Databases!

William Lowe

Hertie School

26th September 2020

Databases

Where the data lives

- → Old old technology: *Relational* databases
- → New new technology: Everything else

Think linear vs non-linear functions

Databases

Where the data lives

- → Old old technology: *Relational* databases
- → New new technology: Everything else

Think linear vs non-linear functions

Small secret: the world still runs on relational databases

Larger secret: and will continue to do so for the foreseeable future because they work *really well* for most sorts of data

Relational Databases

We should distinguish:

→ The data structure: tables, columns, keys, normal forms

→ The data manipulations: selects, joins, grouping

→ The database system, e.g. Postgres, Oracle, SQL Server, sqlite

→ The query language, e.g. SQL

→ Server and embedded systems, e.g. Oracle vs sqlite

We *won't* cover SQL itself or any particular database system.

→ Technology and fashions change

We *will* cover the essential structures and manipulations, and a little of the system interfaces

Non-relational Databases

These differ mostly in

- → The data structure: documents, key-value stores, graphs, tuples
- → The data manipulations, e.g. search, query
- → The query language, e.g. native code, json,

But often the same database system (Postgres, Oracle, etc.) has a non-relational module/add-on

Otherwise, dedicated 'NoSQL' systems. Some established representatives

- → Couch, Mongo (documents)
- → Redis, Ignite (key value)
- → Neo4j (graphs)

Relations and Tables

The data structures: table

- → You've met tables before – hint `data.frames` are tables: equal length variable-typed columns
- → Columns have names and types
- → No row names or numbers.

Relations and Tables

The data structures: table

→ You've met tables before – hint `data.frames` are tables: equal length variable-typed columns

→ Columns have names and types

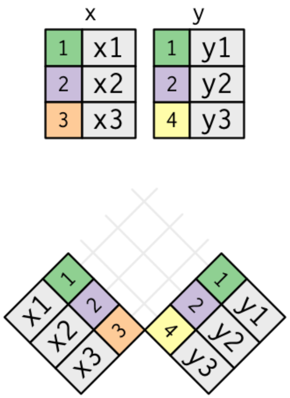
→ No row names or numbers.

A *primary* key is either

1. A column containing a (usually autogenerated) identifiers that uniquely identify each row. (Unordered, despite what they might look like)
2. Several substantively meaningful columns whose row values *taken together* uniquely identify each row

Relations and Tables

Abstractly, we'll think of them all like this



Modelling with Tables

In a table of OECD annual multi-sector investment data a primary key might be

- → country, year, sector

country and year would be insufficient:

- → would not pick out a single row

country year, sector, and value *would* be be unique, but impractical

The useful distinction is between identifying the row, and assigning some other values to it using the non-key columns.

Autogenerated primary keys (maybe just id) are the easiest to deal with, and we'll mostly assume these.

About those rows

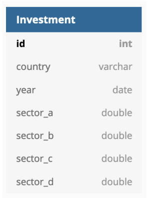
We could have represented each of the 4 sector's investment values for each country and year as four extra columns

That would have been *bad*

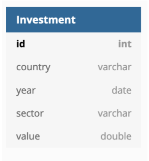
→ Sectors can't change without rebuilding the whole database structure

We can easily allow adding, removing, and renaming sectors by *making sector a value not a variable*

We try to extend downwards as new rows, not sideways as new columns



Investment	
id	**int**
country	varchar
year	date
sector_a	double
sector_b	double
sector_c	double
sector_d	double



Investment	
id	**int**
country	varchar
year	date
sector	varchar
value	double

Translation manual

For economists

- wide to long: make variables values
- long to wide: make values variables

For dplyr-ers

- gather, pivot_to_long: make variables values
- spread, pivot_to_wide: make values variables

This decision to make sector a variable, and value another one (rather than sector1_value, sector2_value, etc.) is called *normalization*

It's a whole bunch of database theory

- Check out 1st through 6th *normal form*
- yes, the tidyverse stole the best bits

Foreign keys

We were a bit vague about how country was represented in the OECD investment data

- → was it a string?
- → If so are we using names, or ISO codes, or what?

Now we have the opportunity to make it a real thing. Let's say

- → a country is represented by a row in the Country table

Let's model Investment as having columns

- → country_id, year, sector, value

where country_id refers to row of Country

How does it do that?

