Introduction to SQL

This is the first of three activities to teach you about relational databases and the SQL programming language.

Videos

The videos for this activity are:

Video	Length	Slides
Introduction to databases	16 minutes	PDF
Entity Relationship Diagrams	11 minutes	PDF
Intro to SQL	29 minutes	PDF
Normal Forms	14 minutes	PDF

Exercises

- Set up the database
- ER diagrams
- More modelling
- Explore the database
- Bristol elections
- The UK census
- Normal forms

Set up the database

History

In this unit, we will be using the free database MariaDB, which is a clone of MySQL and was written by the same author - Michael "Monty" Widenius, from Finland. The history behind the naming is that Monty first created MySQL (named after his daughter, My) which became the most popular open-source database. He sold MySQL to Sun, who were in turn bought by Oracle - who are famous for their commercial database software. As a result, Monty created a clone of MySQL called MariaDB (named after his other daughter). MySQL and MariaDB are compatible in many ways and most of what we learn on this unit will apply equally to both of them, although both Oracle (for MySQL) and the open-source community (for MariaDB) have added new features to their databases.

Although we will be using MariaDB, in some places the name MySQL will still appear - most notably as the name of the command-line program that you use to access the database.

Install the database

Open your VM and install the database with the appropriate commands. On Debian that is:

```
$ sudo apt install mariadb-{server,client}
```

Advanced note

What's that funky squiggly bracket doing? Bash expands it to `mariadb-server mariadb-client`. Similarly if you typed: `cc -o hello{,.c}` it would get expanded to `cc -o hello hello.c`. There are *loads* of shorthand tricks for typing stuff quicker in a shell: try and pick them up as you go!

Question: What does `echo {a,b,c}-{1,2,3}` print? Try and guess before running it.

The `mariadb-server` package contains the server that stores the data and lets clients log in. `mariadb-client` is a command-line client (with the command name `mysql`); later on we will also use a Java client to access a database from a Java application.

The database server is now installed, but our system won't start it unless you ask it to. On Debian the service manager is *SystemD*. We can start the server running with:

```
$ sudo systemctl start mariadb
```

Check that the service is running with

```
$ sudo systemctl status mariadb
$ sudo journalctl -u mariadb
```

Set it to run by default with:

```
$ sudo systemctl enable mariadb
```

Advanced note

Whilst Linux has *more or less* standardized on using SystemD as the service manager... it is unpopular in certain quarters. Other systems exist! Alpine Linux (which was the VM image we *used* to use) uses OpenRC. The BSDs mostly do it with a hodge-podge of shellscripts. Macs use something called `launchctl`.

The argument against SystemD is that it breaks compatibility with the POSIX OS standard and goes against *The UNIX Way* (do one thing well); that the developer Lennart Poettering is a bit controversial (soft skills are important!); and quite frankly the overreach of the project is incredible. As well as managing services it can also encrypt your home folder, manage WiFi connections, manage your log files and system name and much, much, more.

The arguments for it are that it is fast, gets rid of a bunch of janky shell scripts, and standardizes things in a way that makes sense for Linux. Linux distro's used to be much more diverse but nowadays the choice is mostly what package manager do you want to use and what desktop do you want by default.

For now SystemD seems to be what we've settled on for Linux. It's mostly fine once you learn it but do try a BSD system and see what it used to be like and if you prefer it!

Security

To log in to a database, you normally need a user account and an authentication factor (such as a password, or a private key). However, in the latest version, mysql user accounts are linked to system user accounts. You should probably secure it though. Running a public-facing database without security will end in databreaches and fines quicker than you can type `metasploit`.

The default set-up will allow anyone to log in and see some of the database, for example the `test` tables but this is not particularly secure. Most distributions come with a `mysql_secure_installation` script that sets up more secure access rights. Run it and set a password for the root user (otherwise it'll be the default root password or blank).

Creating a database

Right, you have a mysql server running! Lets connect it to a database! To create the database:

```
mysqladmin -u root -p create mydatabase
```

To connect to it:

```
mysql -u root -p mydatabase
```

That should drop you into a prompt! Congratulations! You have a running database.

Importing sample data

We have prepared some sample data that we will be using in this and the following weeks.

First, download the following two files and place them in the same folder as your Vagrantfile. This folder is important as the `sample-data.sql` script contains hard-coded absolute paths starting in `/vagrant/` to import some of the data. You can download the files the same way as you did before with the secure setup file.

```
cd /vagrant
wget https://raw.githubusercontent.com/cs-
uob/COMSM0085/master/code/databases/sample-data.sql
wget https://raw.githubusercontent.com/cs-
uob/COMSM0085/master/code/databases/sampledata.tar
tar -xvf sampledata.tar
```

This creates a folder sampledata with the files we need.

If you are using a local copy of this repository, you can also find the files under `/code/databases`.

The `tar` file is a *tape archive*: a file that contains further files and folders, as if it were a folder itself.

Advanced note

The options here are x=extract a file, v=verify (print the name of every processed file to standard output), f=the filename is in the following argument. You may sometimes see this command without the '-' for these options -- this works because `tar` supports an older convention where options are not prefixed with a dash, but to be safe you should stick to the modern convention (which it also understands).

To create a tar file yourself, the command would be `tar -cvf ARCHIVE.tar FILE1 FILE2...` where c=create the archive if it doesn't exist, and assume all arguments not consumed by another flag refer to files to be added. In fact, `tar -xvf ARCHIVE.tar FILES...` also works and only extracts the named files from the archive.

Load the sample data with the following command:

```
mysql -u root -p -e 'source /vagrant/sample-data.sql'
```

This pulls in some data in CSV files (have a look at the script if you want) and creates a default user "vagrant" who can log in to the database without a password, but can only read and not write the two sample databases "census" and "elections". There is another database called "data" which starts out empty, and "vagrant" can both read and write it.

Advanced note

It also seems to throw a bunch of errors and was left to us by the people who previously ran the unit. It's on our list of jobs to fix, but it seems to work anyway? YOLO. Pull requests appreciated.

You can now log in to the database and try the following:

- `mysql` on its own logs you in, you now get the database prompt `MariaDB [(none)]>` to show that you haven't loaded a particular database yet.
- `SHOW DATABASES;` on the database prompt gives you a list of databases which you have access too.
- Select one with the USE command, for example `USE elections;`. Your prompt should now read `MariaDB [elections]>`.
- `SHOW TABLES;` will show the tables in the selected database.
- There's a Party table in the elections database, so `SELECT * FROM Party;` will show you the data.
- You could also use `DESCRIBE Party;` to show a list of columns and types in the Party table.
- Finally, `exit` or Control+D on a line of its own gets you back to the shell.

You can open the SQL and CSV files under `/vagrant/sampledata` to compare with the output you get from MariaDB. Study this until it makes sense to you. The `setup.sql` files contain the database schemas and the `import.sql` ones pull in the sample data.

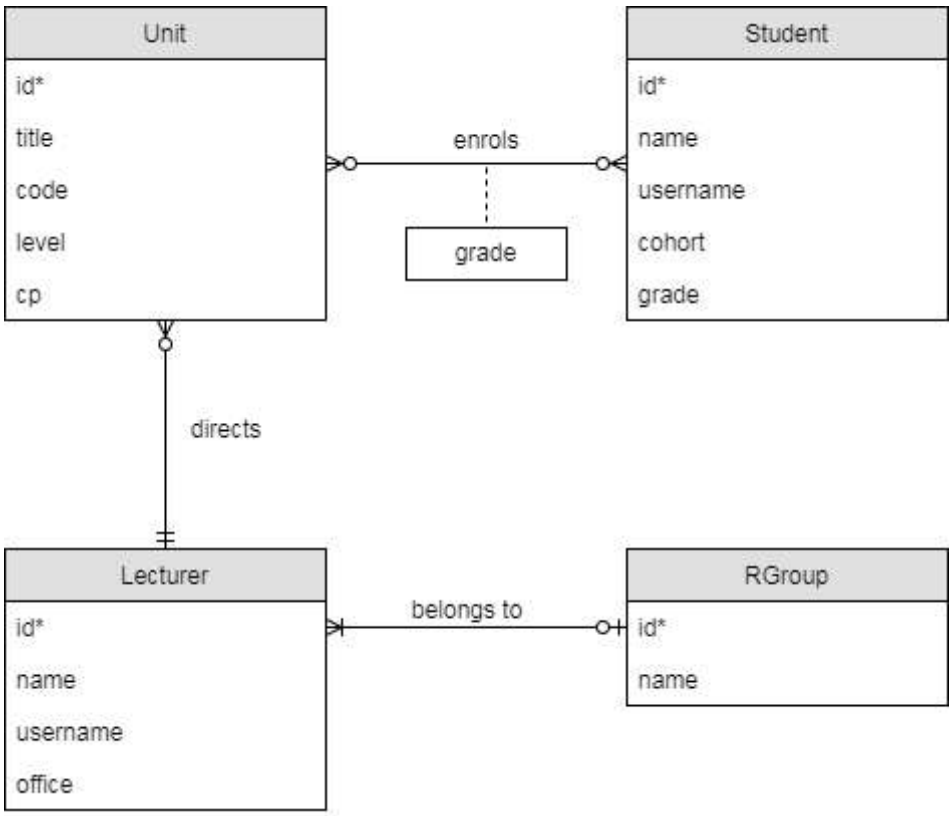
On a lab machine

On a lab machine, to save disk space your VMs may not remain between reboots - and because they are not hosted on the network file system, if you log in to a different machine next time, your changes will not be saved either but you will get the VM reinstalled from

scratch. To make sure the database is ready whenever you need it, open the `Vagrantfile` in a text editor and add the setup commands to the provisioning script. The commands are the same ones that we have done just now manually.

