Week 8, continued

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. SQL

- Pretty much any website you visit has some sort of database to store information.
- On Problem Set 7, C$50 Finance, you'll use a database to store stock and user information.
- If you have a database of users, they might have a username, id, password, email address, and many more fields.
- MySQL, and SQL more generally, is a **relational database** - that is, it stores data in rows and columns.
- SQL has four main operations - `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- We'll use the tool phpMyAdmin (coincidentally written in PHP), which provides a graphical user interface (GUI) to MySQL's functionality.
- Like C (and unlike PHP), MySQL requires us to specify types (such as `CHAR`, `VARCHAR`, `INT`, `BIGINT`, `DECIMAL`, `DATETIME`, and many others)
- Let's look at an example in phpMyAdmin. We start with an empty database called `lecture`, which doesn't contain any tables yet.
  # We can then add a table called `users`, with two columns - just a username and a name.
  # We then have to actually configure these columns - we'll name one `username` and one `name`, and we have to designate what datatypes these columns will be.

    # There are a huge variety of possible types in MySQL! Some are broad and basic (like `INT`), while others are very domain-specific (like `GEOMETRY`, which lets you store geographical data in a column of your database).

    # We'll keep it simple here and look at just the string types… of which there are still several: `CHAR`, `VARCHAR`, `TEXT`, `TINYTEXT`, `MEDIUMTEXT`, and `LONGTEXT`. `CHAR` is not a `char` in the C sense, but rather a fixed number of characters.

Let's set `username` as a `CHAR` column of length 16, so users can pick usernames up to 16 characters in length (shorter usernames will be padded with spaces or nul bytes for storage).

What about `name`? We don't know how long people's names are, so we have to make a judgement call about how long a field to store them in. A common choice is 255, because that used to be the actual maximum that MySQL could provide (and even though nobody's name is probably going to be 255 characters long, making the column that long sort of saves us having to even think about how long people's names really are).

    # This reserves 255 bytes for everyone's name! Since most names are much shorter than that, we'd be wasting a lot of memory this way.

    # Fortunately, MySQL provides another type, `VARCHAR`, that can vary in length. We specify a maximum length for the field, but the database will intelligently use fewer bytes than that if the user enters a shorter name.

    # So why don't we use `VARCHAR` for everything? Dynamically allocating memory can slow things down a bit, but more importantly, if we don't know exactly how long each field is, we lose random access (meaning that we can't index directly to the location in memory where a particular value is, because we have to scan through the memory to figure out where each field ends).

We also have to specify whether or not columns are allowed to be `NULL`, i.e., whether that value is required in every row or whether it's optional. We want every user to have a username (obviously) and provide their name, so neither of our columns can be `NULL`.

Another option we have in MySQL, unlike in a spreadsheet in Excel or the like, is to apply an **index** to one or more columns. An index is a way of telling the database in advance how you're going to query your data - so if you know you're going to be

looking users up by their usernames often, you can tell the database to index the `username` column, meaning that it will store the information in a way that makes lookups much more efficient (often a tree of some kind, like the binary search trees we discussed earlier in the course).

    # There are some other conditions we can ask the database to enforce, like `UNIQUE`, which in addition to indexing the data for lookups, also requires that all entries in that column be distinct. We want `username` to be `UNIQUE`, and we don't really care about whether `name` is indexed in any way.

- We can insert data into our database with code. We've written a PHP function for you called `query`, which lets you run MySQL queries, so to insert David as a user into our `users` table, we can do the following:

```
<?php

    query("INSERT INTO users (username, name) VALUES('malan', 'David
 Malan')");

?>
```

    # `'malan'` and `'David Malan'` are strings, which we might usually enclose in double quotes, but here we use single quotes to avoid confusion with the outermost quotes enclosing the entire SQL statement (we could also use double quotes and escape them with backslashes, but that's significantly uglier).

    # We can also type this straight into phpMyAdmin, which has a SQL tab that lets you directly make SQL queries. When we execute the above query by typing `INSERT INTO users (username, name) VALUES('malan', 'David Malan')` in this tab, we can go back to our table and see that we now have a row in our database.

- Because SQL lets us interact with our database programmatically, we can do something like this:

```
<?php

    query("INSERT INTO users (username, name) VALUES(?, ?)",
 $_GET["username"], $_GET["name"]);

?>
```

    # Recall that in PHP, variables beginning with `$_` are **superglobal** variables, and `$_GET` contains the parameters passed with the `GET` request (i.e., in the URL).

    # We've replaced the hardcoded values for `username` and `name` with question marks, which let us insert variable values into our query string. We then pass our `query` function additional arguments telling it what variables to fill in for the question marks (just as we did with format strings in `printf`).

        # Note that we don't have to worry about what kind of quotation marks we're using here, since PHP will handle that for us in filling in the values.

