  2. Do not perform any computation in the spreadsheet. They are not reproducible and you are bound to make mistakes (or simply forget what you did).

• Best practices:
  1. Comma-separated values (CSV) files are easier to share among co-authors, computers, and across time.
  2. Load the CSV file into Julia or R and run a script file on it. Store the script!
  3. Use Jupyter, Hydrogen, or similar if you care about showing all the steps in detail.
  4. Use tidyverse in R to interact with Excel and other standard spreadsheets.
• A database is a self-described, organized collection of records (tuples), each of them with multiple attributes.

• Components:
  1. Data: the records and attributes of the database.

• A spreadsheet is, then, just a very simple database.

• Similarly, a flat file (i.e., a simple CSV file) is a trivial database.

• A blockchain is a distributed database updated by consensus through a proof-of-work ticket.
Why databases? I

• Complex data structures require a more sophisticated database (either single or multi-user) with a database management system (DBMS) that stores, manages, and facilitates access to records.

• For instance, your data cannot fit into a simple table without excessive redundancies or without loss of efficiency in its processing.

• Examples in economics: CEX data, individual firm data, ....

• Other times, the data is too large to be stored in RAM and you just want to select and manipulate some observations in an efficient way.
<table>
<thead>
<tr>
<th>ID</th>
<th>EnNum</th>
<th>Name</th>
<th>Title</th>
<th>HireDate</th>
<th>Skill</th>
<th>SkillDate</th>
<th>Skill</th>
<th>SkillDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>02345</td>
<td>Brian Oates</td>
<td>DBA</td>
<td>2/14/1995</td>
<td>Basic Database Management</td>
<td>2/14/2002</td>
<td>Advanced Database Management</td>
<td>2/14/2005</td>
</tr>
<tr>
<td>6</td>
<td>10282</td>
<td>Richardson, Amanda</td>
<td>Clerk</td>
<td>8/11/2011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>09362</td>
<td>Susan Mathis</td>
<td>Database Programmer</td>
<td>8/2/2010</td>
<td>Basic DB Design</td>
<td>8/2/2012</td>
<td>Basic Database Manipulation</td>
<td>8/2/2012</td>
</tr>
<tr>
<td>8</td>
<td>14311</td>
<td>Duong, Leo</td>
<td>Programmer</td>
<td>8/1/2014</td>
<td>Basic Web Design</td>
<td>9/1/2015</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>Master Database Programming</td>
<td>Basic Spreadsheets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13363</td>
<td>Raymond F. Matthews</td>
<td>Programmer</td>
<td>3/12/2012</td>
<td>Basic C# Programming</td>
<td>3/12/2014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>09263</td>
<td>Chavez, Juan</td>
<td>Clerk</td>
<td>7/4/2010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>13932</td>
<td>Lee, Megan</td>
<td>Programmer</td>
<td>9/29/2013</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A good design

**Table name: EMPLOYEE**

<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>Employee_Name</th>
<th>Employee_Lname</th>
<th>Employee_HireDate</th>
<th>Employee_Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>00345</td>
<td>Johnny</td>
<td>Jones</td>
<td>3/4/1996</td>
<td>CR</td>
</tr>
<tr>
<td>00343</td>
<td>Franklin</td>
<td>Johnson</td>
<td>3/1/2000</td>
<td>Functional Agent</td>
</tr>
<tr>
<td>00341</td>
<td>Patricia</td>
<td>Robinson</td>
<td>6/1/2000</td>
<td>CR</td>
</tr>
<tr>
<td>00324</td>
<td>Jessica</td>
<td>Finkel</td>
<td>3/3/2000</td>
<td>Programmer</td>
</tr>
<tr>
<td>00323</td>
<td>Mary</td>
<td>Clark</td>
<td>7/2/2000</td>
<td>Analyst</td>
</tr>
<tr>
<td>00322</td>
<td>Ben</td>
<td>Jarman</td>
<td>5/23/2000</td>
<td>Clerk</td>
</tr>
<tr>
<td>00339</td>
<td>Jean</td>
<td>Chavez</td>
<td>7/4/2000</td>
<td>Clerk</td>
</tr>
<tr>
<td>00337</td>
<td>Jessica</td>
<td>Johnson</td>
<td>5/9/2000</td>
<td>Database Programmer</td>
</tr>
<tr>
<td>00329</td>
<td>Amanda</td>
<td>Richardson</td>
<td>4/1/2000</td>
<td>Clerk</td>
</tr>
<tr>
<td>00328</td>
<td>Raymond</td>
<td>Mattheis</td>
<td>3/1/2000</td>
<td>Programmer</td>
</tr>
<tr>
<td>00327</td>
<td>Robert</td>
<td>Almond</td>
<td>9/9/2000</td>
<td>Analyst</td>
</tr>
<tr>
<td>00326</td>
<td>William</td>
<td>Lee</td>
<td>9/9/2000</td>
<td>Programmer</td>
</tr>
</tbody>
</table>