Reading an ER diagram

Here is an ER diagram for a fictional university database:



The foreign key columns are not included in the tables - in this diagram, they are implied by the relationships, e.g. the `Unit.director` column comes from the "directs" relationship.

Looking at the diagram and the table schemas, answer the following questions for yourself:

- Which relationships are mandatory or optional? (For example, must every unit have at least one student enrolled?)
- Which relationships are one-one, one-many or many-many?
- How do the above affect the placement of foreign keys? For example, why is the foreign key for "lecturer belongs to research group" on the Lecturer table?

Drawing an ER diagram

Draw an ER diagram for the following scenario.

> The University of Bristol Hoverboard Society (HovSoc) wants to create a database to manage its membership and events. Each member has a name, an optional student number, a contact e-mail address and a hoverboard riding skill level (represented as an integer, minimum 0). We assume that e-mail addresses are unique among members.

> The committee consists of some of the members, each of which has a unique committee role. We assume that committee roles do not change during the year and that each committee role must be filled every year.
>
> An event has a date, a name, a location, an optional description and an organiser who must be a society member (not necessarily a committee member). An event is attended by a set of members. There is never more than one event at the same location on the same date but event names are not unique.

You can draw the diagram with pen and paper or you can use a free modelling tool like draw.io.

- For draw.io, open the "Entity Relation" section in the menu on the left and use the "Table" (first item) object for tables. Clicking on it adds a table to your diagram.
- To add a row to a table, select an existing row and press Control-D (duplicate item). To delete a row, press the delete key.
- To add a relationship, select a table by clicking its header and drag one of the blue triangles that appear round the edges onto another table. You can change the type of a relationship in the details panel on the right (the "line start" and "line end" boxes).
- File/Save as lets you download your diagram in an XML-based format, which you can open and edit later. File/Export as lets you download it as an image.

Implementing a Schema

Write a CREATE/DROP script for the schema that you have just designed.

- A create/drop script starts with a sequence of DROP TABLE IF EXISTS statements followed by a sequence of CREATE TABLE scripts. The effect of running it is to make sure all tables exist and are empty, whether or not the tables existed before.
- If table A has a foreign key to table B then you must create table B before A and drop table A before dropping B. The simple way to do this is work out the CREATE order, then put all DROP statements in the exact opposite order.

Save your script as a file (the extension `.sql` is usual for SQL scripts).

To test that it works, log in to the database with `mysql` on the command line in the lab machine, from a terminal in the same folder as your create/drop script. Then run the command

```
USE data;
```

to select the (initially empty) database called `data`, on which you have read and write permissions. Note that there is a semicolon at the end.

As long as you started your MariaDB session in the folder with your script, you can now run the command `\. SCRIPTNAME.SQL`, that is a backslash, a period, a space and then the name of the script. As this is a command directly for the MariaDB client rather than a command to be run on the server, it does not take a semicolon.

If you get any errors, then `SHOW ERRORS;` will display more information. If not, check with `SHOW TABLES;` that your tables exist.

Now, run the script a second time. If the order of all commands is correct, then it should run through again without errors.

More modelling

Using what you have learnt so far about relational modelling, think about and discuss in groups how you would model a university database to store student's academic progress, such as units enrolled on and completed, marks obtained etc. based on your understanding of how the University of Bristol works. For example, a unit can have different assessments with different weights. You will of course also need a `Students` table, and you can make the model more involved if you like by including that different students are on different degree programmes, and that sometimes students have to resit units.

You should end up with a more detailed version of the model briefly shown at the top of the previous page - if you have the time you can make both an ER diagram and a create/drop script.

This is also a good point to mention another fact of how marks and credit points work: exam boards. At the end of each academic year around May, your unit directors all report their marks for each student to an exam board, which sits and decides on final marks and awarding credit. For example, an exam board can moderate the marks for a unit. This is why you do not get your exam marks until a long time after the exam has happened, even if it's a multiple choice exam that can be marked automatically: the marks still have to go through an exam board. (There is another, smaller exam board around the start of Teaching Block 2 so you don't have to wait until summer for your January exam marks to be released.) If you want to model this in your schema, the idea here is that a student has two marks associated with each unit: the "actual mark" (the input to the exam board) and the "agreed mark" (the one that comes out of the board and goes on your transcript). Of course, for most students most of the time, the two are the same. Your schema will need to store "agreed marks" explicitly, but there are ways of doing the model that does not store the "actual mark" directly. Think about how you could recompute it from other information in the database - we will of course learn how to do this in SQL in a later activity.

The key idea in relational modelling is not to store information more than once if you can avoid it. If you have stored in several places that Fred is taking Introduction to Programming, and then Fred switches his units, you don't want to end up with a situation where this change is only reflected in some parts of the database.

Explore the database

Open the virtual machine and type `mysql`. Assuming you have installed the database correctly as in the previous activity, you should see the prompt `MariaDB [(none)]>` which shows that you are connected to a database server but you have not selected a database yet.

Have a look at the databases that exist with the command

```
SHOW DATABASES;
```

Like all SQL commands, it needs a semicolon at the end. You should see four databases including `census` and `elections`. Let's select one of them:

```
USE elections;
```

SQL keywords like `USE` are not case-sensitive, but it is convention to write them in all upper case. SQL names of tables, columns etc. like `elections` however are case-sensitive. Your prompt should now show `MariaDB [elections]>`.

Have a look at the tables in this database with

```
SHOW TABLES;
```

You should see `Candidate`, `Party` and `Ward`. Let's have a look at one:

```
DESCRIBE Candidate;
```

The output will look like this:

```
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          |
+-------+--------------+------+-----+---------+----------------+
| id    | int(11)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(100) | NO   | UNI | NULL    |                |
| party | int(11)      | YES  | MUL | NULL    |                |
| ward  | int(11)      | YES  | MUL | NULL    |                |
| votes | int(11)      | YES  |     | NULL    |                |
+-------+--------------+------+-----+---------+----------------+
```

The first two columns tell you the names and types of columns in this table. The third column (Null) tells you if NULL values are allowed in this column. The Key column tells us that id is the primary key (PRI), name has a unique constraint (UNI), party and ward are foreign keys (MUL) and there are no key constraints at all on votes.

For even more information, try this:

```
SHOW CREATE TABLE Candidate;
```

The output is a bit messy but it shows you (more or less) the statement used to create the table. From here we can read off the details of the foreign keys:

```
CONSTRAINT `Candidate_ibfk_1` FOREIGN KEY (`party`) REFERENCES `Party` (`id`)
CONSTRAINT `Candidate_ibfk_2` FOREIGN KEY (`ward`) REFERENCES `Ward` (`id`)
```

So the `party` column is a foreign key pointing at the `id` column in the `Party` table.

Now let's look at some data. This command shows you all entries in the Candidate table:

```
SELECT * FROM Candidate;
```

There are 141 entries. Looking at the first one:

```
+-----+--------------------------------+-------+------+-------+
| id  | name                           | party | ward | votes |
+-----+--------------------------------+-------+------+-------+
|   1 | Patrick Dorian Hulme           |     1 |    1 |    16 |
```

The party and ward ids on their own don't tell us much, but as they are foreign keys we can use them to join on the tables that do contain this information:

```
SELECT * FROM Candidate
INNER JOIN Party ON Party.id = Candidate.party
INNER JOIN Ward ON Ward.id = Candidate.ward;
```

On the MariaDB prompt, if you don't end with a semicolon then the program assumes you want to type a command over multiple lines, and shows the continuation prompt `->` for the next ones. This also allows you to copy-paste multi-line commands from a text editor into the MariaDB client. Ending a line with a semicolon executes the query and drops you back to the main prompt.

You will now see a much longer listing, starting like this (I have shortened some columns):

```
+-----+---------------------+-------+------+-------+----+-------------+----+-
----------+------------+
| id  | name                | party | ward | votes | id | name        | id |
name      | electorate |
+-----+---------------------+-------+------+-------+----+-------------+----+-
----------+------------+
|   7 | Matthew Simon Melias |     7 |    1 |  1067 |  7 | Conservative |  1 |
Avonmouth |       9185 |
```

The first thing to note is that the results are no longer in the same order: P. D. Hulme is no longer at the top. Unless you tell the database that you want a particular order, it is allowed to choose its own one and depending on what joins you do, this might change.