- Suppose a user now wants to log in, and we want to check that they're giving us a real username for an existing user in our database. We can do this like so:

```
<?php

    query("SELECT * FROM users WHERE username = ?", $_POST["username"]);

?>
```

    # If one row is returned, we know the user exists; if zero rows are returned, that user has never been to our site before. We can't have more than one row returned, because we specified `UNIQUE` for the `username` column!

    # We can type this query into phpMyAdmin's SQL tab, and we get back one row.

- We can also `SELECT` just certain columns, for example:

```
SELECT name FROM users WHERE username = 'malan'
```

    # So now we get one row back, but instead of getting the entire row, we get just the one column `name`.

- Rather than just storing a username and a name for each user, we should have an `id` field, and make it a numeric type (we'll just use `INT`, since we don't care about what size it'll be).

    # We're given various attributes we can apply to our new `id` column, including `BINARY` and `UNSIGNED` - `UNSIGNED` makes sense for an identifier, so we'll use that for our `id` column.

    # Now, under Index, we'll choose `PRIMARY`, which makes `id` our **primary key**. A primary key must be unique and not null, and it will be used to find rows from the database.
    # Because an `INT` is only 32 bits, or 4 bytes, while our usernames are 16 bytes, it's much faster to select rows by `id` than by `username`, even though `username` is also guaranteed to uniquely identify a single user.

- What if we add three more columns to indicate a user's city, state, and ZIP code? We can make `city` a `VARCHAR` of length 255 and `state` a `CHAR` of length 2, but how should we store `zip`?
    # We could save `zip` as an `INT`, since it's a number, but we need to be able to handle leading zeroes like in Cambridge's ZIP code, 02138, so instead we'll use a `CHAR` of length 5.
    # But this is silly - why are we storing for every user `city`, `state`, and `zip`, when a ZIP code uniquely identifies a city and state? Instead, we can just store `zip` in our `users` table, and have a separate table to associate ZIP codes to their corresponding cities and states.
        # Best design indicates that we should factor out common data, just like we factor out common code.
        # But now we have to combine data from two different tables. Fortunately, SQL provides a way of joining different tables based on shared values, as below:

```
SELECT * FROM users JOIN zips ON users.zip = zips.zip
```

        # This means "give me back the combination of the users table and the zips table by putting together their rows where the zip field is the same."

2. Transactions

- Now let's say we have multiple users interacting with our database at the same time. How do we make sure that their queries don't mess each other up?
    # Think about the example of your refrigerator in your dorm room - imagine you open the fridge, see there's no milk, and go to CVS to buy some. Then your roommate comes home, opens the fridge, sees there's no milk, and goes to a different CVS to buy some. You both bring milk home, and now you have twice as much milk as

necessary. Your roommate made a decision on the basis of the state of a variable (the presence or absence of milk) that was in the process of being changed by you.

To solve this, you could put a lock on the refrigerator while you're out getting milk.

With real databases, particularly in applications with sensitive financial information, the consequences might not just be that you have too much milk - someone could withdraw more than the amount of money in their bank account by simultaneously withdrawing from two different ATMs, so that when each one checks whether there's enough money in the account to make the withdrawal, they both return the answer yes and give the money.

We've provided, in the spec for Problem Set 7, some code to help you handle this problem.

3. SQL Injection

- Another thing to be careful of (although we'll help you out with this one too) is a **SQL injection attack**.

  # Imagine a login system implemented in PHP like the following:

  ```
  $username = $_POST["username"];
  $password = $_POST["password"];
  query("SELECT * FROM users WHERE username='{$username}' AND
   password='{$password}'");
  ```

  # Now if someone types in the password field `12345' or '1' = '1`, that query becomes:

  ```
  query("SELECT * FROM users WHERE username='username' AND
   password='12345' OR '1' = '1'");
  ```

  # Because `'1' = '1'` is always true, this will always log the person in.

  # This is why you should always use the question mark syntax, rather than interpolating variables directly, because when PHP fills in the variable values into the query for you, it escapes any potentially dangerous characters - so that query instead looks like this:

```
query("SELECT * FROM users WHERE username='username' AND
 password='12345\' OR \'1\' = \'1'");
```

- There have been various jokes about this, including this XKCD[1].

4. Final Projects

- We've previously mentioned Hue lightbulbs, which can be controlled via an **API**, or application programming interface, using HTTP requests (rather than `GET` or `POST`, you can use `PUT` to change the state of the lightbulb).
- We also have some Microsoft Bands, somewhere in the smartwatch realm.
- You can write iOS code as well.
- We have some Arduinos, which can be connected to various sensors.
- We have some Leap Motion devices, that respond to hand motions without touch.
- We also have Myo armbands, which can read the muscle movements in your arm to control various computational actions.
- We have Google Cardboard for 3D environments, and Samsung Gear (the more expensive, non-cardboard version of the same).
- If you'd like to use any of these tools in your final project, let us know!

[1] https://xkcd.com/327/