ILLINOIS

Introduction to SQL I

Lecture 6

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

SQL

Resources:

- http://www.sqltutorial.org/ is well done and comprehensive
- has example setups for different database backends
- 'sandbox' at http://www.sqltutorial.org/seeit/ to try code example directly in the browser.

Lots of other resources out there but 'buyer beware'. Another (less well-done) tutorial with a focus on R use is at https://www.hackerearth.com/blog/machine-learning/exclusive-sql-tutorial-on-data-analysis-in-r/

SQL

- One of the most widely used programming languages
- Invented by IBM in early 1970s, standardized in 1980s + 1990s
- Frequently tied to a particular database backend
  - i.e. Oracle, Sybase, MS SQL Server are commercial ones
  - PostgreSQL, MySQL, SQLite are open source ones
- In theory SQL should be “platform neutral”
  - in practice some features tied to vendor extensions
  - we will focus on “generic” SQL that should work everywhere
  - be aware that there are “dialects” around common core
- Really important for all things ‘data’

SQL and ODBC

- Another term you sometimes hear is ODBC
- An intermediate protocol that your database may export to
- … and which an application can import from
- This provides some flexibility "in theory"
- In practice this is once again hampered by drivers and non-standard features
- We will not use ODBC here, but connect directly

SQL and SQLite

- SQLite is an *immensely* popular and very widely deployed database engine
- It (relatively speaking) "small and light" which is ...
- ... likely already in your cell phone and web browser as ...
- The C core of SQLite is available under a very liberal license
- So we will use SQLite directly and indirectly
- There are binaries for every operating system
- So you can use this on your computer
- It is available on RStudio Cloud as `sqlite3`

Basics of SQLite use: Getting started

- The "database" for SQLite is commonly a file.
- The following two lines create, and fill, a file `tutorial.sqlite`
- The two SQL files are from the tutorial site listed earlier.

```
# create the database tables
sqlite3 tutorial.sqlite < createTables.sql
# fill the database tables
sqlite3 tutorial.sqlite < fillTables.sql
```

And see how we are using redirection via < as discussed in the shell lecture ;-)