There are several columns here named id, one from each of the tables - in general, doing `SELECT *` on a joined table gets you more data than you need. This would be a nicer query unless you're actially interested in the ids:

```
SELECT Candidate.name AS name,
Party.name AS party,
Ward.name AS ward,
votes,
electorate
FROM Candidate
INNER JOIN Party ON Party.id = Candidate.party
INNER JOIN Ward ON Ward.id = Candidate.ward;
```

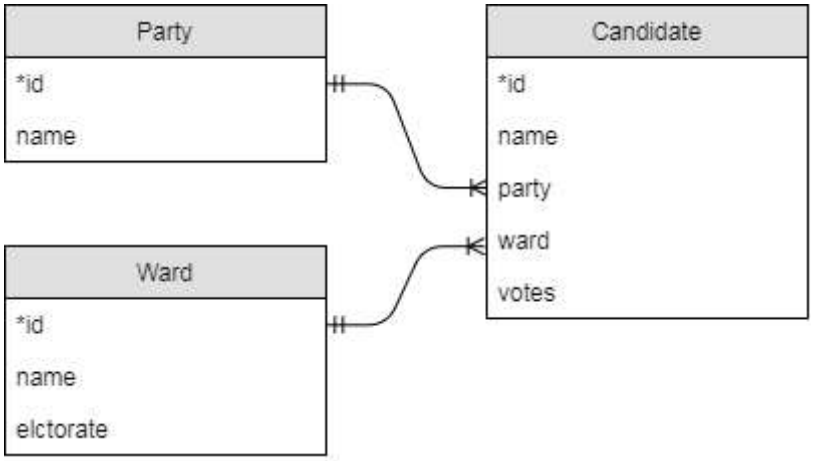
Here is the start of the output that I get:

```
+----------------------+--------------+-----------+-------+------------+
| name                 | party        | ward      | votes | electorate |
+----------------------+--------------+-----------+-------+------------+
| Matthew Simon Melias | Conservative | Avonmouth |  1067 |       9185 |
```

Explore the elections database a bit to get a feel for how the data is structured. For example, what party and ward did Patrick Dorian Hulme from above stand for? What is the schema of the elections database - you might want to draw a diagram?

Bristol elections

In 2014, Bristol held council elections for 24 of its wards. Each ward elected one councillor to represent the ward on the city council. The results are in the `elections` database, with the following schema as you have hopefully just discovered:



From an ER diagram you can derive a *JOIN strategy*, a way of representing all the useful information in a database. For individual queries, you may only need a subset of this information so you can leave off unnecessary parts of the full JOIN strategy. In this case, the following would work:

```
SELECT Candidate.name AS name, Party.name AS party, Ward.name AS ward,
Candidate.votes, Ward.electorate
FROM Candidate
INNER JOIN Party ON Candidate.party = Party.id
INNER JOIN Ward ON Candidate.ward = Ward.id
```

Exercises

Find SQL statements to answer the following questions. Your answer to each question should be a single query, and you should not hard-code any ids. For example, if a question is about the Labour party, you should use the string `'Labour'` in your query somewhere, not look up the party id and hard-code that in your query.

Although you can answer all the exercises in this section by taking the join strategy and adding clauses where necessary, there is sometimes a quicker way. But if you don't know where to start, consider how you would extract the result you want from the joined table. The WHERE clause determines which rows appear in the result and the SELECT clause picks the columns that appear in the result.

1. List the names of all parties that stood in the election, ordered alphabetically by name.
2. List the names of all parties that stood in the Bedminster ward.

3. How many votes did Labour get in the Stockwood ward?
4. List the names, parties and number of votes obtained for all candidates in the Southville ward. Order the candidates by number of votes obtained descending (winner comes first).
5. List the name, party and number of votes obtained for the winner only in the Knowle ward. *(Hint: apart from changing the ward name, you only need one small modification to the statement from the last question. You may assume no ties.)*

The 2011 UK Census

In 2011, the UK took a census of all households. We will look at the data for one particular question: "KS608 Occupation".

Background: UK Geography

The United Kingdom of Great Britain and Northern Ireland (UK) is a country that contains the individual countries England, Wales, Scotland (at the time of writing) and Northern Ireland (but not the Republic of Ireland). The census data for this question comes from the Office of National Statistics (ONS) which is for England and Wales only. Scotland and Northern Ireland have separate statistical offices.

England itself can be divided into 9 regions:

- The North West
- Yorkshire and The Humber
- The North East
- The West Midlands
- The East Midlands
- The South West
- The East
- The South East
- London

The UK used to be further divided into counties but these are much less important nowadays. In fact there is a mix of counties, unitary authorities and boroughs - 152 in total. These are all called "county-level units" (CLUs).

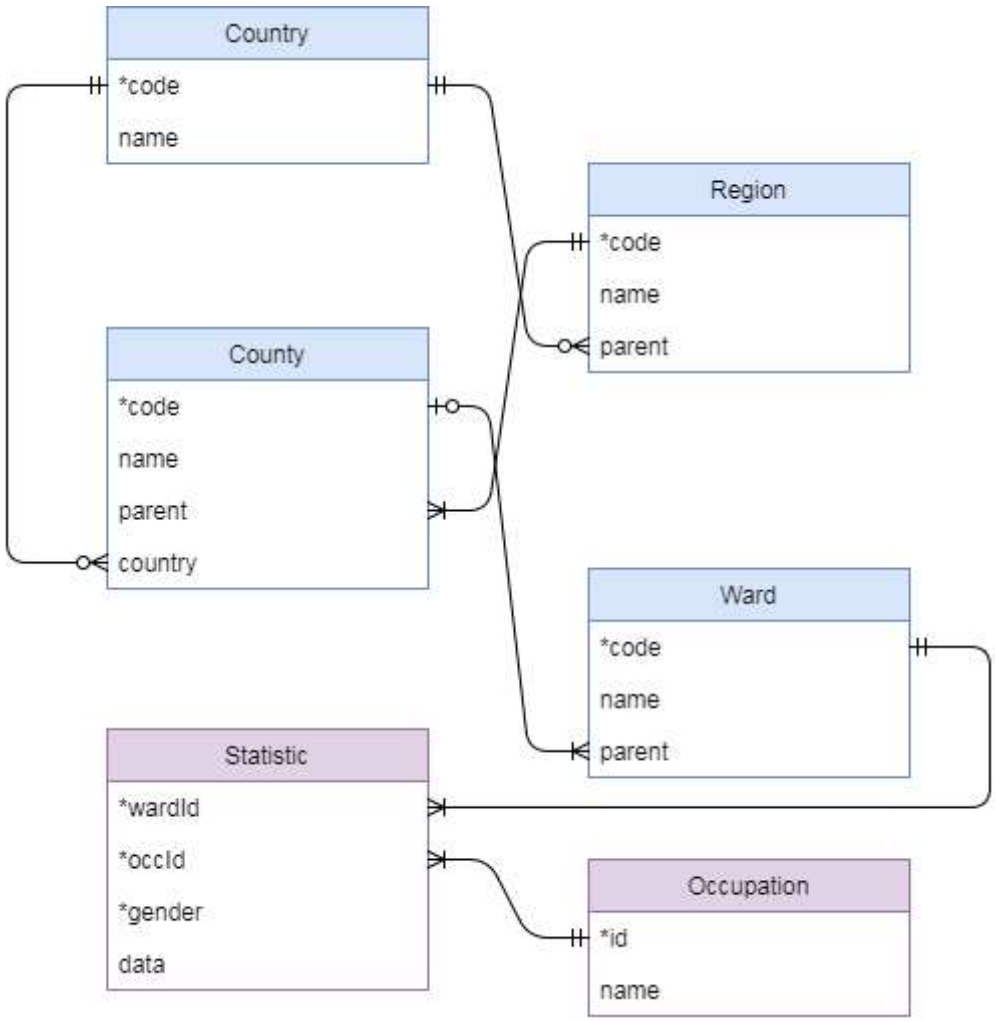
The smallest relevant unit for political and statistical purposes is the electoral ward, often simply called ward. Wards elect their own councillors (as we have seen) and national statistics are available at a ward level. There were 8570 wards at the time of the 2011 census. For example, the City of Bristol unitary authority contained 35 wards, of which the University of Bristol was in the Cabot ward.

Each statistical unit (ward, county, unitary authority etc.) is assigned a 9-character code by the ONS of the form `Xaabbbbbb` where X is E for England and W for Wales. The first two digits `aa` identify the type of unit: 05 is a ward, 06 a unitary authority; in England only E12 is a region, E08 is a borough (metropolitan), E09 is a borough of London and E10 is a county. The last 6 digits identify the individual unit. Finally, the codes for all of England and Wales are E92000001 and W92000004.

An interactive online map of the statistical units of the UK is available online at UK data explorer or DataShine. All census data for individual questions is publicly available. (The most interesting data - correlations between questions - is not all openly available due to privacy issues. Researchers can apply to the ONS to get access to particular data sets.)

Data format

The `census` database has the following schema.



For wards in England, the `parent` FK points at the county table and identifies the CLU (county-level unit) to which the ward belongs. For wards in Wales, the `Ward.parent` column is `NULL` which you need to take into account when working out a JOIN strategy.

KS608: Occupation

Question KS608 on the 2011 census asked all UK citizens in employment and between the ages of 16 and 74 to classify themselves as one of 9 occupation classes.

- Have a look at the `Occupation` table in the `census` database to see the 9 occupation classes.