**Table name: CERTIFIED**

<table>
<thead>
<tr>
<th>Employee_ID</th>
<th>Skill_ID</th>
<th>Certified_Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>00343</td>
<td>110</td>
<td>6/24/2000</td>
</tr>
<tr>
<td>00324</td>
<td>110</td>
<td>3/24/2005</td>
</tr>
<tr>
<td>00323</td>
<td>110</td>
<td>6/24/2000</td>
</tr>
<tr>
<td>00322</td>
<td>110</td>
<td>4/24/2007</td>
</tr>
<tr>
<td>00320</td>
<td>210</td>
<td>1/2/2007</td>
</tr>
<tr>
<td>00319</td>
<td>210</td>
<td>3/24/2009</td>
</tr>
<tr>
<td>00318</td>
<td>210</td>
<td>1/22/2012</td>
</tr>
<tr>
<td>00317</td>
<td>210</td>
<td>6/14/2013</td>
</tr>
<tr>
<td>00316</td>
<td>110</td>
<td>6/22/2012</td>
</tr>
<tr>
<td>00315</td>
<td>210</td>
<td>5/7/2013</td>
</tr>
<tr>
<td>00314</td>
<td>110</td>
<td>3/12/2014</td>
</tr>
<tr>
<td>00313</td>
<td>110</td>
<td>9/30/2014</td>
</tr>
<tr>
<td>00312</td>
<td>110</td>
<td>6/24/2015</td>
</tr>
</tbody>
</table>

**Table name: SKILL**

<table>
<thead>
<tr>
<th>Skill_ID</th>
<th>Skill Name</th>
<th>Skill_Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>Basic Database Management</td>
<td>Create and manage database user accounts.</td>
</tr>
<tr>
<td>120</td>
<td>Basic Web Design</td>
<td>Create and maintain HTML and CSS documents.</td>
</tr>
<tr>
<td>130</td>
<td>Advanced Spreadsheets</td>
<td>Use of advanced functions, user-defined functions, and macros.</td>
</tr>
<tr>
<td>140</td>
<td>Basic Process Modeling</td>
<td>Create core business processes models using standard libraries.</td>
</tr>
<tr>
<td>150</td>
<td>Basic Database Design</td>
<td>Create simple data models.</td>
</tr>
<tr>
<td>160</td>
<td>Master Database Programming</td>
<td>Create integrated triggers and procedure packages for a distributed environment.</td>
</tr>
<tr>
<td>170</td>
<td>Basic Spreadsheet</td>
<td>Create single tab worksheets with basic formulas.</td>
</tr>
<tr>
<td>180</td>
<td>Basic Programming</td>
<td>Create single-tier data aware modules.</td>
</tr>
<tr>
<td>190</td>
<td>Advanced Database Management</td>
<td>Manage Database Server Cursors.</td>
</tr>
<tr>
<td>200</td>
<td>Advanced Process Modeling</td>
<td>Evaluate and redesign core functional internal and external business processes.</td>
</tr>
<tr>
<td>210</td>
<td>Advanced Off Programming</td>
<td>Create multi-tier applications using multi-tiering.</td>
</tr>
<tr>
<td>220</td>
<td>Basic Database Manipulation</td>
<td>Create simple data retrieval and manipulation statements in SQL.</td>
</tr>
<tr>
<td>230</td>
<td>Advanced Database Manipulation</td>
<td>Use of advanced data manipulation methods for multi-tiered assets, set operations, and correlated subqueries.</td>
</tr>
</tbody>
</table>
Why databases? II

- Often, you can build your own database in your code using object-orientation and user-defined types.

- However, sometimes you need:
  1. Refined capabilities of selection/joins.
  2. Scalability.
  3. Ensure safe concurrent operations on data.
  4. Avoid data anomalies.
  5. Prevent data loss from hardware/software crashes.
  6. Interact with an already built database (e.g., at a statistical agency).
  7. Build your own database.
  8. Parallel computation and optimized data structures.
Database engines

- Plenty of industry-strength, scalable DBMS.

- At the core of each DBMS, you have a database engine that creates, reads, updates, and deletes (CRUD) data.

- You can always access the engine directly with an API (for instance, to use within your code in C++ or R). This is likely the most common case for researchers.

- In addition, there is usually a GUI to interact with the DBMS (most famous: Microsoft Access).

- Also, general query language for APIs not tied to any database, such as GraphQL.

Popularity by category

- Relational DBMS: 76%
- Search engines: 4.7%
- Key-value stores: 4.8%
- Native XML DBMS: 0.3%
- RDF stores: 0.3%
- Graph DBMS: 1.4%
- Document stores: 8.3%
- Time Series DBMS: 0.4%
- Wide column stores: 3.3%
Open source vs. commercial databases

![Chart showing the comparison between commercial and open source database rankings over time. The chart indicates a declining trend for commercial licenses and a rising trend for open source licenses. The data is sourced from DB-Engines.com.](image)
Databases vs. IBM RAMAC, 1956
Database management systems I

• As mentioned before, a DBMS plays three roles:

1. Data storage: special attention to system and disk failures and to data structures that deliver good performance.


2. Data management: how data is logically organized, who has access to it (read, write), and consistency conditions.

3. Data access: how access is accessed (queries) and what types of computations are allowed in them.

• In real-life applications, these three task can involved high levels of complexity.