Foreign keys

Tables are related by *foreign key* relationships

→ A foreign key is a column containing primary key(s) *from another table*

Example

→ Table Investment contains annual multisector investment data

→ Table Country contains facts about each country, e.g. name, political system

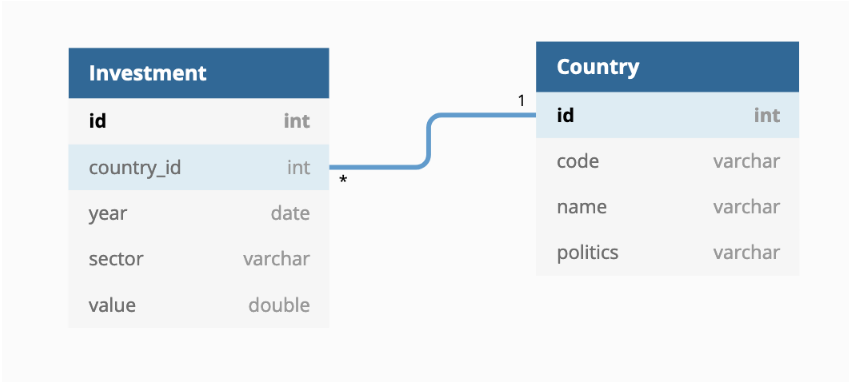
Assume 20 years of investment data on 33 OECD countries[1] in 4 sectors

→ Table Investment has 2640 rows

→ Table Country has 33 rows

[1] I know. This is not quite right because four countries joined in 2010

Foreign keys



Schema

These sorts of arrangements are the database *schema*

Schemas

- → express a model of the substantive domain
- → define tables, and the columns within them
- → are tricky to change without disruption
- → built and maintained by a Database Administrator

Data

- → represent entities of the types defined in the schema
- → are represented as rows
- → are fairly easy to add, edit, and remove without disruption
- → maintained by Developers and Data entry folk

Relational data modeling

Many rows of Investment refer to the same 1 row of Country, so this is a *many-to-one* relationship

→ Implemented by adding a foreign key column country_id to Invest and filling it with the relevant primary key from Country (called id there)

If you've programmed an object-oriented language, think of the foreign key as a *reference* to a Country object

If you've worked on ontology before, this is a *has-a* relationship rather than an *is-a*

→ Relational databases understand the world *entirely* in these terms

Relationship status: it's complicated

The *many-to-one* relationship is key to the expressiveness and efficiency of a relational system, but sometimes we have to work to make it represent what we want to represent

Example: Countries sign treaties, join organizations, and group together in various ways

→ How to represent that?

Let us count the ways

→ Dummy variable strategy: new column called WTO that is true or false

Not great because there's lots of things to record about a country's relationship to the WTO and there are other organizations

→ e.g. joining date, leaving date

and all we've got is a yes / no (or a string)

Relationship status: it's complicated

How about the

→ Foreign key strategy: new foreign key relationship a table representing the WTO

But then we can't represent the joining and leaving dates for any particular country without even more columns

Also, why would we have a one row table called WTO?

Relationship status: it's complicated

Our problem is that countries have *many-to-many* relationship with organizations:

- → Each country can be a member of more than one organization
- → Each organization can have more than one country

Reconceptualize in terms of *membership*

- → A membership connects a country to an organization
- → It is the *membership* that has beginning and ending dates of a *membership*

So represent membership as…a table Membership, with columns:

- → country_id (a foreign key to a row of Country)
- → organization: the name of the organization
- → begin_date and end_date for that country

We can now represent as many organizations and organizational memberships as we like

Membership has its privileges

There's still something not quite right about this picture

- → Organizations are no more than things to belong to
- → What if we wanted to record something about the organization *itself*, independent of its members?
- → How do we stop the data entry folk from adding 'World Trade Organization' and 'WTO' and 'W.T.O' as separate entries?

Organizations need to be more real. And you become real in a database model by being…a table

Membership has its privileges

So we make another table Organization with columns full_name, abbreviation, start_date, end_date

Now each Membership has columns:

→ country_id (a foreign key to a row of Country)

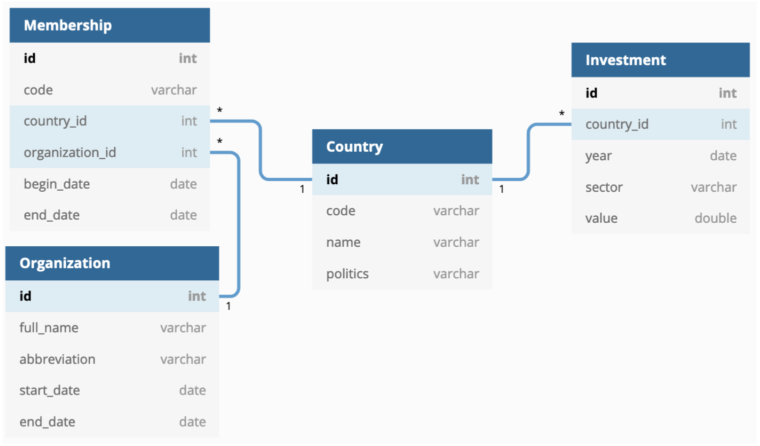
→ organization_id (a foreign key to a row of Organization)

→ begin_date: when the country joined

→ end_date: when the country left

Two many-to-one relations and an intermediate 'join' table completely represents the *many-to-many* relationship between countries and organizations

Foreign keys



Joined up modelling

Great, we've got our domain modelled, how to get data out of it?