The answers to the question are recorded in the `Statistic` table in the following format. Gender is 1 for women an 0 for men; in the 2011 census these where the only gender options.

```
+-----------+-------+--------+------+
| wardId    | occId | gender | data |
+-----------+-------+--------+------+
| E05000001 |     1 |      1 |   54 |
+-----------+-------+--------+------+
```

This row records that in ward `E05000001` (Aldersgate), 54 women (gender=1) said that they worked as "Managers, directors and senior officials" (occId=1).

Note how (in the ER diagram) the primary key of the Statistic table is a composite of three columns (wardId, occId, gender). This is exactly what you would expect in a table that has one data value per ward, occupation class and gender.

Exercises

1. The university of Bristol is situated in the `Cabot` ward (ward names are not always distinct, but this one is). Find the names and codes of the CLU, region and country containing the Cabot ward (CLU = county level unit = "row in County table").
2. If you used multiple SQL queries for the last question, do it in one single query now. (In other words, find a join strategy for the tables you need.)
3. Find the number of women in occupation class 1 (managers etc.) in the Cabot ward. You may use ward code for Cabot that you found in the first query and the occupation id 1 directly - you do not need any JOINs for this query.
4. For the Stoke Bishop ward (E05002003), list the 9 occupation class names and the number of men in each occupation. Your table should have two columns called `name` and `number`. You can use the provided ward code, you do not need to join on the ward name.

We will soon be able to do more interesting statistical queries on the census data but for that we need SQL's statistical functions which we will learn next week.

Here's a slightly more tricky question to finish off with. It can be done with only the techniques that we have learnt so far.

- Find all ward names that are not unique, and print them in alphabetical order (only once each).

There are 400 distinct such names in total (for example, there are 21 wards called 'Abbey') so your query will produce quite a long table. Your query might also take a while to execute, there are faster ways to do this but not with the material we've learnt so far. The table starts "Abbey, Alexandra, All Saints" and ends "Worsley, Wyke, Yarborough".

Normal Forms

For this exercise, you may want to work in groups. There are two schemas, for both of which you have to decide which normal form(s) they are in, and how you would change the schemas to be in 3NF (BCNF if possible).

The standard way of doing this is:

1. Identify the candidate key(s) in every table.
2. From this, deduce the key and non-key attributes in every table.
3. Find the functional dependencies (FDs) in each table.
4. Determine which normal forms from (1NF, 2NF, 3NF, BCNF) the schema does or does not satisfy.
5. If the schema is not in BCNF, normalise it as far as possible by splitting tables using Heath's Theorem on the FDs that are causing the problem.

Schema 1

A school's database looks like this (it was set up by someone more used to spreadsheets):

stuId	name	gender	unit	grade
101	Fred	M	Mathematics	75
101	Fred	M	German	65
101	Fred	M	English	90
102	Sam	X	Mathematics	60
102	Sam	X	English	60
...	...	...	...	...

stuId is a student id that is unique per student. Students' names are not required to be unique, i.e. you can have two 'Fred's in the school. Gender is one of {M, F, X}. For each student and each unit they take, there is one row containing among other things the student name, unit name and the grade (0-100) that the student got on this unit. In the example above, we can see that Fred took three units (Mathematics, German and English). No two units have the same name but a unit name can appear several times in the database since many students can take the same unit. The first row of the example tells us that there is a student called Fred with id 101, who is male, and took the Mathematics unit and got a grade of 75 on it.

Schema 2

The CIA world factbook contains geographical, political and military information about the world. Here is part of one table listing principal cities from 2015:

*city	country	pop	co_pop	capital
...	...	...	...	...
Paris	France	10.8M	66.8M	yes
Lyon	France	1.6M	66.8M	no
Marseille	France	1.6M	66.8M	no
Papeete	French Polynesia	133K	285K	yes
Libreville	Gabon	707K	1.7M	yes
...	...	...	...	...

We will assume for this exercise that city names are globally unique and therefore the "City" column has been chosen as the primary key for this table. The "pop" column lists the city's population and the "co_pop" lists the population of the country in which the city is located (with abbreviations K = 1000, M=1000000). The "capital" column is a Boolean yes/no value that is set to "yes" for exactly one city in each country. (While the capital is included in the table for every country however small, non-captial cities are only included if they are of international significance.)

Intermediate SQL

This is the third of four activities to teach you about relational databases and SQL.

Videos

The videos for this activity are:

Video	Length	Slides
Intermediate SQL	22 minutes	Slides

Exercises

The exercises for this activity are all on one page:

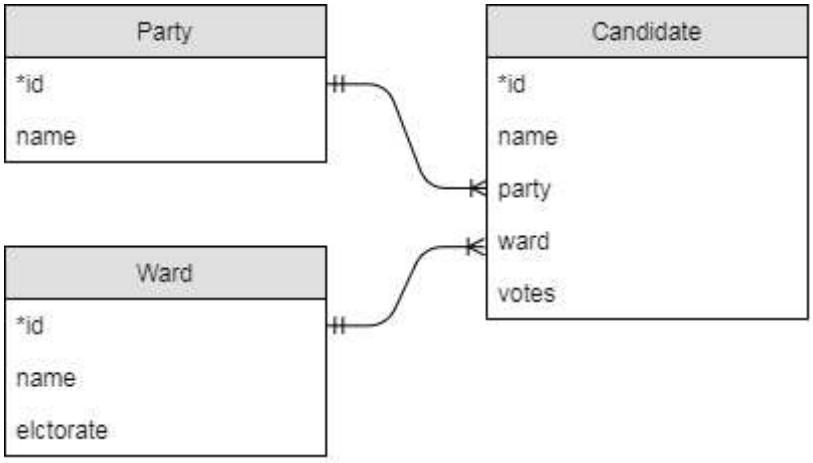
- Intermediate SQL

Census and elections exercises

Here are some more advanced exercises based on the census and elections schemas from the last activity.

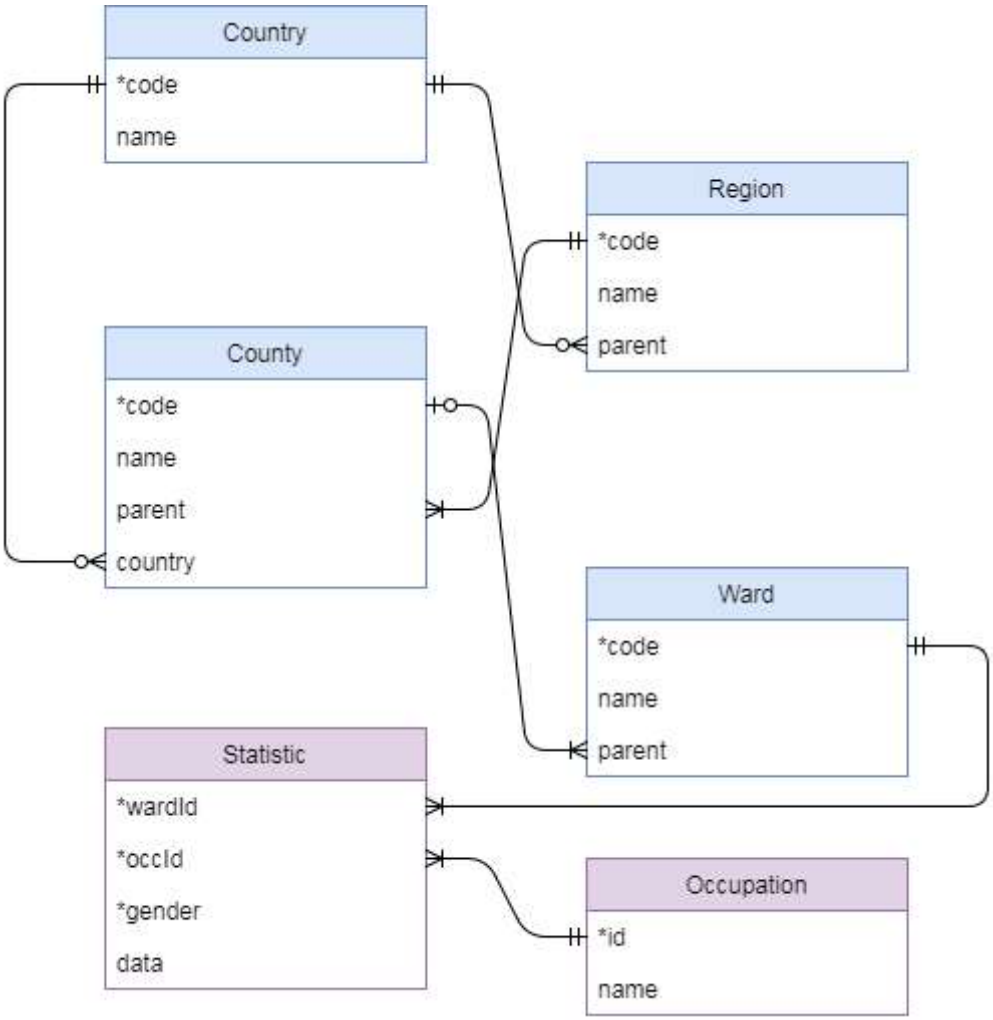
Your answer to each question should be a single SQL query, that is one semicolon at the end. JOINs, subqueries, WITH clauses etc. are allowed of course. Where an identifier is given in the question you may use it, e.g. when I say "the Cabot ward (E05001979)" you can use the ward id directly and do not need to join on the ward table to look up the name, but if I say "the Green party" then you do need to join on the party table to look up the name instead of hard-coding the party id.