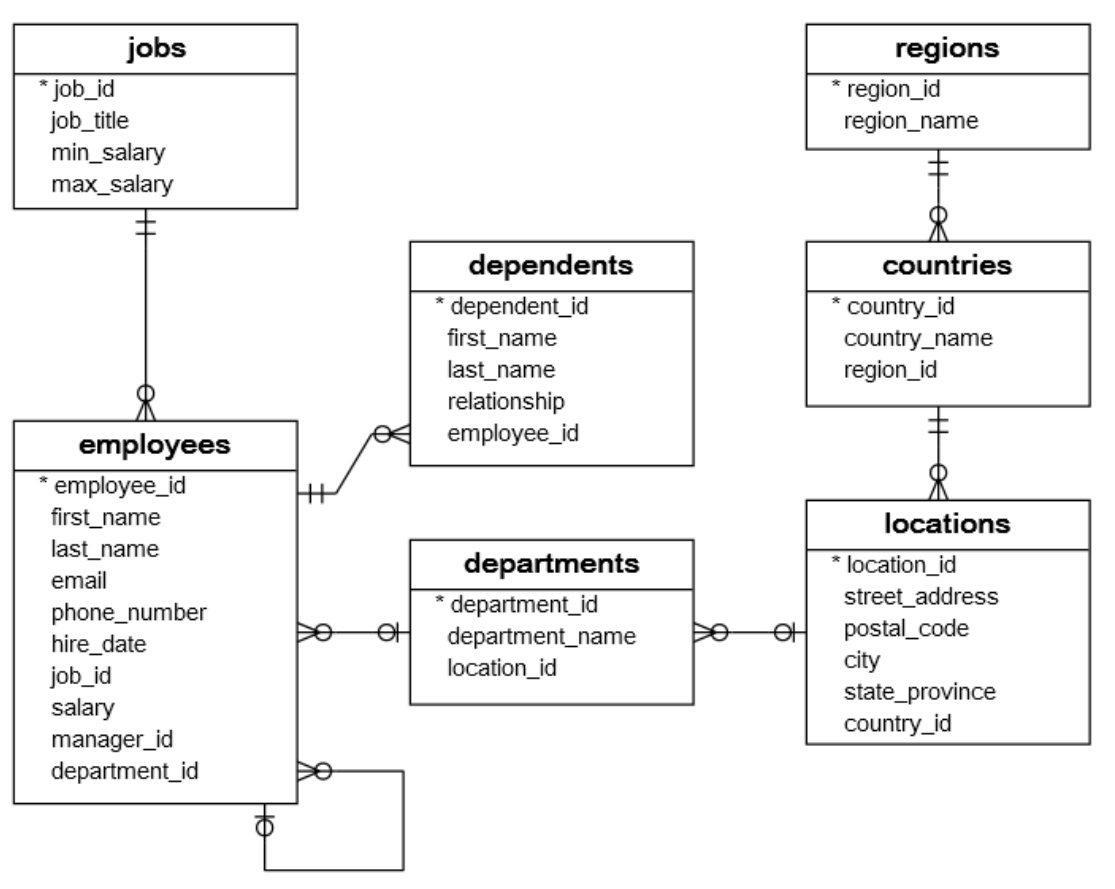
Basics

- Use the database file as the first argument:

  `sqlite3 tutorial.sqlite`

  (or any other suitable filename)
- Interactive use with some help (type `.help`)
- Commands can be given interactively, as well as in scripts
- Several options useful for scripts and automated use
- More documentation at the website

Example database



Each box describes one table.

Each box describes the columns in that table: variable name and type.

Observations are rows

Tables have relations.

Tables are indexed by the ‘starred’ column

Creation

- The `CREATE TABLE` command fully describes the table
- Below from the example we use:
  - Defines two columns of integer and text
  - Declares them to be non-empty
  - Auto-increments the index column, and uses it as key

```
CREATE TABLE regions (
    region_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
    region_name text NOT NULL
);
```

`CREATE TABLE`

- Something you need to know exists as a command
- For larger projects DBs may be set up by database admins
- For your projects you can also 'cheat' by copying R data
- But both these steps use `CREATE TABLE`
- Which in essence just defines list of columns by *column name* and *column type*
- `Integer`, `Numeric` (or `Real`), `Text`, `Date` (and `Datimetime`), `Boolean` are commonly used type
- You can request a `key` (or `index`) for performance
- You can impose `not null` to require a value

Insertion

- The `INSERT` command (and the `UPDATE` command) can insert (and alter)
- Note how the column names are listed, and mapped to values

```
INSERT INTO regions(region_id,region_name) VALUES (1,'Europe');
INSERT INTO regions(region_id,region_name) VALUES (2,'Americas');
INSERT INTO regions(region_id,region_name) VALUES (3,'Asia');
INSERT INTO regions(region_id,region_name)
       VALUES (4,'Middle East and Africa');
```

Selection

- The main workhorse of SQL is `SELECT`
- Basic form: `SELECT cols FROM table WHERE cond`
  - columns cols can be one or several colums
  - columns can be renamed, values recomputed too
  - `FROM table` is most common form
  - in its place can also be other statements
  - `WHERE cond` can impose a variety of relational statments
  - other options can follow
  - eg `LIMIT 10` shows first ten results
- Many options, maybe best to learn from example

SELECT

- The tutorial at http://www.sqltutorial.org/sql-select/ is good and makes it easy to run (and modify) simple examples
- `SELECT * FROM employees` shows all rows and columns, unconditionally
- `SELECT employee_id, first_name, last_name, hire_date FROM employees` picks four columms
- `FLOOR(DATEDIFF(CURRENT_DATE, hire_date)/365) YoS` computes a quantity and names it (in MySQL dialect)
- For SQLite: `(julianday('now') - julianday(hire_date)) / 365`

Full Example

```
SELECT employee_id, first_name, last_name,
       round((julianday('now') -
              julianday(hire_date))/365.25) as YoS
FROM employees;
```

SQLite, being small and simple, has fewer 'add-on' functions than other systems but is still pretty featureful.

