An SQL Beginner Tutorial

Simon Symeonidis

Fri Dec 6 00:03:29 EST 2013

This document aims to give you beginner level knowledge of the following points in databases:

- Learn about relationships
- Learn about views
- Learn about polymorphic relationships
- work with SQLite, MySQL, and PostgreSQL (their differences, and use cases)
- slight talk about other data storage facilities such as Redis, and MongoDb

Approach

We will be going over 2 SQL implementations. One of the implementation is SQLite3. The next implementation we will focus on is PostgreSQL. SQLite3 will be used in the beginning due to its easy configuration with the system. We will be able to disregard other issues, such as database user management, and get at the coding as fast as possible. As stated this is to learn the language, and SQLite3 should not normally be considered as a good database for a system that requires higher concurrency and responsiveness (though please note that it still has valid use cases on other applications)

At the end of the document, we'll briefly talk about other database-like storage alternatives that might be used as well. We will make a distinction however, and emphasize that for the right job, we must choose the right tool.

Minor Forenote

A database contains tables. Tables contain records, often called rows. A database usually contains data of a specific application. Tables represent entities in the domain logic of that application, that require persistence.

For example, a `Person` class could require to have its state persisted. A `Person` class, may aggregate (have many of / have a list of) `Book` classes. On the programming level, these relationships are possible using memory allocation and references to objects. When persisting such relationships, we need to identify the type of relationship (has-one, has-many), and represent them using unique keys, and proper schemas (table structures).

SQLite3

First you'll need to install the database system. Please follow the installation instructions here:

http://www.sqlite.org/download.html

Specifically, download the precompiled binaries for windows (sqlite-shell-win32).

If you're on linux, use the package manager specific to your distribution to install sqlite3.

Both of these should provide you with a shell that will allow you to manipulate sqlite3 databases.

Using the shell

When you run the sqlite3 shell, you'll see the following:

```
[psyomn@aeolus ~ 0]$ sqlite3
SQLite version 3.8.1 2013-10-17 12:57:35
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Creating a table

Consider that we want to persist information in a database of people. The information we want to store for each person is their name, surname and their age. If we had a notebook, and we wanted to store this information, we would create a table by separating the page in three parts: name, surname and age. This is what we want to do as well with the table in this databse.

```
sqlite3> create table person (
    ...>   name varchar(50),
    ...>   surname varchar(50),
    ...>   age int);
```

Inserting into a table

Run the following provided you have created the required table.

```
sqlite3> insert into person (name,surname,age) values ('jon', 'doe', 18);
```

If you want to enter a list of people in a single command, you can do the following:

```
sqlite3> insert into person (name,surname,age) values
    ...> ('jon', 'doe', 18),
    ...> ('jon' 'snow', 32),
    ...> ('marry', 'spanakopitakis', 21);
```

With the previous code sample, we now have a database that contains the current table:

Name	Surname	Age
jon	doe	18
jon	snow	32
marry	spanakopitakis	21

Selecting from the Table

Naturally, after inserting information to the table, we want to be able to retrieve it. The keyword to memorize for this aspect is *Select*. We need to provide it what fields we are interested from the table, and finally the table name as well. For an example let us select all the surnames and ages from the table *person*.

```
sqlite3> select surnamge, age from person;
```

But what if we wanted to impose a constraint? For example, let us require that we select the fields *name* and *age* for the person named 'marry'.

```
sqlite3> select name, age from person where name='marry';
```

That is good and all, but problem arises when we have more people being recorded in the system called 'marry'. Then we won't be able to distinguish who is who. This is where keys come into effect, and how specific find queries are usually implemented in *SQL*.

Destroying a Table

Since we want the ability to distinguish between data pairs where we can not rely on the uniqueness of each field, we can assign a key. We call this key an `id`. However we already defined the table person. Let's get rid of it, so we can create a new table with the id field.

```
sqlite3> drop table person;
```

Now let's recreate the table with the wanted id.

```
sqlite3> create table person (
    ...>   id integer primary key autoincrement,
    ...>   name varchar(50),
    ...>   age  int);
```

We set the key as an integer, and denote that it is a primary key. We also express the wish of making it automatically increment by adding `autoincrement` in the end. Each time a record is added to this table, an id will automatically be assigned to it, by increasing the previous maximum key by one. So for example, if we were to repeat the entries of *jon doe, jon snow, and marry spanakopitakis*, jon doe would have a key with value '1', jon snow '2', and marry '3'. We can now reference rows properly and form relations using them.

Deleting a row from a table

If we wish to remove an entry from the database table, we can do so by using `delete`. In order to delete something in particular we need to tell *sqlite3* specifically what row. So here we add a WHERE clause. Let's take the case that we want to remove the row with id '2'. We would need to do the following:

```
delete from person where id = 2;
```

Be cautious when writing this query. If you acccidentally forget to add the where clause, and write:

```
delete from person;
```

All the rows will be deleted!

Updating a row in a table

Once we store the information in a row, there might be a situation where somewhere in the future, we wish to update the information. For example, let us consider the following table:

id	Name	Surname	Age
1 2	Jon Frank	Johnson Frankson	23 25

Jon came in a few weeks ago, and they entered his information. But the person entering his information did a mistake and instead of typing in '32', they typed in '23'. Jon comes in, and wants to have his proper age on the system.

The query we need to come up with consists of at least the keyword 'update'. However we need to pinpoint where the update will happen exactly. Since we want to update Jon, we need to add a `where` clause on `id = 1`. We need to specify also the columns that we want to modify. We use the keyword `set` for this.

Here is the required query:

```
update person set age=32 where id = 1;
```

Frank Frankson changed his name to Charlie Thawson. He came in in order to update this information. We have to update two columns simultaneously now. We just need to use `set` once, and then separate the columns by comma. And then we're done.

```
update person set name='charlie', surname='thawson' where id = 2;
```

Altering a table