• In particular: multiple people have access to them and they involve multiple units of hardware and software (think about an airline reservation system).
Modern DBMS hide how data is stored from end user applications:

1. Thus, systems can evolve over time (i.e., hardware and software implementation of data structures and optimized storage) without affecting you.

2. Similarly, you can change the database (e.g., add a new table) without having to modify code that queries the database and manipulates the results of the query.

3. The DBMS can handle abstract applications instead of being specifically tied to one design of a concrete application.

4. Most DBMS are declarative, not imperative (tell the software what you want, not how to get it):
   4.1 Easier to use for non-programmers (many users will not be)...
   4.2 ...but harder to optimize.
Optimized data structures

![Diagram of optimized data structures]

- Read Optimized
  - Hash
  - Point & Tree indexes
    - B-Tree
    - Trie
    - Skiplist
  - Cracking
  - Adaptive structures
    - Merging
  - Sparse Index
    - Bloom filter
    - Bitmap
- Write Optimized
  - PDT
  - LSM
  - Differential structures
    - PBT
    - MaSM
- Space Optimized
B trees

Diagram of a B-tree:
- Root node with keys 7, 16
- Child nodes with keys:
  - 1, 2, 5, 6
  - 9, 12
  - 18, 21
B+ trees
The CAP theorem


- In a distributed database, you can only choose two of:
  
  1. Consistency.
  
  2. Availability.
  
  3. Partition tolerance.

- If you think about it, the real trade-off is between consistency and availability since the problem comes from the existence of a partition tolerance.

- Extension: PACELC theorem (Daniel J. Abadi, 2012): even in the absence of partitions, one has to choose between latency (L) and consistency (C).
Relational database management systems

- Relational database management system (RDBMS) manage data stored in relations (i.e., a table).

- Each relation has a schema (description of attributes, their types, and constraints). An instance is data satisfying the schema.

- Each record (tuple) is a row of the relation and each attribute is a column.

- Each attribute has a domain consisting of a finite set of possible values within a few primitive types.

- Each attribute might have constraints (important for safety).

- The schema of the database is the set of relation schemas.

- The relations, not just the individual observations, are of interest.
This diagram is a little overwhelming, but it's simple compared to some you'll see in the wild! The key to understanding diagrams like this is to remember each relation always concerns a pair of tables. You don't need to understand the whole thing; you just need to understand the chain of relations between the tables that you are interested in.

For `nycflights13`:

- `flights` connects to `planes` via a single variable, `tailnum`.
- `flights` connects to `airlines` through the `carrier` variable.
- `flights` connects to `airports` in two ways: via the `origin` and `dest` variables.
- `flights` connects to `weather` via `origin` (the location), and `year`, `month`, `day`, and `hour` (the time).

Exercises

1. Imagine you wanted to draw (approximately) the route each plane flies from its origin to its destination. What variables would you need? What tables would you need to combine?

2. I forgot to draw the relationship between `weather` and `airports`. What is the relationship and how should it appear in the diagram?
Importance of constraints

HI, THIS IS YOUR SON’S SCHOOL. WE’RE HAVING SOME COMPUTER TROUBLE.

OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–

DID YOU REALLY NAME YOUR SON ‘ROBERT’? DROP TABLE STUDENTS;–?

OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.

WELL, WE’VE LOST THIS YEAR’S STUDENT RECORDS. I HOPE YOU’RE HAPPY.

AND I HOPE YOU’VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.
Relational model, algebra, and calculus

Built around two elements:

1. Relational model:
   
   
   1.2 Data is organized as tuples grouped into relations and independent of physical properties of storage.
   
   1.3 Consistent with first-order predicate logic.

2. Relational algebra and calculus:

   2.1 Proposed, again, by Edgar F. Codd (1972).

   2.2 A collection of operations (mutating joins, filtering joins, and set operations).

   2.3 A way defining logical outcomes for data transformations.
Edgar F. Codd (1923-2003)
The result of joining airlines to flights2 is an additional variable: name. This is why I call this type of join a mutating join. In this case, you could have got to the same place using mutate() and R's base subsetting:

```r
flights2 %>%
  select(-origin, -dest) %>%
  mutate(name = airlines$name[match(carrier, airlines$carrier)])
```

#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier
#>   <int> <int> <int> <dbl>   <chr>   <chr>
#> 1  2013     1     1     5  N14228      UA
#> 2  2013     1     1     5  N24211      UA
#> 3  2013     1     1     5  N619AA      AA
#> 4  2013     1     1     5  N804JB      B6
#> 5  2013     1     1     6  N668DN      DL
#> 6  2013     1     1     5  N39463      UA
#> # ... with 3.368e+05 more rows, and 1 more variable:
#> #  name <chr>

But this is hard to generalize when you need to match multiple variables, and takes close reading to figure out the overall intent. The following sections explain, in detail, how mutating joins work. You'll start by learning a useful visual representation of joins. We'll then use that to explain the four mutating join functions: the inner join, and the three outer joins. When working with real data, keys don't always uniquely identify observations, so next we'll talk about what happens when there isn't a unique match. Finally, you'll learn how to tell dplyr which variables are the keys for a given join.
Inner join

The colored column represents the "key" variable: these are used to match the rows between the tables. The gray column represents the "value" column that is carried along for the ride. In these examples I'll show a single key variable and single value variable, but the idea generalizes in a straightforward way to multiple keys and multiple values.