→ it's query time

The basic query type for database is (row) selection

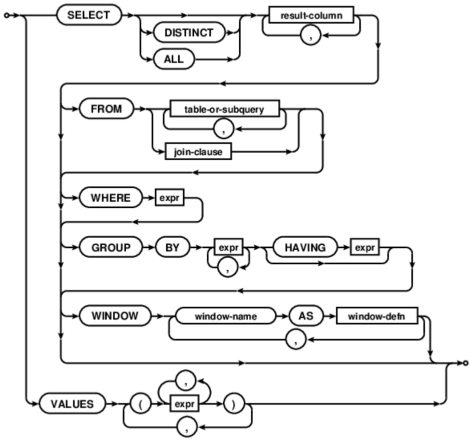
In SQL it is the SELECT command

→ Equivalent to dplyr's filter, select, group_by, summarise, and arrange functions

Built for single tables

For information spread out *across* tables we'll need to join up the information so that select can operate on it

Joined up modelling

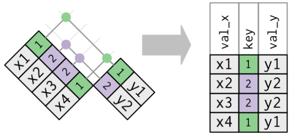


The ‘core’ SELECT function syntax, from the sqlite3 homepage (link).

Joins

So if we joined Investment to Country on country we would get a table with the same number of rows as Investment, but extra columns corresponding to facts about the country mentioned in each row of Investment

There would be duplicated country information of source, but it would all be (redundantly) inline and available for filtering and selecting together.

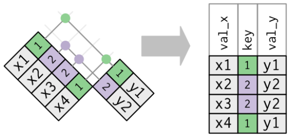


Joins

So if we joined Investment to Country on country we would get a table with the same number of rows as Investment, but extra columns corresponding to facts about the country mentioned in each row of Investment

There would be duplicated country information of source, but it would all be (redundantly) inline and available for filtering and selecting together.

```
SELECT
  inv.year, inv.sector, inv.value,
  cou.name, cou.politics
FROM
  Investment inv
LEFT JOIN
  Country cou
ON
  inv.country_id = cou.id;
```

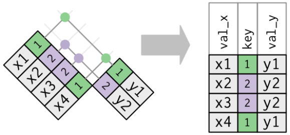


Joins

So if we joined Investment to Country on country we would get a table with the same number of rows as Investment, but extra columns corresponding to facts about the country mentioned in each row of Investment

There would be duplicated country information of source, but it would all be (redundantly) inline and available for filtering and selecting together.

```
SELECT
  inv.year, inv.sector, inv.value,
  cou.name, cou.politics
FROM
  Investment inv
LEFT JOIN
  Country cou
ON
  inv.country_id = cou.id;
```



val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1

```
left_join(Investment, Country,
    by = c("country_id" = "id")) %>%
  select(year, sector, value,
           name, politics)
```

Joins

In our example, if we want to select all countries that exceeded some threshold value in sector A while being a member of the WTO (phew) then we need to connect several

One (not the best) sequence of operations would be

- → Join Investment and Country and call it InvCountry
- → Select the rows of InvCountry with values above threshold in Sector A and only keep the unique ones
- → Join Organization to Membership and call it OrgMem (gets the real names of the each organization)
- → Join OrgMem to InvCountry where the organization name is 'WTO', keeping only unique country names

You can think of other ways to do this

- → Some are much more computationally efficient than others. This is 'query optimization'

Speed

One of the key reasons people use relational representations is speed

- → Underneath, tables are not stored as tables but tree structures for fast retrieval
- → These structures have been optimized for more than 20 years

If things are not going fast enough, then we can add *index*es to particular variables that we tend to select often

- → Advantage: speed
- → Disadvantage: space

(Databases are generally a compromise between space and time)

Views

Because the result of every join is another table it is sometimes useful to keep the results around

This derived table (a table made by joining other table) is called a *database view* (not all databases have them)

Strengths:

→ We get a non-normalized table, but it's only pretend. Our real data is efficiently stored

