# Database Script

## *Intro*

Q: Who is working in life sciences?

- Those in ls will increasingly find data pushed out onto the web e.g. bioinformatics
- Other sciences are beginning to follow – astronomy, chemistry, even paleontology (paleodb.org)
- This session will show how to create database so that when it’s your turn to push it out onto the web it's easier for others to use
- We’ll also show you enough ways to query databases to accomplish probably around 80% of all queries you’ll ever do
- i.e. going to help build up your ‘toolbox’ of approaches

# *Chapter 1*

Let’s begin

- http://homepages.cs.ncl.ac.uk/stephen.mcgough/swc/experiments.db

```
$ sqlite3 experiments.db                        # load in db to sqlite3
> select FirstName, LastName from Person;       # our first query!
```

What are we seeing?

- Nearly all dbs in prod. use are relational dbs - sqlite is relational
- Reldbs: Every record a row in a table. A record may contain different information; each field of information is held within a column
- On table creation, decide fields (as the columns). Once created, shape is frozen. Rows flexible, columns are fixed

What does the Person table look like, in structure?

- So Person is our table, and we’re requesting the first and last names of everyone in the Person table.
- Could, e.g.

```
> select * from Person;                         # * means show every column
```

- What other fields are there?

```
> .schema Person                                   # .schema sqlite dependent
```

- 3 pieces of info associated with each person…
- So this is the structure of table Person, the information (the only info) we can store in this table for a person

Can we be more selective to what records we want from the table?

- Yes. One record from a particular person… can say

```
> select * from Person where FirstName = 'Sofia';
```

- This shows us records matching Sofia as first name, obviously

Now: how would we retrieve records that match against e.g. one of two specific logins?

- e.g. sys administrator wants to check who logged in on login ids skol or mlom

```
> .schema Person
> select * from Person;
```

- Again: devising a test to determine the correct answer before we write the code
- Manual test in this case, still a test

```
> select * from Person where Login in ('skol', 'mlom');
```

- Q: who can tell me the answer

Let's talk about 'order' of returned records

- When results returned, shouldn't depend on order of rows
- Q: any ideas why?
- A: database might reorder rows for efficiency. Table rows more like unordered sets than ordered lists
- However, can impose an explicit order

```
> select LastName from Person order by LastName;
> select LastName from Person order by LastName asc;
> select LastName from Person order by LastName desc;
```

**Exercise 1 in pairs – need to find out those people that have the last name Pavlov or Sakharov, with the results in ascending order of their login id**

```
[> select Login, FirstName, LastName from Person where LastName in ('Pavlov', 'Sakharov')
order by Login;]
```

# *Chapter 2*

All things happening occur in a well-defined order – a pipeline

- Above exercise:
    - 1. retrieving records from Person (FROM)
    - 2. filtering on the records by the fields we want (WHERE)
    - 3. filtering the results we show on the columns by the fields we want (SELECT)
    - 4. ordering the results by a particular column (ORDER)
- Come back to this later – important when constructing queries

Let's look at our database in more detail – what else is there?

```
> .tables                                        # again, implementation dependent
```

- Look at experiment

```
> .schema Experiment
```

```
> select * from Experiment;
```

- Too much data. Let's build up a query e.g. for those experiments with more than 1 hour registered

```
> select * from Experiment where Hours > 1;
> select * from Experiment where (Hours > 1) and (Hours < 10);
> select * from Experiment where (Hours < 0) or (Hours > 10);
```

- For simple things SQL is reasonably intelligent and very straightforward
- For larger things, more complicated – we'll see this later.

In general, what are we doing?

- Building up a query a part at a time to get the results we want
- Testing it, using that query
- You'll mostly be using DBs in this way

But: not good for e.g. 5tb databases. What to do?

- Select 0.1% of large db, develop against that, then test against large db
- Some people use whiteboards to develop a query that takes a long time to run
- Almost always wrong. This approach gives a fast turnaround

**Exercise 2 in pairs – devise a query to get all experiments where number of those involved > 1, order results in ascending order by experiment id**

```
[> select * from Experiment where NumInvolved > 1 order by ExperimentId asc;]
```

Have now seen enough SQL to handle around 25-30% of all queries you'll ever write!

# *Chapter 3*

All good. Let’s have a look at the bad…

> select * from Experiment;

How about getting all experiments after midnight Jan 1 1900?

- Could do

> select * from Experiment where ExperimentDate >= ‘1900-00-00’;

- Correct – in this instance! But… highlights a problem for future queries
- Q: what is it?

e.g. how would we get all experiments after Jan of ANY year?