A join is a way of connecting each row in \( x \) to zero, one, or more rows in \( y \). The following diagram shows each potential match as an intersection of a pair of lines:

(If you look closely, you might notice that we've switched the order of the key and value columns in \( x \). This is to emphasize that joins match based on the key; the value is just carried along for the ride.)

In an actual join, matches will be indicated with dots. The number of dots = the number of matches = the number of rows in the output.

Inner Join

The simplest type of join is the inner join. An inner join matches pairs of observations whenever their keys are equal:
The most commonly used join is the left join: you use this whenever you look up additional data from another table, because it preserves the original observations even when there isn’t a match. The left join should be your default join: use it unless you have a strong reason to prefer one of the others.

Another way to depict the different types of joins is with a Venn diagram:

However, this is not a great representation. It might jog your memory about which join preserves the observations in which table, but it suffers from a major limitation: a Venn diagram can’t show what happens when keys don’t uniquely identify an observation.

Outer joins
Only the existence of a match is important; it doesn’t matter which observation is matched. This means that filtering joins never duplicate rows like mutating joins do:

The inverse of a semi-join is an anti-join. An anti-join keeps the rows that don’t have a match:

Anti-joins are useful for diagnosing join mismatches. For example, when connecting flights and planes, you might be interested to know that there are many flights that don’t have a match in planes:

```r
flights %>%
  anti_join(planes, by = "tailnum")
  %>%
  count(tailnum, sort = TRUE)
```

#> # A tibble: 722 × 2
#>   tailnum     n
#>     <chr> <int>
#> 1    <NA>  2512
#> 2  N725MQ   575
#> 3  N722MQ   513
#> 4  N723MQ   507
#> 5  N713MQ   483

```
Most popular relational database management systems
• SQL (Structured English Query Language) is a domain-specific language for defining, managing, and manipulating, data in relational databases.

• Developed at IBM in the early 1970s. Popularized by Oracle in the late 1970s.

• Based on Codd’s twelve rules (actually, 13, from 0 to 12) of a RDBMS:

  https://computing.derby.ac.uk/c/codds-twelve-rules/.


• Many different implementations (both open source and commercial) with some differences in syntax and adherence to current standard.
Good implementations follow the ACID (Atomicity, Consistency, Isolation, Durability) standard:

1. Atomicity: either all operations in the database succeed or none do.
2. Consistency: a transaction in the database cannot leave the database in an inconsistent state.
3. Isolation: one transaction in the database cannot interfere with another.
4. Durability: a completed transaction persists, even after applications restart.

Thus, you can understand SQL as choosing consistency over availability in the CAP theorem (although “consistency” in ACID and the CAP theorem are slightly different concepts). Most likely, the right choice in research.
Morover SQL has more procedural instructions than originally.

In fact, SQL, after the introduction of Persistent Stored Modules (PSMs), is Turing complete.

Also, over time, SQL has incorporated many object-oriented features ⇒ object-relational database management system (ORDBMS).

Distributed computation: Apache Drill at https://drill.apache.org/ (also for many NoSQL databases).

You can try basic SQL instructions at http://sqlfiddle.com/.
Open-source implementations I: PostgreSQL


- Evolved from the Interactive graphics and retrieval system (Ingres) project at Berkeley, led by Michael Stonebraker (Turing Award 2014).

- Powerful ORDBMS implementation that can handle the most complex tasks.

- Available for all OS (for instance, it is the default in macOS Server).

- Highly extensible: user-defined data types, custom functions, and allows for programming in different languages (including the definition of DSLs).

- Many add-ons, such as the PostGIS geospatial database extender.

- Multiversion concurrency control (less important for economists unless you have many coauthors and RAs).
Open-source implementations II: SQLite

- Available at [https://sqlite.org/about.html](https://sqlite.org/about.html), but pre-installed in macOS and most Linux distributions.

- Current release: 3.25.2.

- Uses PostgreSQL as a reference platform, but SQLite is serverless.

- Extremely popular, as it does not require a client-server engine (it is contained in a C programming library) and its installation is rather compact and with “zero configuration.” Attractive features for economics.

- Bindings for all popular programming languages.

- Faster than regular file I/O in your operating system with a carefully designed application file format: a complete SQLite database is stored in a single cross-platform disk file.
To lunch command-line shell

```bash
sqlite3
```

You can also add commands after `sqlite3` as in any other Unix/Linux program.

To exit:

```bash
sqlite> .exit
```

Alternative GUI ⇒ SQLite Studio: https://sqlitestudio.pl/
Some SQLite instructions: basic interaction II

Help:

```sql
sqlite> .help
```

To read commands from script files:

```sql
sqlite> .read myfile
```

To print a string:

```sql
sqlite> .print STRING
```

To load a file:

```sql
sqlite> .output FILENAME
```

Finally, to comment:

```sql
sqlite> -- This is a comment
```
Some SQLite instructions: basic interaction III

To check existing databases and associated files:

```
sqlite> .databases
```

To create a database

```
sqlite3 Economists.db
```

To check existing tables:

```
sqlite> .tables
```

To check schema of tables:

```
sqlite> .schema
```

In practice, you automatize the task we will describe below with script files and mixing-in your favorite programming language.
To create a table:

```
sqlite> CREATE TABLE Faculty (  
Name TEXT, NOT NULL,  
Age INTEGER CHECK (Age=>0 and Age<100),  
Field CHAR (20),  
PhD CHAR (25),  
PRIMARY KEY(Name),  
FOREIGN KEY(id));
```

Note:

1. Capital case, optional (SQLite is mainly case insensitive) but common.

2. Keys are also optional.

3. SQLite uses dynamic typing. Most SQL database engines use static, rigid typing. I am following here standard typing convention in SQL and relying on affinity rules.
Beyond the standard types (text, integer, character, XML,...), we can define our own types:

```
sqlite> CREATE ROW TYPE personalAddress (  
    Street CHARACTER VARYING (25),  
    City CHARACTER VARYING(20),  
    State CHARACTER (2),  
    PostalCode CHARACTER VARYING (9));
```

To alter a table (note: some of the options of ALTER TABLE are not supported by SQLite):

```
sqlite> ALTER TABLE Faculty ADD COLUMN Phone INTEGER;
sqlite> ALTER TABLE Faculty ADD COLUMN personalAddress addr_type;
```

To drop a table:

```
sqlite> DROP TABLE Faculty;
```
Some SQLite instructions: DML - Data Manipulation Language

To insert record:

```sql
sqlite> INSERT INTO Faculty (Name, Age, Field, PhD)
VALUES('Adam Smith', 35, 'Economics', 'Glasgow');
VALUES('David Ricardo', 42, 'Economics', 'London');
```

To modify record:

```sql
sqlite> UPDATE Faculty SET Name = 'David Ricardo' WHERE Name = 'Adam Smith';
sqlite> UPDATE Faculty SET Age = Age+1;
```

To delete record:

```sql
sqlite> DELETE FROM Faculty WHERE Field = 'Economics';
```
Some SQLite instructions: DQL - Data Query Language I

To list records:

sqlite> SELECT Name, Field FROM Faculty;

To select records:

sqlite> SELECT * FROM Faculty ORDER BY Age ASC;
sqlite> SELECT * FROM Faculty WHERE Age>50;
sqlite> SELECT * FROM Faculty WHERE Age>50 ORDER BY Age DESC;
sqlite> SELECT Name FROM Faculty WHERE Name ~ 'A.*'
sqlite> SELECT MIN(Age) FROM Faculty;
sqlite> SELECT MAX(Age) FROM Faculty WHERE Field = 'Economics';
sqlite> SELECT Field AVG(Age) FROM Faculty GROUP by Field;
sqlite> SELECT Field AVG(Age) FROM Faculty GROUP by Field HAVING COUNT(*)>2;
sqlite> SELECT Field AVG(Age) AS avg_age, COUNT(*) as size FROM Faculty GROUP WHERE Age>50 by Field HAVING COUNT(*)>2 ORDER BY Age DESC;
SELECT dept, AVG(gpa) AS avg_gpa, COUNT(*) AS size
FROM students
WHERE gender = 'F'
GROUP BY dept
HAVING COUNT(*) > 2
ORDER BY avg_gpa DESC

What does this compute?

To (inner) join records:

```sql
sqlite> SELECT Name Dues FROM Faculty INNER JOIN AmericanEconomicAssociation on Faculty.Name = AmericanEconomicAssociation.Name;
```

Similar instructions for cross and outer joins.

You can insert NULL

```sql
sqlite> INSERT INTO Faculty (Name, Age, Field, PhD) VALUES('J.M. Keynes', NULL, 'Economics', 'Cambridge');
sqlite> SELECT * FROM Faculty WHERE Age IS NOT NULL;
```
SQL JOINS

**LEFT JOIN**

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

**INNER JOIN**

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

**RIGHT JOIN**

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

**FULL OUTER JOIN**

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```
You can create your own views:

```sql
sqlite> CREATE VIEW Econ_Faculty_View AS 
SELECT Name, Age 
FROM Faculty 
WHERE Field = 'Economics';
```

The select can be as sophisticated as you want or subselect from the view.