Elections



1. How many votes were cast in all of Bristol in the 2014 elections?
2. How many votes were cast in the 'Windmill Hill' ward and what percentage of the electorate in this ward does this represent? Your statement should produce a table with one row and two columns called 'votes' and 'percentage'.
3. List the names, parties and *percentage* of votes obtained for all candidates in the Southville ward. Order the candidates by percentage of votes obtained descending.
4. How successful (in % of votes cast) was the Conservative party in each ward?
5. Which rank did Labour end up in the 'Whitchurch Park' ward? Your statement should produce a table with a single row and column containing the answer as a number. You can assume no ties.
6. What is the total number of votes that each party got in the elections? Your result should be a table with two columns party, votes.
7. Find all wards where the Green party beat Labour and create a table with two columns ward, difference where the difference column is the number of Green votes minus the number of Labour votes. Your table should be ordered by difference, with the highest one first.

Census



1. How many *women* work in sales and customer service occupations and live in the Cabot ward of Bristol (E05001979)?
2. How many *people* work in sales and customer service occupations and live in the Cabot ward of Bristol (E05001979)?
3. How many people work in caring, leisure and other service occupations (occupation class 6) in all of the City of Bristol CLU (E06000023)?
4. In the Cabot ward (E05001979), produce a table listing the names of the 9 occupation classes and the number of people in each of the classes in this ward.
5. Find the working population, ward name and CLU name for the smallest ward (by working population) in the 2011 census.
6. The same as the last question, but now produce a table with two rows, one for the smallest and one for the largest ward. There's no quicker way than repeating the last query twice, the question is how to stick the two "copies" together.
7. Find the average size of a ward's working population in the London (E12000007) region.
8. The same as the last question but now for every region - your query should produce a table with one row per region. The intention here is *not* to repeat the above query 9 times.
9. Produce a table that lists, for each of the 9 regions of England, the percentage of people in managerial (class 1) occupations who are women.

10. For all CLUs in the London (E12000007) region, produce a table with three columns called `CLU`, `occupation` and `count` such that:
    - `CLU` is the CLU name.
    - `count` is the number of people of the occupation class in question in the given CLU.
    - `occupation` is the name of the occupation class.
    - Only rows with `count >= 10000` appear in the table.
    - The table is sorted by `count` ascending.
11. Create a table with three columns `occupation`, `women` and `men` and one row per occupation class. The `occupation` column should list the occupation class names. The `women` and `men` columns in each row should list the total number of women resp. men in the row's occupation class in the whole dataset. The intention here is not to have to copy-paste a subquery 9 times.
12. The same as question 9, but now with a 10th row in the table listing the value for all of England. You can use the string `'England'` for the region column.

SQL and Java

In this activity you will learn how to connect to an SQL database from a Java program using the JDBC interface and the Hibernate ORM.

Videos

The videos for this activity are:

Video	Length	Slides
JDBC	25 minutes	Slides

Exercises

The exercises for this activity are:

- JDBC
- Hibernate
- SQLite

JDBC

In this activity you will learn to connect to a SQL database using Java and the JDBC classes. JDBC (Java DataBase Connectivity) is a low-level, not particularly object-oriented mechanism for accessing a database, on top of which other systems (e.g. Hibernate ORM) can be built.

JDBC Interfaces

The JDBC classes live in the `java.sql` package. Most methods on these classes can throw a `SQLException` which is a checked exception, meaning your code won't compile if you don't handle it. For simple programs, this means wrapping in a `RuntimeException` to terminate the program if something goes wrong:

```
try {
    // do stuff with JDBC
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

Two comments on this pattern:

1. In a real program e.g. web server, you of course do not want to take the server down whenever an individual method causes an error. Here you need to do something like log the error, and display an error message to that particular caller (e.g. HTTP 500 Internal Server Error) while keeping the rest of the server running. How you achieve this depends on which libraries or frameworks you are using.
2. Most JDBC work will actually take place in try-with-resources blocks, which work the same as far as exception handling is concerned but have an extra bracketed term immediately after the `try`.

Try-with-resources

A resource is something that you need to close exactly once when you are done with it, but only if it got properly opened in the first place. For example, in C heap memory is a resource: you acquire (open) it with `malloc`, and when you are done you must call `free` exactly once on the memory, except if `malloc` returned NULL in the first place (allocation failed) in which case calling `free` is an error. (You do check your `malloc` return value for NULL, don't you?) It is also an error to call `free` twice on the same memory, and it is a memory leak not to call it at all.

In Java, we don't have to manage memory by hand, but there are other kinds of resources:

- Files.
- Network connections.
- Graphics objects in some drawing/window systems.
- Database connections.

To help manage these, Java provides an interface `java.lang.AutoCloseable` with a single method `void close()` and the try-with-resources construction:

```
try (Resource r = ...) {
    // do things with r here
}
```

As long as the resource implements `AutoCloseable` (it's a syntax error to use this pattern otherwise), this pattern guarantees that

1. If the initialisation statement (in the round brackets) fails, either by returning null or throwing an exception, then the block will never be executed.
2. If the initialisation succeeds, then when the block exits, `r.close()` will be called exactly once, whether the block reached its end, exited early (e.g. return statement) or threw an exception.

A try-with-resources block can, but does not have to, include one or more `catch` statements, in which case they apply to the block, the initialisation statement and the implied `close()`.

You can also include more than one resource in the try statement by separating them with semicolons inside the bracketed term.

Advanced note

Earlier versions of java used the `finally` keyword to achieve something similar, but it was more challenging to get right especially if the close function could also throw an exception. Since Java 7, try-with-resources is the correct pattern to use and you should almost never need a `finally` block. You can implement `AutoCloseable` on your own classes to support this.

Opening a connection

You open a connection by calling

```
try(Connection c = DriverManager.getConnection(connection_string)) {
    // do stuff with connection
} catch (SQLException e) {
    // handle exception, for example by wrapping in RuntimeException
}
```

The connection string is a string containing a URL such as

```
jdbc:mariadb://localhost:3306/DATABASE?
user=USER&localSocket=/var/run/mysqld/mysqld.sock
```

When you try and open a connection, Java looks for a driver on your classpath that implements the addressing scheme (e.g. `mariadb`) that you have requested. This makes setting up the classpath a bit tricky, but we have maven to manage that for us.

However, we need to understand a bit about networking and security to make sense of that URL.

In a traditional database set-up, the database lives on its own machine (or cluster of machines) and applications connect to it over the network. To do this, by default, databases listen on TCP port 3306.

For security reasons, a competent adminstrator will set things up so that there is a firewall preventing access to the database from any machines except those of applications (and possibly developers and administrators), the database machine will certainly not be available directly from the internet. Then, applications will also need a username and password to connect to the database. Since these passwords are used by applications, and do not need to be remembered by humans, there is absolutely no excuse for choosing weak ones: the absolute minimum is something with 128 bits of entropy. Computers have no problems remembering something this long! In this case you add the extra `pass=` argument to the connection string, and to prevent passwords being sent unencrypted over the network (even if it's your internal network) you can also set up TLS or a similar tunneling technology.

Advanced note

Learning how to secure things takes time but doing things like long passwords and encryption by default is only a start. If you want to get clever you could muck about with behavioural checks so that if someone starts doing something they don't normally do it triggers logs. How are you going to spot when your database has been attacked? How are you going to tell when its being attacked but hasn't yet been broken in to? How are you going to get it back up and running before someone comes and yells at you?

Whilst the clever stuff is all fun and good an *awful* lot of this ultimately boils down to good old fashioned sysadmining. Backup everything regularly. Note what changes. Get your logs off the machine ASAP before they can be tampered with. Rotate your keys regularly because you can write a shellscript for it and whilst it make anything more secure it'll make a compliance person happy.

For a more complete guide to managing a computer, see any of Michael W Lucas's books.

On our VM, when you set up mariadb by default, the database server and client are both running on the same machine, so you can gain both security and performance by not using a network connection at all - instead when you type `mysql` it connects over a POSIX socket, another special type of file (type `s` in `ls -l`), in this case `/var/run/mysqld/mysqld.sock`. A POSIX socket is like a pair of pipes in that it allows bidirectional, concurrent communication between different processes, but with an API closer to that of a network (TCP) socket.

The point of all this discussion is that for your VM, your connection string will look like this (all on one line with no newlines):

```
jdbc:mariadb://localhost:3306/DATABASE?
user=USER&localSocket=/var/run/mysqld/mysqld.sock
```

The `localSocket` option overrides the host/port at the start. For this to work, you need the mariadb driver and a library called JNA (Java Native Access) on your classpath, and of course your system needs to support sockets.

The more standard connection string for a TCP connection would look like this:

```
jdbc:mariadb://localhost:3306/DATABASE?user=USER
```

Which really does connect to TCP port 3306 on localhost.

Advanced note

You can configure this on your VM if you wanted to: the main mariadb configuration file is `/etc/my.cnf` which in our case just contains a statement to include all files in the `/etc/my.cnf.d/` folder; in there we have `mariadb-server.cnf` which contains the lines