Full Example in RStudio

We can also run the example in RStudio. Open File -> New File -> SQL and adjust the first line to read as follows (if `tutorial.sqlite` is in the current directory)

```
-- !preview conn=DBI::dbConnect(RSQLite::SQLite(), "tutorial.sqlite")

SELECT employee_id, first_name, last_name,
       round((julianday('now') - julianday(hire_date))/365.25) as YoS
FROM employees;
```

Then hit Shift-Ctrl-Enter, or the 'Preview' button, and the result set should appear.

Several ways to execute SQL code

- start `sqlite3 somedb.sqlite` on the command-line
  - this opens the named file
  - you can type the commands
- alternative: call `sqlite3 somedb.sqlite < somefile.sql`
- another alternative: have a shell script
- we will cover the `SELECT` command next but here are four quick ways to execute SQL code

A first shell script building on the earlier lessons:

```
#!/bin/bash

# A so-called HERE document uses the <<HERE marker, often EOF which
# stands for 'end of file' or end of input
#
# It allows us to embed instructions passed to another program, here
# sqlite3.
#
# Here we pass no other arguments to SQLite besides the actual table
sqlite3 tutorial.sqlite  <<EOF
   SELECT employee_id, first_name, last_name,
          round((julianday('now') -
                 julianday(hire_date))/365.25) as YoS
   FROM employees;
EOF
```

A second approach: run it to see how the options affect the display of results.

```
#!/bin/bash

# Here we pass two more arguments to sqlite
sqlite3 -header -column tutorial.sqlite  <<EOF
   SELECT employee_id, first_name, last_name,
          round((julianday('now') -
                 julianday(hire_date))/365.25,1) as YoS
   FROM employees;
EOF
```

A third approach: pipe command into SQLite

```
#!/bin/bash

# instead of HERE document we can also pipe code into sqlite3
echo "
   SELECT employee_id, first_name, last_name,
          round((julianday('now') -
                 julianday(hire_date))/365.25,1) as YoS
   FROM employees;
" | sqlite3 -header -column tutorial.sqlite
```

A fourth approach: pipe command into SQLite

```
#!/bin/bash

# we can also pipe sql code from a file directly into sqlite
# this allows for the file to change, or disappear, or ...
cat basicExample3.sql | sqlite3 -header -column tutorial.sqlite
```

RStudio convention

- The earlier example showed how to make it work from RStudio
- The file contained the line below
- It is a comment to SQL and SQLite (as started by `--`)
- The remainder tells RStudio to use a particular R package (more on that later) via the `!preview` argument
- So as argument to `conn` to specify a connect ...
- ... we call `DBI::dbConnect()` with two arguments to
  - make it SQLite connection as provided by the `RSQLite` package
  - using the file `tutorial.sqlite` in the current directory

```
-- !preview conn=DBI::dbConnect(RSQLite::SQLite(),"tutorial.sqlite")
```

Sorting: `ORDER BY`

```
SELECT employee_id, first_name, last_name,
       hire_date, salary
FROM employees
ORDER BY first_name, last_name DESC;
```

DISTINCT: Remove duplicates

```
SELECT DISTINCT job_id, salary
FROM employees
ORDER BY job_id, salary DESC;
```

Qualifying selection with WHERE

Imposing a greater-than:

```
SELECT employee_id, first_name, last_name, salary
FROM employees
WHERE salary > 14000
ORDER BY salary DESC;
```

Qualifying selection with **WHERE**

Also works with text and wildcards: names starting with H

```
SELECT employee_id, first_name, last_name
FROM employees
WHERE last_name like 'H%'
ORDER BY salary DESC;
```

SELECT

- Look at the next few sections in th SQL Tutorial
- See other SELECT qualifier and examples
- Try them!

Summary

- SQL is a widely-used important data programming language
- Designed for relational structured data that is stored in tables
- While SQL is standardized different dialects exists
- We work with SQLite, a small and portable SQL engine
- We saw how to `CREATE` table and how to `INSERT` data.
- We used `SELECT` for a number of queries along with
  - several optional qualifiers
  - `WHERE` queries