```sql
sqlite> SELECT * FROM Econ_Faculty_View;
```

You cannot, however, to DELETE, INSERT or UPDATE statements on a view.

To drop a view:

```sql
sqlite> DROP VIEW Econ_Faculty_View;
```
You can run SQL in R or R in the SQL server.

The former approach is probably more common in research.

Check:


In addition, new versions of RStudio integrate interaction with SQL.
Package `dplyr`: provides a flexible grammar of data manipulation centered around data frames. In particular, `dplyr` allows you to translate the `dplyr` verbs into SQL queries and use the SQL Engine to run the data transformations. You need to install `dbplyr` (a backend for databases) as well: it translates R code into database-specific variants.

Package `RSQLite`: embeds the SQLite database engine in R and provides an interface compliant with the DBI package (a database interface definition for communication between R and relational database management systems).

Package `odbc`: provides a DBI-compliant interface to Open Database Connectivity (ODBC) drivers, including SQL Server, Oracle, and MySQL (and also PostgreSQL, SQLite).

Package `dbplot`: allows to process the calculations of a plot inside a database.
Use dplyr to interact with the database
Open Source Databases

<table>
<thead>
<tr>
<th>Data</th>
<th>Access &amp; Wrangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables &amp; Views</td>
<td>RMySQL package</td>
</tr>
<tr>
<td>SQL Engine</td>
<td>RPostgreSQL package</td>
</tr>
<tr>
<td></td>
<td>SQLITE package</td>
</tr>
<tr>
<td></td>
<td>bigquery package</td>
</tr>
<tr>
<td>Database</td>
<td>DBI package</td>
</tr>
<tr>
<td></td>
<td>dplyr package</td>
</tr>
</tbody>
</table>

R Studio
Commercial Databases

<table>
<thead>
<tr>
<th>Data</th>
<th>Access &amp; Wrangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables &amp; Views</td>
<td></td>
</tr>
<tr>
<td>SQL Engine</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>odbc package</td>
</tr>
<tr>
<td></td>
<td>DBI package</td>
</tr>
<tr>
<td></td>
<td>dplyr package</td>
</tr>
</tbody>
</table>

R Studio
Let us first clear everything:

```r
rm(list=ls())
```

We install required R packages:

```r
install.packages(c("dplyr", "dbplyr", "RSQLite"))
```

We load relevant packages

```r
library(dplyr)
library(dbplyr)
library(RSQLite)
```
We download a standard SQLite database used to teach and install it in a new directory:

```r
dir.create("data_class_computation", showWarnings = FALSE)
download.file(url = "https://ndownloader.figshare.com/files/2292171", destfile = "data_class_computation/portal_mammals.sqlite", mode = "wb")
```

We connect R to SQLite:

```r
mammals <- DBI::dbConnect(RSQLite::SQLite(), "data_class_computation/portal_mammals.sqlite")
```

We inspect the database:

```r
src_dbi(mammals)
```
We select some observations with SQL syntax:

```r
mySelection <- tbl(mammals, sql("SELECT year, species_id, plot_id FROM surveys"))
```

We look at the top 5 observations:

```r
head(mySelection, n = 5)
```

But it is easier to select with with `dplyr` syntax:

```r
mySelection <- tbl(mammals, "surveys")
```

We can look again at the top 5 observations:

```r
head(mySelection, n = 5)
```

You can also load the query in a R Notebook.
We pipe the selection:

```r
mySelection %>%
  filter(weight < 5) %>%
  select(species_id, sex, weight)
```

We link across tables:

```r
species <- tbl(mammals, "species")

left_join(mySelection, species) %>%
  filter(taxa == "Rodent") %>%
  group_by(taxa, year) %>%
tally %>%
collect()
```

dplyr allows to implement all four joins for dataframes with ease.
NoSQL I

- Databases not based on tabular relations.
- Originally, it meant No+SQL.
- Today most NoSQL databases include some SQL features, so most people call it Not only SQL.
- Concept existed since the 1960s (such as hierarchical databases), but it became popular in the early 2000s.
- Interesting example of move towards less abstraction.
- Why?
  1. Usually better dealing with big and distributed data because of their capability to scale and parallelize.
  2. Schemaless data representations require less planning and allow for easier ex post adjustments.
  3. Faster to code.
So you remember NoSQL

3 SQL DATABASES WALK INTO A

NoSQL BAR...

...A LITTLE WHILE LATER THEY WALK OUT.
BECAUSE THEY COULDN'T FIND A

TABLE
NoSQL II

- Instead of ACID, NoSQL follows BASE:
  
  1. Basic availability: each request gets a response (successful or not).
  2. Soft state: the state of the database changes over time, even without any input (for eventual consistency).
  3. Eventual consistency: the database may be momentarily inconsistent, but will eventually reach consistency.

- Some NoSQL such as Neo4j, though, still deliver ACID.

- NoSQL chooses availability over consistency over in the CAP theorem. Note importance for web applications.
NoSQL databases systems include a wide set of alternative approaches:

2. Key-value stores: pairs of keys and values ⇒ Redis, Memcached.
3. Wide column stores: store data in records with very large numbers of dynamic columns ⇒ Cassandra, HBase.
4. Time series DBMS: optimized for handling time series data: each entry is associated with a timestamp ⇒ InfluxDB, Graphite.
5. Graph DBMS: represent data in graph structures as nodes and edges. ⇒ Neo4j, AllegroGraph.
6. XML ⇒ MarkLogic, BaseX.
7. Search engines ⇒ Elasticsearch, Splunk.
8. Multimodel ⇒ Amazon DynamoDB, Microsoft Azure Cosmos DB.

Also, object databases (although they have not taken off).
NoSQL: popularity

DB-Engines Ranking

- Oracle
- MySQL
- Microsoft SQL Server
- PostgreSQL
- MongoDB
- DB2
- Redis
- Elasticsearch
- Microsoft Access
- Cassandra
- SQLite
- Teradata
- Splunk
- MariaDB
- Solr
- Hive
- HBase
- SAP Adaptive Server
- FileMaker
- Amazon DynamoDB
- SAP HANA
- Neo4j
- Couchbase
- Memcached
- Microsoft Azure SQL Database
- Informix
- Vertica
- Microsoft Azure Cosmos DB
- Firebird
- CouchDB
- Netezza
- Amazon Redshift
- Google BigQuery
- Spark SQL
- Impala

© October 2018, DB-Engines.com
NoSQL in economics

- Uses in economics:
  1. Graph databases, for their potential to allow us discover important relational patterns.
  2. Time Series DBMS, to deal with financial and other high-frequency data.
  3. Data collections that might chance over time in structure.

- Additional references:
  2. *Next Generation Databases: NoSQL, NewSQL, and Big Data*, by Guy Harrison.
MongoDB

- MongoDB (from “humongous”), most popular NoSQL database.


- Built around BSON, Binary JSON, a version of JSON.

- Dual structure:
  1. Documents are stored in collections using the BSON format. A collection is a group of related documents with shared indices.
  2. MongoDB collections belong to a database.

- Used, for example, by CERN to collect data from the Large Hadron Collider.

- Versatile and easy to use (expressive language for queries).
Mongo data model