```
[mysqld]
skip-networking
```

Remove the last line and restart the server (`systemctl restart mariadb`) and then your mariadb server (`mysqld`) will really be listening on port 3306.

We didn't notice this with the console client `mysql` because that by default tries the socket first, and then port 3306 if the socket doesn't exist. However the JDBC driver will only try exactly the options you give, and if you don't tell it to use the socket, it will try port 3306 and throw and exception if nothing is listening there.

POM file

Under `code/jdbc/` in this unit's repository you can find a minimal JDBC application that uses the `elections` database. Download this to your VM with `wget` (or get it from the unit repository, if you have cloned it there) and extract it to an otherwise empty folder (`tar -xvf jdbc-example.tar`). It contains a file `pom.xml` and a file `src/main/java/org/example/Example.java` .

In the POM file, we note the following dependencies:

```
<dependencies>
    <dependency>
        <groupId>org.mariadb.jdbc</groupId>
        <artifactId>mariadb-java-client</artifactId>
        <version>2.7.1</version>
    </dependency>
    <dependency>
        <groupId>net.java.dev.jna</groupId>
        <artifactId>jna</artifactId>
        <version>5.6.0</version>
    </dependency>
    <dependency>
        <groupId>net.java.dev.jna</groupId>
        <artifactId>jna-platform</artifactId>
        <version>5.6.0</version>
    </dependency>
</dependencies>
```

The first one is the mariadb JDBC driver. When maven runs a program, it automatically puts the dependencies on the classpath; if you left this off then creating the `Connection` would throw an exception.

The other two are the JNA (Java Native Access) libraries for your platform, that the driver uses to connect to a POSIX socket. If you left these off, the driver would ignore the `localSocket` option, try to connect to port 3306, and throw an exception because there is nothing listening there (unless you have configured it).

- Run `mvn compile` in the folder with the POM file to download the dependencies and compile the example program.
- Run `mvn exec:java` to run the program. This uses the `exec-maven-plugin` configured later in the POM file to launch the program with the main class `org.example.Example` and the correct classpath. It should print out a list of parties.
- If you want, you can use `mvn package` to build two JAR files in `target` : one is just the compiled example class, but the more interesting one has `jar-with-dependencies` in its name and contains the compiled class and all dependencies, namely the JDBC mariadb driver and JNA (and all their dependencies). You can run this jar with `java -cp jdbc-example-0.1-jar-with-dependencies.jar org.example.Example` without

using maven if you want to. This file is built by the `maven-assembly-plugin` which we have also configured in the POM file.

SQL from Java

In the `Example.java` class, we can see an example of JDBC in action:

```
private void readData(Connection c) {
    String SQL = "SELECT id, name FROM Party";
    try (PreparedStatement s = c.prepareStatement(SQL)) {
        ResultSet r = s.executeQuery();
        while (r.next()) {
            int id = r.getInt("id");
            String name = r.getString("name");
            System.out.println("Party #" + id + " is: " + name);
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

- The SQL command goes in a string. If we wanted to add parameters for a prepared statement, we put question marks here.
- We create a `PreparedStatement` in another try-with-resources block: we need to close statements as soon as we are done with them, but it's ok for a program to keep the database connection open for as long as it runs.
- In this case, we have no parameters to pass, so we execute the query to get a `ResultSet`, a "cursor" (a kind of iterator) on the results. (A `ResultSet` is also a resource, but it closes automatically when its statement closes, so we don't have to handle this ourselves.)
- We iterate over the rows of the result and do something with them, in this case printing to standard output.
- All these operations can throw an `SQLException`, so we catch it at the end, and because this is just a small example program, we handle it by throwing an exception to terminate the whole program.

Result Sets

Java is an object-oriented language with a compile-time type system: if you declare that a `class Person` has a field `int age`, then the compiler will stop you from ever trying to put a string in there. JDBC cannot offer you this level protection because when you compile your Java program, you don't know what tables and fields exist in the database (even if you do

know, someone could change them after the program has been compiled). So you have to fall back to some more C-like patterns for using the database.

A result set can either be pointing at a row in the database or not. If it is pointing at a row, you can read values with the `get...` methods. If the result set is not pointing at a row, then trying to read anything throws an SQLException. The rules here are:

- When you get the result set back, it starts out pointing *before the first row*, so reading immediately would throw an error.
- Calling `boolean next()` tries to advance a row. If this returns true, then you have got a new row and can read from it; if you get false then you have stepped beyond the last row and it would be an error to read. (If there are no rows in your result at all, then the first call to `next()` will already return false.)

The correct pattern to use the result set is normally a while loop, as each time you land in the loop body you're guaranteed to have found a row:

```
while (r.next()) {
    // we have a row, do something with it
}
```

There are however a couple of exceptions to this pattern. First, some statements like `select count(...)` will always return exactly one row - maybe the value in the row will be zero, but that's not the same thing as no row at all - so in this case you can do

```
if (r.next()) {
    // this should always happen
} else {
    // this should never happen
    // throw an exception or something
}
```

It would still be an error to access the one row before calling `next()`, and we are not the kind of people who ignore return values from API calls.

Another special case is if you want to do something special with the first row:

```
if (r.next()) {
    // if we get here then there was at least one row
    // we're on the first row and can do something special with it
    do {
        // this block will be called exactly once for every row
        // including the first one
    } while (r.next())
} else {
    // if we get here then there were no rows at all
}
```

The `do-while` loop lets us write a block that is called exactly once for every row, while still letting us do something special with the first row without needing an ugly `boolean isFirstRow` flag or something like that.

Inside a result set, as long as we're sure we're on a row, we can read values from columns by declaring their name and type:

```
int id = r.getInt("id");
```

This tells JDBC that there is a column named `id` of type `int`, and to get its value. (You get an exception if the name or type are wrong.) Other methods include `getString`, `getDouble` etc.

For this reason, in your SQL statement, you want to be clear about the names and order of the colums you're fetching:

- If a column is something more complicated than a field, then give it an alias, e.g. `SELECT COUNT(1) AS c` then you can do `getInt("c")`.
- Never do `SELECT *` from JDBC, always list exactly the columns you need. This both fixes the order you get them in, and is more efficient if you don't need all of them.

Exercise 1

Modify the example program so it takes a party ID as a parameter, and displays only that party's name, or the string "No party with this ID." if there isn't one in the database.

To do this you will have to change the following:

- `main` reads a parameter off `args`, or prints an error message if you didn't pass an argument.
- `main` passes the parameter as an extra int argument to `readData`.
- Set the parameter as a question mark in the prepared statement, and then bind the parameter to the statement.

The command for binding a parameter is `s.setInt(pos, value)` where `pos` is the index of the parameter (question mark) in the string, *starting the count at 1 not 0*. So you simply want `s.setInt(1, value)`. Of course there are also `setString` etc. and these methods on the statement take a second parameter of the appropriate type.

The easiest way to run your command with a parameter is to build a jar with dependencies and then to call `java -cp JARFILE MAINCLASS PARAMETERS`, any further command line parameters to Java after the main class get passed as arguments to this class.

Exercise 2

A service is a piece of code that can be called by other code and typically accesses resources for them. This exercise is about writing a `DataService` that abstracts away the JDBC access for the rest of your program.

Create the following classes in their own files (you can use private fields with public constructors/getters/setters if you prefer):

```
public class Party {
    public int id;
    public String name;
}

public class Ward {
    public int id;
    public String name;
    public int electorate;
}

public class Candidate {
    public int id;
    public String name;
    public Party party;
    public Ward ward;
    public int votes;
}
```

Now, write a class `DataService` with the following description:

- DataService implements `AutoCloseable`. It has one public constructor that takes a connection string and creates a `Connection` using this string, which it stores in a private field. The `close()` method closes the connection.
- A method `public List<Party> getParties()` that returns a list of all parties in the database, by using JDBC on the provided connection.
- A method `public Party getParty(int id)` that returns the party for this id, if there is one, otherwise null.

These methods should handle all possible cases (e.g. `getWards` must still work if there are no wards in the database). but they should not throw an SQLException. Instead, make your own exception class called something like `DataServiceException` derived from `RuntimeException` (this can be an inner class of `DataService` if you want) and wrap the `SQLException` in that.

This is not an excuse for sloppy programmers to ignore the exceptions, by the way!

Now, adapt the `Example` program so that

- The main program uses the `DataService` and the domain classes (e.g. `Party`) and doesn't know about JDBC directly.
- If you pass an id as a parameter, it fetches the party with this id (if there is one) and displays party information from the `Party` instance.
- If you don't pass an id, it displays the list of all parties.

Advanced note

The `DataService` class is a resource, and it opens a connection in its constructor and closes it when you close the instance. This is a standard pattern and works perfectly if the programmer using it uses it in a try-with-resources block.

However, someone could create an instance, close it manually, and then try to continue using it, which would cause a `SQLException` on the connection. To handle this case by programming defensively, you could:

1. In the close method, set the connection field to null after closing it as a sign that this instance has been closed.
2. In all other methods, check that the connection field is not null first thing in the method and throw and exception if it is (you can write your own private method for this). According to Java conventions, the correct exception to throw in this case is a `java.lang.IllegalStateException` which means roughly *you are calling a method at the wrong time* - in this case after closing the resource in question.