- Far less easy – split string, manipulate, messy!
- Show to a reasonable programmer, they’d claim it was designed badly

Why is it designed badly?

- Composite field squashed into single field
- Not unreasonable for an app to want to get part of that field e.g. month

Arguable that this is the most important part of DB design

- This is how a human writes such a field – easy for us
- But not easy for computers – atomic fields easier
- Key approach: store in a db easier for a computer, display so it’s easier for humans

Going further…

- Remember Scottish law verdict – guilty, innocent, unproven
- Logic in dbs also has 3 values. e.g.

> select * from Experiment where ExperimentDate < ‘1900-00-00’;

- Field with ‘null date not included
- Should it?
- It’s a separate case
- Real data always has missing information e.g. nulls

e.g. research conducted in Canada

- Correlation between low income and antibiotic-resistant tb rates
- Used postcode as a means to identify locations with low income

Q: can you see a problem?

- What about homeless? No postcode, very likely have a low income

- Thrown away data!
- What about business travelers?
- Bad statistics = bad science

Another e.g. www.oldweather.org

- Old weather records from ships logs
- Want to process for climate science
- e.g. ask for incidence of ice in Baltic in 1900. Very logical. Ships make logs of incidents and of course weather
- But – is data good?
- Q: can anyone guess why the data had a lack of bad weather reports in logs?
- Hint: crew was busy doing something else when weather was bad
- A: Like trying to survive!
- Thrown away extremis of the data
- Always good to have a healthy skepticism when looking at data sets

Going back to our database armed with this knowledge of null data…

```
> select * from Experiment where ExperimentData is not NULL;
```

- Now starting to do real science!

What about getting total hours for an experiment? We do have…

```
> select * from Experiment where ProjectId=1214;
```

What next?

- There's a build-in aggregation function we can use for sqlite for this
- Adding to the end of our pipeline – last thing to do
- Different DBMS' have different functions

```
> select total(Hours) from Experiment where ProjectId=1214;
```

- Can also do

```
> select count(*) from Experiment where ProjectId=1214;
```

**Exercise 3 in pairs – devise a query to give a total number of hours for experiments that have more than 4 hours registered – but don't have an experiment id of 1**

[ > select total(Hours) from Experiment where Hours > 4 and ExperimentId != 1; ]

# *Chapter 4*

Let's ask a bigger question: who is involved in what project by project name?

- Start simple
- Look at Project table
- Look at Involved table

```
> .schema Project
> .schema Involved
```

- Hmm… going to need to link, or JOIN, these tables somehow to do the query
- How to do?

```
> select * from Project join Involved;
```

Lots of data! How has this happened? [bring up contents of both tables]

- Not just getting results from one table: from two
- Project table has 2 columns
- Involved has four
- Effectively done a cross-product and included all columns (6)
- And… involved has 9 rows, Project has 3 – total of 27 rows!
- Could add more tables in this manner

BUT… both tables have a ProjectId!

- Those that share the same ProjectId are the ones we want

```
> select * from Project join Involved where Project.ProjectID = Involved.ProjectID;
```

- Need to specify the tables for each reference to ProjectID to be clear which ProjectID we are referring to
- Filter on what we want to see

```
> select Project.ProjectName, Involved.Login from Project join Involved where
Project.ProjectID=Involved.ProjectID;
```

- Again, generate cross-product, filter rows, filter columns
- Now have data we wanted
- But duplicate entries…
- How to tidy?

```
> select distinct Project.ProjectName, Involved.Login from Project join Involved where
Project.ProjectID=Involved.ProjectID;
```

- Here, we are filtering out the duplicate entries in the returned table
- In this case, duplicate Project.ProjectName and Involved.Login duplicates

So:

- Brought together almost everything we've seen today
- Seen most SQL you are ever going to use

- Note – didn’t write the whole query from start – build it up

**Exercise 4 in pairs – extend the above query to also retrieve the last name of the person**

- **Hint: there’s a table we’ve already looked at that contains this information**
- **Bigger hint: you’ll need to do another ‘join’, increase the ‘select’ filtering and use an additional ‘where’**

[> select distinct Project.ProjectName, Involved.Login, Person.LastName from Project join Involved join Person where Project.ProjectID=Involved.ProjectID and Involved.Login=Person.Login; ]

# *Chapter 5*

Hmm… building up queries this size on the sqlite command line…

- A bit messy!
- Can we do this more cleanly?
- Yes!