```json
{
    _id: <ObjectId>,
    username: "123xyz",
    contact: {
        phone: "123-456-7890",
        email: "xyz@example.com"
    },
    access: {
        level: 5,
        group: "dev"
    }
}
```
NoSQL and R

- Less polished support than for SQL.
- Package nodbi for general backend.
- For MongoDB, we have package mongolite:

```r
install.package("mongolite")
library(mongolite)

m <- mongo("mtcars", url = "mongodb://readwrite:test@mongo.opencpu.org:43942/jeroen_test")

alldata <- dmd$find('{}')
print(alldata)

test <- dmd$find(  
  query = '{"cut" : "Premium"}',  
  fields = '{"cut" : true, "clarity" : true}',  
  limit = 5)  
print(test)
```
Spark

- Available at https://spark.apache.org/.
- Current version: 2.3.2.
- A fast and general-purpose cluster computing system.
- Modern alternative to Hadoop (although without a file management system).
- High-level APIs in Java, Scala, Python, and R.
- Interacts well with SQL and has a beautiful machine learning library, MLlib.
- Allows for real-time processing and querying.

Spark stack

Spark Core
Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines resilient distributed datasets (RDDs), which are Spark’s main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections.

Spark SQL
Spark SQL is Spark’s package for working with structured data. It allows querying data via SQL as well as the Apache Hive variant of SQL—called the Hive Query Language (HQL)—and it supports many sources of data, including Hive tables, Parquet, and JSON. Beyond providing a SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java, and Scala, all within a single application, thus combining SQL with complex analytics. This tight integration with the rich computing environment provided by Spark makes Spark SQL unlike any other open source data warehouse tool. Spark SQL was added to Spark in version 1.0.

Spark Streaming
Spark Streaming is a Spark component that enables processing of live streams of data. Examples of data streams include logfiles generated by production web servers, or queues of messages containing status updates posted by users of a web service. Spark Streaming is a key component of Spark’s real-time processing capabilities.
RDDs

• Organized around resilient distributed datasets (RDDs).

• An RDD is a collection of items distributed across computer nodes that can be manipulated in parallel.

• Operations: transformations (“map”, “filter”) and actions (“count”, “collect”).

• Why resilient? Automatically rebuilt on failure.

• It can be stored on disk or memory.

• Completely lazy evaluation.
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(xrange(0, NUM_SAMPLES)) \n    .filter(inside).count()

print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
Spark and R
Let us first clear everything:

```r
rm(list=ls())
```

We install required Spark package:

```r
install.packages("sparklyr")
```

We load relevant package and install Spark:

```r
library(sparklyr)
spark_install(version = "2.3.0")
```
We connect to Spark:

```r
sc <- spark_connect(master = "local")
```

We install package with some cute data:

```r
install.packages(c("nycflights13"))
```

We load relevant package and install Spark:

```r
library(dplyr)
flights_tbl <- copy_to(sc, nycflights13::flights, "flights")
src_tbls(sc)
```

Some piping:

```r
flights_tbl %>% filter(dep_delay == 2)
```
Many of the most popular algorithms in machine learning are coded in reliable, state-of-the-art libraries.

Most famous:


Note, however, that if you are going to write frontier papers in machine learning, chances are you will need to write much (most?) of your own code.
GIS

• Geographic information systems (GIS) capture, store, manipulate, and display geographic information data.

• Goes back to John Snow’s 1855 map of the Soho cholera outbreak.

• Why current boom? Spatial econometrics and quantitative spatial economics:
  
  1. *A Primer for Spatial Econometrics: With Applications in R*, by Giuseppe Arbia.
  

• Resources:
  
  
  2. https://gisgeography.com/
John Snow, cholera epidemics 1858
The effects of the Mita 1

**FIGURE 1.** The mita boundary is in black and the study boundary in light gray. Districts falling inside the contiguous area formed by the mita boundary contributed to the mita. Elevation is shown in the background. This discrete change suggests a regression discontinuity (RD) approach for evaluating the long-term effects of the mita, with the mita boundary forming a multidimensional discontinuity in longitude–latitude space. Because validity of the RD design requires all relevant factors besides treatment to vary smoothly at the mita boundary, I focus exclusively on the portion that transects the Andean range in southern Peru. Much of the boundary tightly follows the steep Andean precipice, and hence has elevation and the ethnic distribution of the population changing discretely at the boundary. In contrast, elevation, the ethnic distribution, and other observables are statistically identical across the segment of the boundary on which this study focuses. Moreover, specification checks using detailed census data on local tribute (tax) rates, the allocation of tribute revenue, and demography—collected just prior to the mita’s institution in 1573—do not find differences across this segment. The multidimensional nature of the discontinuity raises interesting and important questions about how to specify the RD polynomial, which will be explored in detail.

Using the RD approach and household survey data, I estimate that a long-run mita effect lowers equivalent household consumption by around 25% in 1864.
The effects of the Mita II

The results can be seen graphically in Figure 2. Each subfigure shows a district-level scatter plot for one of the paper’s main outcome variables. These plots are the three-dimensional analogues to standard two-dimensional RD plots, with each district capital’s longitude on the x axis, its latitude on the y axis, and the data value for that district shown using an evenly spaced monochromatic color scale, as described in the legends. When the underlying data are at the microlevel, I take district-level averages, and the size of the dot indicates the number of observations in each district. Importantly, the scaling on these dots, which is specified in the legend, is nonlinear, as otherwise some would be microscopic and others too large to display. The background in each plot shows predicted values, for a finely spaced grid of longitude–latitude co-
The effects of the Mita III

The image displays various maps with data points and color gradients indicating different variables. The maps are labeled as follows:

- (c) Haciendas (1940)
- (f) Education (1876)
- (g) Road Density (2000)
- (h) Age Market Participation (1994)

The text continues:

...from a regression of the outcome variable under consideration on a cubic polynomial in longitude–latitude and the mita dummy. In the typical RD context, the predicted value plot is a two-dimensional curve, whereas here it is a three-dimensional surface, with the third dimension indicated by the color gradient. The shades of the data points can be compared to the shades of the predicted values behind them to judge whether the RD has done an adequate job of averaging the data across space. The majority of the population in the region is clustered along the upper segment of the mita boundary, giving these...
The effects of the Mita IV

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;100 km of Bound.</td>
<td>&lt;75 km of Bound.</td>
<td>&lt;50 km of Bound.</td>
<td>&lt;100 km of Bound.</td>
</tr>
<tr>
<td>(1)</td>
<td>(2)</td>
<td>(3)</td>
<td>(4)</td>
<td>(5)</td>
</tr>
<tr>
<td><strong>Mita</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>−0.284</td>
<td>−0.216</td>
<td>−0.331</td>
<td>0.070</td>
</tr>
<tr>
<td></td>
<td>(0.198)</td>
<td>(0.207)</td>
<td>(0.219)</td>
<td>(0.043)</td>
</tr>
<tr>
<td><strong>R^2</strong></td>
<td>0.060</td>
<td>0.060</td>
<td>0.069</td>
<td>0.051</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Panel A. Cubic Polynomial in Latitude and Longitude</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mita</strong></td>
<td>−0.337***</td>
<td>−0.307***</td>
<td>−0.329***</td>
<td>0.080***</td>
</tr>
<tr>
<td></td>
<td>(0.087)</td>
<td>(0.101)</td>
<td>(0.096)</td>
<td>(0.021)</td>
</tr>
<tr>
<td><strong>R^2</strong></td>
<td>0.046</td>
<td>0.036</td>
<td>0.047</td>
<td>0.049</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Panel B. Cubic Polynomial in Distance to Potosí</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mita</strong></td>
<td>−0.277***</td>
<td>−0.230**</td>
<td>−0.224**</td>
<td>0.073***</td>
</tr>
<tr>
<td></td>
<td>(0.078)</td>
<td>(0.089)</td>
<td>(0.092)</td>
<td>(0.023)</td>
</tr>
<tr>
<td><strong>R^2</strong></td>
<td>0.044</td>
<td>0.042</td>
<td>0.040</td>
<td>0.040</td>
</tr>
<tr>
<td>Geo. controls</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Boundary F.E.s</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Clusters</td>
<td>71</td>
<td>60</td>
<td>52</td>
<td>289</td>
</tr>
<tr>
<td>Observations</td>
<td>1478</td>
<td>1161</td>
<td>1013</td>
<td>158,848</td>
</tr>
</tbody>
</table>
• QGIS, current version: 3.20.

• Check https://qgis.org/en/site/. Also, note large number of pluggins.

• Works with PostGIS, which adds support for geographic objects to the PostgreSQL.

• Alternative: to work directly in Python or R.

• Check https://www.jessesadler.com/post/gis-with-r-intro/.