Exercise 3

Implement a `Candidate getCandidate(int id)` method on the data service too. This will require you to use a JOIN in your SQL and to create instances of all three domain classes.

Hibernate

JDBC lets us connect to a database and pull out data, but we have to do a lot of manual coding to check and convert types, deal with result sets and exceptions etc. We can abstract away most of this into a service class like the `DataService` at the end of the last page, but we still have to write the data service - even though it is a lot of boilerplate code that we almost need to copy-paste from a template for each new class. We could write a script that given an ER diagram in some machine-readable format, automatically generates a data service for this class to automate all this - or we could use Hibernate to do this for us.

Hibernate is an object-relational mapping (ORM) framework, that is to say it automatically generates an advanced form of data service at runtime which includes many extra features such as sessions, transactions, caches and connection pools. It implements the Java Persistence Api (JPA), an API that in turn builds on JDBC for the purpose of implementing ORMs. Actually Hibernate has its own features that go beyond JPA, such as its own query language.

In a real application, you will be using an ORM most of the time, but you can fall back to SQL for more advanced queries that the ORM cannot support easily.

Example application - set-up

There is an example application in `code/orm` in the unit repository.

The POM file simply has an extra dependency on Hibernate:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.27.Final</version>
</dependency>
```

Hibernate's configuration lives in `src/main/resources/hibernate.cfg.xml`. Hibernate uses a *Session Factory* to create sessions, in which you can make queries:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>

    <!-- These properties set up the database connection. -->
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="connection.url">jdbc:mariadb://localhost/elections?
localSocket=/var/run/mysqld/mysqld.sock</property>
    <property name="connection.username">vagrant</property>

    <!-- Don't use this in production, use a connection pool instead. -->
    <property name="current_session_context_class">thread</property>

    <!-- Display generated SQL on the console. -->
    <property name="show_sql">true</property>

    <!-- The classes to map to database tables. -->
    <mapping class="org.example.Candidate" />
    <mapping class="org.example.Party" />
    <mapping class="org.example.Ward" />

  </session-factory>
</hibernate-configuration>
```

- The *dialect* selects the SQL dialect to speak to the database, in this case *MySQL* (there's no separate MariaDB one because the two are for all practical purposes identical).
- The connection string lives in `connection.url`, minus the username/password because there are separate properties for these. Note though that we have included the socket option here.
- Username and, if you need it, password go in separate properties.
- The connection pool is really important for performance in real applications, but we don't bother with that here and just say one connection per thread (we don't use multiple threads in our program so it doesn't matter as much).
- `show_sql` is a debuging property that prints all generated SQL to standard output. This is useful to know about when debugging your own applications.
- Finally, we list all the classes we want Hibernate to take care of. In SPE, you are going to use the Spring Framework which takes care of this automatically, but for now we're listing them all.

The classes themselves are standard Java value classes (private fields, public getter/setter) decorated with JPA annotations to explain to Hibernate how they work:

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Party {
    @Id private int id;
    private String name;

    public Party() {}

    public int getId()      { return id; }
    public String getName() { return name; }

    public void setId(int id)         { this.id = id; }
    public void setName(String name)  { this.name = name; }
}
```

- `@Entity` means this is something that maps to a table in the database. By default, Hibernate guesses the table name from the class name, but you could change this with a parameter e.g. `@Entity(name="Parties")`.
- `@Id` indicates the primary key.

The candidate class is a bit more interesting as it has foreign keys:

```
@ManyToOne
@JoinColumn(name = "party")
private Party party;
```

- `@ManyToOne` tells Hibernate that this is a foreign key for a many-to-one relationship (there is also `@ManyToMany`).
- `@JoinColumn` sets the name of the foreign key column, as the default here would be `party_id`.

Example application - querying

Let's look at the main class (Example.java). First, Hibernate uses a session factory to manage its own database connections and sessions:

```
import org.hibernate.SessionFactory;
import org.hibernate.Session;
import org.hibernate.cfg.Configuration;

public class Example implements AutoCloseable {

  SessionFactory sessionFactory;

  public Example() {
    sessionFactory = new Configuration().configure().buildSessionFactory();
  }

  public void close() {
    sessionFactory.close();
  }

  // ...
}
```

You get one from the global `Configuration` class, to which you could pass parameters to override the ones in the XML file if you wanted to (this would let you change settings in response to command line arguments, for example). A session factory is a resource, so we make our own example class a resource too which lets us run it like this:

```
public static void main(String[] args) {
    try (Example example = new Example()) {
        example.run();
    }
  System.out.println("Done.");
  System.exit(0);
}
```

The exit command at the end makes sure the program exits even if Hibernate still has a background thread running.

In the `run()` method, we use a Hibernate session, which manages a database connection:

```
try (Session session = sessionFactory.openSession()) {
    // code
}
```

We then have three examples of using Hibernate.

Loading an entity by ID

```
Party p1 = session.get(Party.class, 1);
System.out.println("    The party with id=1 is: " + p1.getName());
```

To fetch an instance by id, we just call `get` on the session with the class we want and the id. Passing the class both tells Hibernate which table to load from, and it tells the compiler what type of object to return - the way Java generics work, this lets us assign the result to a `Party` variable directly without having to do a cast.

Loading with a query

```
TypedQuery<Ward> query = session.createQuery("FROM Ward", Ward.class);
List<Ward> wards = query.getResultList();
System.out.println("  Wards:");
for (Ward ward : wards) {
    System.out.println("    " + ward.getName());
}
```

A `TypedQuery` object takes a string in HQL, Hibernate's own query language (based on SQL) and the class of the object to return - this is so that the Java compiler can figure out the return type. `getResultList` returns all results as a list.

Advanced note

The `TypedQuery` interface is part of JPA and is declared roughly as follows:

```
interface TypedQuery<T> {
  // ...
  List<T> getResultList();
}
```

This use of generics allows the compiler to be sure that the return type matches the type parameter you gave when you created the query. Of course, you don't create it directly but you ask for a query object from the Hibernate session, but behind the scenes there's an org.hibernate.query.internal.QueryImpl as well as many other Hibernate-related implementation classes.

These classes could take the type parameter as an argument to their constructor, but if you look at the sources, they don't: it seems that Hibernate does a cast internally to get the types right, which is safe as long as the Hibernate developers know what they're doing.

Queries with joins and parameters

For a more involved example, we look at the following:

```
TypedQuery<Candidate> q = session.createQuery("FROM Candidate c WHERE
c.party.name = :name", Candidate.class);
q.setParameter("name", "Labour");
List<Candidate> candidates = q.getResultList();
System.out.println("  Labour Candidates:");
for (Candidate c : candidates) {
    System.out.println("    " + c.getName() + " (" + c.getWard().getName() +
")");
}
```

HQL, like SQL, allows a WHERE clause to filter results. It also allows prepared statements, but goes one better than JDBC in that parameters can have names. You declare a parameter with a colon in the query string (`:name`) and then use `setParameter(name, value)` to bind it - this method is written so that it can take a value of any type, presumably with a combination of overloading for int/float types and `Object` for everything else.

Hibernate will automatically do the JOINs necessary to fetch the party associated with each Candidate, the SQL it runs for this query looks like this on my machine:

```
select party0_.id as id1_1_0_, party0_.name as name2_1_0_ from Party party0_
where party0_.id=?

select candidate0_.id as id1_0_, candidate0_.name as name2_0_,
candidate0_.party as party4_0_, candidate0_.votes as votes3_0_,
candidate0_.ward as ward5_0_ from Candidate candidate0_
cross join Party party1_ where candidate0_.party=party1_.id and party1_.name=?
```

Hibernate has decided to do two queries here (maybe in parallel, so the order the statements are printed on the terminal may not be accurate). The first one is because if there is no party with the supplied name, then Hibernate can stop and return an empty list; if there is then it continues with the second query to join the two tables.

The N+1 problem

Note that in the queries above, Hibernate does not fetch the ward names. It doesn't matter here because they are already in Hibernate's cache from the previous query. However, if you comment out the first two queries and leave only the third one (lines 41-50 in Example.java), something horrible happens:

```
select candidate0_.id as id1_0_, candidate0_.name as name2_0_,
candidate0_.party as party4_0_, candidate0_.votes as votes3_0_,
candidate0_.ward as ward5_0_ from Candidate candidate0_ cross join Party
party1_ where candidate0_.party=party1_.id and party1_.name=?
select party0_.id as id1_1_0_, party0_.name as name2_1_0_ from Party party0_
where party0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
select ward0_.id as id1_2_0_, ward0_.electorate as electora2_2_0_, ward0_.name
as name3_2_0_ from Ward ward0_ where ward0_.id=?
```

Hibernate is firing off one query per ward! This is called the N+1 problem, because one HQL query that returns N results ends up producing N+1 SQL queries, which is horribly inefficient expecialy for large N.

Looking at the loop structure:

```
TypedQuery<Candidate> q = session.createQuery("FROM Candidate c WHERE
c.party.name = :name", Candidate.class);
q.setParameter("name", "Labour");
List<Candidate> candidates = q.getResultList();
System.out.println("  Labour Candidates:");
for (Candidate c : candidates) {
    System.out.println("    " + c.getName() + " (" + c.getWard().getName() +
")");
}
```

From the query itself, it is not clear to Hibernate whether ward names will be needed or not, so Hibernate does not JOIN them by default which would be more efficient if you don't actually need them. In the for loop at the bottom however, you do access the names, so on every pass through the loop, Hibernate is forced to run a new query to get the name of the relevant ward.

Advanced note

How does Hibernate trigger a query off a simple `.getWard().getName()`, that you have implemented yourself in the Candidate and Ward classes?

The answer is that instead of actual Candidate objects, Hibernate creates a proxy subclass of Candidate at runtime and returns instances of this instead. These proxy objects have `getWard()` overridden to fire off another query if, and only if, they are actually called.

The solution here is to tell Hibernate at query time that you'll need the wards:

```
TypedQuery<Candidate> q = session.createQuery("FROM Candidate c JOIN FETCH
c.ward WHERE c.party.name = :name", Candidate.class);
```

This is HQL for "I'm going to use the wards, so please do a JOIN to load them too". And Hibernate now uses a single query for the Candidates again:

```
select party0_.id as id1_1_0_, party0_.name as name2_1_0_ from Party party0_
where party0_.id=?

select candidate0_.id as id1_0_0_, ward1_.id as id1_2_1_, candidate0_.name as
name2_0_0_,
candidate0_.party as party4_0_0_, candidate0_.votes as votes3_0_0_,
candidate0_.ward as ward5_0_0_, ward1_.electorate as electora2_2_1_,
ward1_.name as name3_2_1_
from Candidate candidate0_
inner join Ward ward1_ on candidate0_.ward=ward1_.id
cross join Party party2_
where candidate0_.party=party2_.id and party2_.name=?
```

There is another way to solve this problem: if every time you load a Candidate, you want the ward name to be loaded as well, then you can declare this on the JPA annotation:

```
@ManyToOne(fetch = FetchType.EAGER)
```

Exercise 1

Implement a JPA/Hibernate example application for the census database using the Country, Region, County and Ward tables (ignore Statistic/Occupation for this exercise). You could implement the following in your main program for example:

- Given a ward code, load the ward and print out all related information (ward name, county name etc.).
- Given a ward name, print out all the counties that have a ward with that name.

Pay attention to avoiding the N+1 problem in the second case.

Exercise 2

What you have learnt so far will allow you to navigate upwards in the hierarchy, e.g. given a ward you can find its associated county. This exercise is about the other way round: given a county object, you want to find all wards in it.

To do this, add the following property to your County class:

```
@OneToMany(mappedBy = "county")
private List<Ward> wards;
```

The argument to `mappedBy` must be the field name of the field in the Ward class that contains the county reference - you might have called it `parent` or something else in your class.

Then, add a getter and setter for this list property.

You have now created what is called a *bidirectional association*: a ward contains a county property, and a county contains a list of wards. You can navigate in both directions in your Java code.

Write a query that loads the City of Bristol county (E06000023) and prints a list of all its ward names with a Java for loop starting from the county object. Make sure you write your HQL query so that you don't cause an N+1 problem here.

This example also helps to explain why we don't make everything eager fetched by default: if you did that with bidirectional associations, then loading any object at all would load the entire database into memory!

SQLite

SQLite is a really good tool for "just getting stuff done", especially when you hit the limits of Microsoft Excel (or other spreadsheet software).

According to its authors, SQLite is used in some form in every single Windows 10, Mac, Android and iOS device as well as built into the Firefox, Chrome (including Edge) and Safari browsers: the browsers store their browsing history and cookies in SQLite-format databases and you can manually edit these with the command line tool if you like.

Currently in version 3.34.1, the command-line program is called `sqlite3` and exists for all major operating systems, you should be able to download it as a single file executable, put in in any folder you like and just run it. On Alpine, you can install the `sqlite` package with `apk`.

SQLite is an embedded database, which means there is no separate server process. Databases are simply files. When you run the command line tool, the tool fulfils both the client and server roles at once; when you use SQLite from a program, the SQLite driver also acts as the "server".

Census exercise

You will need the census files from databases activity 1: make a folder with the `setup.sql` file and the census csv files.

Download or install sqlite and run `sqlite3 census.db` to create a database file.

To create the tables, the command `.read setup.sql` reads and executes the commands in a file (similar to `source` on the shell). SQLite system commands start with a dot and do not take a semicolon at the end - see `.help` for a list. (Don't run the `import.sql` file, that's MariaDB-specific.)

Use `.tables` to show the tables and check that they have been created. With `.schema Ward` for example you can show the create statement for a single table.

The source data is in simple CSV files, for example if (back on the shell) you did a `head -n 5 County.csv` you would see this:

```
E09000001,"City of London",E12000007,E92000001
E09000002,"Barking and Dagenham",E12000007,E92000001
E09000003,Barnet,E12000007,E92000001
E09000004,Bexley,E12000007,E92000001
E09000005,Brent,E12000007,E92000001
```

which matches the following, from the setup script:

```
CREATE TABLE County (
    code VARCHAR(10) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    parent VARCHAR(10),
    country VARCHAR(10),

    FOREIGN KEY (parent) REFERENCES Region(code),
    FOREIGN KEY (country) REFERENCES Country(code)
);
```

To import the data, run the following on the SQLite prompt:

```
.mode csv
.header off
.import Country.csv Country
.import Region.csv Region
.import County.csv County
.import Ward.csv Ward
.import Occupation.csv Occupation
.import Statistic.csv Statistic
```

You should now have all the data imported and you could try any SQL queries from the previous exercises in this database. The SQL dialect of SQLite is not entirely the same as MariaDB: for example, SQLite can do INNER and LEFT OUTER JOIN (and CROSS JOIN) but it cannot do RIGHT OUTER or FULL OUTER JOIN.

SQLite is also extremely forgiving when it comes to data types: it treats types more or less as comments, so it will let you store a string in an integer column if you try. You can also write a CREATE TABLE statement and just leave out the types, SQLite will not mind. SQLite also *does not enforce your FOREIGN KEY constraints by default*, though you can turn this on if you want to. These are all design choices made by the SQLite developers that might or might not be the best ones for any particular use case.

Modes and file output

SQLite has different ways of displaying a table of data. Try `.mode csv`, `.header on` then `SELECT * FROM Region;` to see the regions as a CSV file with headers (mode and header settings apply until you change them or quit SQLite). If you want to actually produce a CSV file, `.once FILENAME` tells SQLite to send the result of the following query only to a file, so you can work on a query until it does what you want, then run the `.once` command and re-run the query (SQLite supports readline on most platforms so the UP key gets the last line back) to send it to a file.

The default mode, `list`, shows columns separated by bar characters. You can also try `.mode column` which is a bit more human-readable, printing in lined-up columns with

spaces in between. Or, if you want to include a table in a HTML page, `.mode html` prints the table with HTML tags. If you are on your host OS (not the alpine VM) and you have Excel installed, `.excel` opens the result of the following query in Excel.

You have now learned an extremely powerful way to analyze data that is presented to you in one or more CSV files - or any data format that you can easily export to CSV (for example most Excel files).

SQLite from Java

You can access a SQLite database from a Java program by using the JDBC driver. In Maven, declare the following dependency:

```
<dependency>
  <groupId>org.xerial</groupId>
  <artifactId>sqlite-jdbc</artifactId>
  <version>3.34.0</version>
</dependency>
```

The connection string takes the format `jdbc:sqlite:FILEPATH`. An absolute path to a file should always work (you have to spell it out and not use the `~` shortcut though), or if you are running a JAR file from the same folder as the database file then just giving the file name with no path should work too.

Exercise: write a minimal Maven/Java application that prints the region names from the sqlite database you just created.

SQLite from Python

Python is a great language for Data Science, and if you ever need to connect to a SQLite database from Python then here is how:

```
# you may have to 'pip install sqlite3'
import sqlite3
with sqlite3.connect('FILENAME') as connection:
    cursor = connection.cursor()
    for row in cursor.execute('SELECT * FROM Ward WHERE name LIKE ?',
WARDNAME):
        # do something with row
        print(row)
```

- The `with` statement is the equivalent to Java's try-with-resources.
- The `execute` statement executes a prepared statement. You put the statement itself in the first arguments, then the values to replace the placeholders as additional

arguments.
- The `execute` command returns an iterator that you can loop over, the rows themselves are python tuples (you can also do `.fetchall()` to get them in a list).

See the Python sqlite documentation for more information.

One important warning here: if you write any data to the database, you have to call `connection.commit()` afterwards otherwise the data will not actually be written to disk.

SQLite from C

The final exercise here is to write a C program, following the instructions in the slides, that reads your SQLite `census.db`, iterates over the regions table and prints "Region NAME is code CODE", where NAME/CODE are replaced with the region's name and code.

Of course you need to handle potential errors from the `sqlite3_` functions: it's only safe to continue if they return `SQLITE_OK`.

You will need to install the `sqlite-dev` package which contains the `sqlite3.h` header file, and link against the library with `-lsqlite3` in your `gcc` command.

If you are using `sqlite3_exec` with a callback function, the callback must return 0 on success and nonzero if it encountered an error, which will in turn cause `sqlite_exec` to return an error too.

For other methods, see the SQLite C API. For example `sqlite3_step` lets you iterate over the rows of a SQL result without using a callback function.