Take our last query – cut and paste it into an editor

- Add indentation, much easier to read (for you and others)
- And save it e.g. as query.sql
- Can now, at the prompt:

```
$ sqlite3 experiments.db < query.sql
```

- Et voila – a query in a bottle!
- Very handy for composing large queries

BUT… look at what else we can do now…

- Remember shell pipes and filters?

```
$ sqlite3 experiments.db < query.sql | cut -d ‘|’ –f 2
$ sqlite3 experiments.db < query.sql | cut -d ‘|’ –f 2 | sort | uniq
```

- Have to sort first; uniq only removes duplicates next to each other
- Useful!

Seen how to retrieve data – but

- Common question – how to insert data?

```
> .schema Person
> insert into Person values(‘asak’, ‘Sakharov’, ‘Andrei’);
> select * from Person;
```

- There you are…

But what about bigger changes?

- e.g. with lots of steps in multiple tables
- e.g. adding warp drive as an experiment for Andrei
- Need to add to Experiment and Project tables
- Want to avoid inconsistency e.g. someone queries whilst you are adding data
- e.g. they might not get info that's about to be added to Project
- Would do: begin transaction; … commit;
- Makes the multiple actions a single atomic 'action' within DB
- -> Very similar in concept to version control
- Coding in a non-atomic process

Talking of bigger queries... what about optimization?

- Shouldn't we consider the best way to formulate queries to be more efficient?
- http://c2.com/cgi/wiki?RulesOfOptimization
- 3 rules of optimization:
  - 1. Don't
  - 2. Don't yet… Get it right first!
  - 3. Profile before optimizing (determine where program is spending lots of its time before optimizing it to go faster)
- If you haven't profiled, you may be optimizing something that only runs a few times and yields a negligible speed improvement
- Most of the time will sacrifice readability for efficiency – a common tradeoff
- For SQL queries - don't try to optimise joins by size of cross-product – DBMS' look forward to what you want and optimise on the fly

## *Big Exercise*

**Exercise 5 in pairs – one table that shows me:**

- **Project Name | Person Last Name | Experiment Id | Hours Worked**

Build up the query…

- i.e. let's work out what we want to see first!

> select Project.ProjectName, Person.LastName, Experiment.ExperimentID, Experiment.Hours …

Need to join the Project, Person and Experiment tables, so…

> select Project.ProjectName, Person.LastName, Experiment.ExperimentID, Experiment.Hours from Project join Person join Experiment;

Need to filter this down… experiment table has ProjectID, and so does Project. Need to associate these…

```
> select Project.ProjectName, Person.LastName, Experiment.ExperimentID,
Experiment.Hours
  from Project join Person join Experiment
  where Project.ProjectID=Experiment.ProjectID;
```

Getting there! But… in order to link person's last name and the project name to an experiment, we must have the involved table…

```
> .schema Involved
```

Note ProjectID, a person's Login and ExperimentID – the way people and projects are linked to an experiment in this database.

```
> select Project.ProjectName, Person.LastName, Experiment.ExperimentID,
Experiment.Hours
  from Project join Person join Experiment join Involved
  where Project.ProjectID=Experiment.ProjectID
       and Involved.ProjectID = Project.ProjectID
       and Person.Login = Involved.Login
       and Experiment.ExperimentID = Involved.ExperimentID;
```

Essentially, what are we doing?

- Start with a single table
- For each additional value we want to see from a different table, we add a join to that table
- For each join, there _has_ to be a corresponding WHERE clause - in order to match. Has to be!

**What we've seen will do ~75% of what you'll ever need to do**

## *In Python*

Seen what we can do with shell pipes and queries. What about more complicated stuff?

- Can do this in Python…

```
import sqlite3

connection = sqlite3.connect("experiments.db")
cursor = connection.cursor()
cursor.execute("SELECT FirstName, LastName FROM Person;")
results = cursor.fetchall()
for r in results:
   print r[0], r[1]
cursor.close()
connection.close()
```

So:

- Connection and cursor are separate
    - Connection is a file (no url)
    - Cursor is where we are currently working
- cursor.execute to run query same as on command line
- Fetch all results from query into (an array)
    - Can do fetch(10), fetch next 10, etc. exactly what happens in amazon 'next page' for example in a big multipage list
    - Results in 2d matrix - [0] is firstname, [1] is lastname
- Show all results from array
- Close stuff - put all toys away when done
- This is a template for stuff you want to do - change query, db, etc. of course. Can do real stuff online - changing connection to a url, real databases, etc.