How to Use SQLite with R

Seth Falcon

January 18, 2006

1 Introduction

This vignette introduces SQLite, a self-contained relational database engine. The excercises provide a mini-tutorial on relational databases and use of structured query language (SQL).

We begin by reviewing relational database system (RDBMS) concepts, introducing structured query language (SQL), and describing features of SQLite. From there, we will work through an extended example in which we will create a SQLite database containing annotation data for the Affymetrix hgu95av2 microarray chip. The example will demonstrate database creation and querying, basic SQL, and techniques for importing large amounts of data into SQLite.

2 An overview of Relational Database Mangement Systems

An RDBMS is a system for managing tabular data and relations among collections of tabular data. Each table has a set of fields (columns) that define the type of information stored in the table. Each row of a table represents a record in the database. Conceptually, a table in an RDBMS has much in common with a *data.frame* in R.

Each row in a table must be uniquely identifiable. Usually, one column is designated to store a value gauranteed to be unique for each record in the table and this column is referred to as the *primary key* for the table.

Relations in a database describe the interdependencies between the pieces of data stored in the system. There are three types of relations that are worth naming: many-to-many, one-to-many, and one-to-one.

Two columns A and B in a database have a many-to-many relationship when a given record in A is related to zero or more records in B and a record in B is related to zero or more records in A. The relationship between Affymetrix ids and PubMed ids, as encoded in the `hgu95av2PMID` environment, is a many-to-many relationship: each Affymetrix id can relate to zero or more PubMed ids and any given PubMed id can be related to zero or more Affymetrix ids.

A one-to-many relationship occurs when a given record in column A is related to zero or more records in column B and a given record in column B is related to exactly one record in A. For example, the relationship between journals and papers is one-to-many; a journal has many papers, but a paper is published (generally) in a single journal.

Columns A and B have a one-to-one relationship if a given record in A corresponds to exactly one record in B and vice versa. There is a one-to-one relationship between GO ids and terms in the GO ontology.

3 Structured Query Language (SQL)

Structured Query Language (SQL) is a language used for manipulating relational databases. It is defined by an ANSI standard that, in theory, is implemented by all RDBMS's. In practice, it is common for the SQL implemented in a given RDBMS to include extensions that provide enhanced functionality at the cost of lost compatibility across different RDBMS's.

A good resource for learning SQL is: `http://sqlzoo.net/`.

4 A lightweight database: SQLite

SQLite (`http://www.sqlite.org/`) is a lightweight, self-contained, zero-configuration, cross-platform, open source database engine. SQLite implements most of the ANSI SQL92 standard, but is missing some features found in an enterprise-class RDBMS's such as stored procedures. Unlike most enterprise-class RDBMS's, SQLite is not a client/server application and for the most part, it is assumed that a single process (user) will access a database at a given time.

Each SQLite database is contained in a single file (portable across platforms) and can be manipulated using the `sqlite` command line tool.

Binding are available for many programming languages that allow for programatic manipulation of SQLite databases. In *How to use DBI: Connecting to Databases with R*, we demonstrate the use of *RSQLite*, a package that allows R to access SQLite databases.

In this vignette, we focus on the use of the `sqlite` command line tool. In order to work the excercises below, you will need a version of SQLite greater than or equal to version 3.1.3.

5 An annotation database for Affymetrix hgu95av2

Now we will build a database of annotation data for the Affymetrix hgu95av2 chip.

5.1 Flat files

We will use the following plain-text "flat files" as the input data for our annotation database:

hgu95av2-acc.txt Two columns. The first column gives Affymetrix probe id, the second is accession number.

hgu95av2-go.txt Three columns: Affymetrix probe id, GO id, evidence code. Each row describes a GO annotation for a probe and gives the evidence used to determine the annotation.

hgu95av2-goId2Ontol.txt Two columns: GO id, GO ontology code (BP, CC, MF).

hgu95av2-pmid.txt Two columns: Affymetrix probe id, PubMed id.

5.2 Starting SQLite

The sqlite command line tool takes the name of a SQLite database file as its first argument. If you want to create a new database, simply specify the name and the file will be created for you. We'll use `hgu95av2-temp.db` as the name of our annotation database for the Affymetrix hgu95av2 chip.