→ Data entry doesn't see views so it is sane while the queries using the view are fast

Limitations:

→ You can't usually add or edit a view, just read from it

→ It's extra computational work to keep the view current and human work to decide what business purposes deserve one

Guarantees

Traditional database systems offer *guarantees*, usually summarised as ACID

- → Atomicity: If a country changes its name, the unit of work is [remove old name, add new name]. There's no time where the country has no name, and they 'roll back' together
- → Consistency: There's never a point where the tables represent something impossible in the domain, e.g. if we remove the WTO row from Organization then all the Membership rows that contain its key will be deleted - the 'cascade'
- → Isolation: Even if the database does things in parallel it should never be possible to tell
- → Durability: If something happened, then the system crashed, it should come back up still happened

Often hard to maintain at scale

NoSQL systems often deliberately violate these, e.g. offering 'eventual consistency' or lacking durability

Connect, only connect

Databases need to communicate to the rest of the data science tools

- → by network connection (traditional)
- → by pretending to be a programming language object

(or often both)

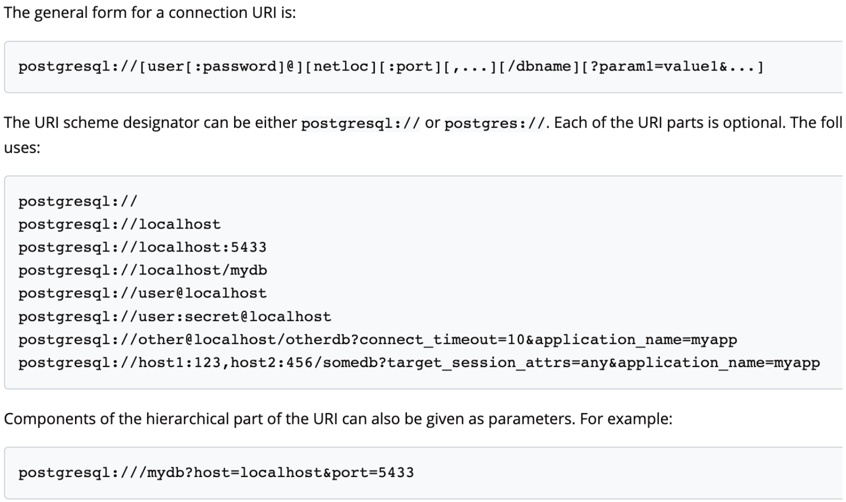
Network

Databases are usually configured as servers.

Access by specifying

- → URL with port
- → login / username
- → password
- → etc.

Postgres connection



The general form for a connection URI is:

```
postgresql://[user[:password]@][netloc][:port][,...][/dbname][?param1=value1&...]
```

The URI scheme designator can be either `postgresql://` or `postgres://`. Each of the URI parts is optional. The foll
uses:

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
postgresql://host1:123,host2:456/somedb?target_session_attrs=any&application_name=myapp
```

Components of the hierarchical part of the URI can also be given as parameters. For example:

```
postgresql:///mydb?host=localhost&port=5433
```

Network

That gets you a *database connection* that you send commands down, usually in SQL

If you run a query, you don't get results directly (there may be millions of rows), but rather a *cursor*

A cursor is an imaginary row marker for all the rows that could be returned from your query

→ You ask the cursor for a row and you get the 'next' one, or the next 50

Faking it as an object

This whole arrangement can be…awkward

In *applications* we often want to pretend that the result is a more familiar object, e.g.

- → In R we want result sets to look like data.frames
- → In Python we might want each row of the result to be an object, e.g. a Country
- → In web applications may want to populate a web page template with result rows

This is broadly referred to as the *impedance mismatch* (because that sounds more engineering that ontological mismatch)

Lots of tools do the translation for you

- → In R, dbplyr
- → In Python the *object-relational mapper*s, e.g. SQLAlchemy, Dango ORM

Summing up

Most data scientists never design an database

→ but they almost *all* end up interacting with them

→ and with DBAs

Lots of academic data work consists of working with *bad reinventions* of the relational database

→ Looking at you, Excel-jockeys

Summing up

Most data scientists never design an database

- → but they almost *all* end up interacting with them
- → and with DBAs

Lots of academic data work consists of working with *bad reinventions* of the relational database

- → Looking at you, Excel-jockeys

Surprise: Thinking about data relationally will help you with *statistics*

- → Multilevel models, time series analysis, network analysis

Tables are Concepts; columns are variables; rows are instances; foreign key relationships are nested variables, many-to-many relationships are crossed variables (possibly unbalanced)

Databases

Maybe not the most exciting technology, but awesomely useful and not going away