Here is how to start the session:

```
sqlite3 hgu95av2-temp.db
sqlite> -- this is a comment
sqlite> .help  -- this will show some useful commands
```

5.3 Creating a new table: CREATE

Tables in a database are created using SQL's CREATE command. Create a table named `go_ont_name` with two columns: `ont` and `ont_name`. It is a good habit to use comments to document the structure of the tables you create.

```
sqlite> CREATE TABLE go_ont_name (ont TEXT,      -- Ontology code
                                  ont_name TEXT  -- Ontology name
        );
sqlite> .tables
go_ont_name
```

5.4 Inserting values by hand: INSERT

Since GO consists of just three top-level ontologies, we can enter the data for this table by hand using SQL's INSERT command.

```
INSERT INTO go_ont_name (ont, ont_name) VALUES ('CC', 'Cell Cycle');
```

Exercise 1
Use INSERT statements to add rows for Biological Process (BP) and Molecular Function (MF).

5.5 Querying the database: SELECT

Now that we've created a table and inserted data into the table, we can extract the data using SELECT.

Exercise 2
Try the following SELECT statements on the test database.

```
SELECT * FROM go_ont_name;
SELECT ont_name FROM go_ont_name WHERE ont = 'BP';
SELECT ont_name FROM go_ont_name WHERE ont = 'foo';
SELECT count(*) FROM go_ont_name;
SELECT * FROM go_ont_name WHERE ont_name LIKE '%le%';
```

5.6 Updating values by hand: UPDATE

```
UPDATE go_ont_name SET ont = 'MMM Cookies' WHERE ont = 'MF';
SELECT * FROM go_ont_name;
```

5.7 Deleting rows: DELETE

```
DELETE FROM go_ont_name WHERE ont = 'MMM Cookies';
SELECT * FROM go_ont_name;
```

5.8 Deleting tables: DROP TABLE

```
DROP TABLE go_ont_name;
.tables
```

5.9 Importing data

Creating database tables by hand using INSERT statements lacks efficiency. Most RDBMS's come with a utility that allows you to quickly import delimited text files. SQLite provides `.import` for this purpose.

SQLite's `.import` command imports data into a single database table. The table must exist in the database prior to calling `.import`.

Here we will use a text file containing a number of table creation SQL commands and a number of `.import` commands to load data into the empty tables.

```
sqlite3 hgu95av2-example.db < affy-annotation-schema.sql
sqlite3 hgu95av2-example.db
sqlite> .tables
acc          go_ont       go_ont_name  go_probe     pubmed
```

5.10 Combining data from more than one table

In database lingo, a *join* is a query that combines data from more than one table.

5.10.1 Inner joins

The `go_ont` table maps GO ids to GO ontology. We can pull in data from the `go_ont_name` table to get the more descriptive listing of the ontology for each GO identifier. This required an INNER JOIN, which can be written as follows:

```
SELECT go_ont.go_id, go_ont.ont, go_ont_name.ont_name
  FROM go_ont, go_ont_name
    WHERE (go_ont.ont = go_ont_name.ont) LIMIT 3;
go_id|ont|ont_name
GO:0005764|CC|Cell Cycle
GO:0006029|BP|Biological Process
GO:0008152|BP|Biological Process
```

Exercise 3
Create a table mapping Affy ids to GO ontology codes.

5.10.2 Self joins

The following compound statement selects all Affy probes annotated at GO ID "GO:0005737" with evidence IDA *and* ISS. This uses a *self join* and demonstrates a common abbreviation syntax for table names.

```
SELECT g1.*, g2.evi FROM go_probe g1, go_probe g2 WHERE
  (g1.go_id = 'GO:0005737' and g2.go_id = 'GO:0005737')
  AND (g1.affy_id = g2.affy_id)
  AND (g1.evi = 'IDA' and g2.evi = 'ISS');
affy_id|go_id|evi|evi
D28235_s_at|GO:0005737|IDA|ISS
L15326_s_at|GO:0005737|IDA|ISS
U04636_rna1_at|GO:0005737|IDA|ISS
```