Relational Database Management Systems

CSC 337, Fall 2013
The University of Arizona
William H. Mitchell
whm@cs

Introduction

CRUD

At the heart of many web applications is CRUD:

Creating records

Reading records

Updating records

Deleting records

Which of the the PHP problems on assignments have been CRUD apps?

Relational Database Management Systems

CRUD operations can be done using a small set of operating system services (primarily read/write/seek).

Over the years, many types of libraries and systems have been developed to make CRUD operations easy.

In 1970, Dr. E. F. Codd at IBM wrote a seminal paper,
A Relational Model of Data for Large Shared Data Banks

The paper introduced the idea of *relational database*.

Web apps often use a relational database to store data.

Classes like CSC 460 provide a thorough treatment of relational databases. In this class we'll only learn a few basics.

RDBMSs, continued

Here's one way to describe the fundamental characteristic of a relational database:

All data is represented as tables.

Some facts about tables in a relational database:

Tables have zero or more rows of data.

Rows have one or more columns of data.

All rows in a table have the same set of columns.

Columns exist even if there are no rows.

Conceptually, all data is just text. There are no "pointers"

Examples of tabular data

Here is a table with information on plants:

id	plantname	price	maxheight	exposure
1	Glossy Abelia	6	6	full sun
2	Grand Fir	25	25	full sun
3	Spanish Fir	25	15	full sun
4	Blue Spanish Fir	25	15	full sun

This table has information about the owner of a residence:

id	ownername	city	state
1	James Grishom	Tucson	AZ
2	Walter Vickers	San Diego	CA
3	Callow Bruce	Casa Grande	AZ
4	Wright Frank	Casa Grande	AZ

Does this data meet the requirements of the previous slide?

(Thanks to Dr. Snodgrass for this data from his CSC 460 examples!)

Tabular data, continued

How could we represent the blog's data with tables?

- What could correspond to rows?
- What would the columns in each be?
- What might some sample rows be?
- How many tables?

MySQL

MySQL

MySQL is a relational database system that's installed with XAMPP.

SQL is an acronym for Structured Query Language but it's not just for querying!

We can use SQL to:

- Create databases
- Create tables in databases
- Insert rows into tables
- Read rows
- Update data in rows
- Delete rows
- Lots more...

The `mysql` command

The `mysql` command provides a line-oriented interface that can be used to enter SQL commands.

```
Z:\whm\337>c:\xampp\mysql\bin\mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.

mysql> select now();
+---------------------+
| now()               |
+---------------------+
| 2013-11-05 20:10:40 |
+---------------------+
1 row in set (0.00 sec)

mysql> select datediff('2013/12/13', now()) "Days to final";
+---------------+
| Days to final |
+---------------+
|            38 |
+---------------+
1 row in set (0.00 sec)

mysql> quit
Bye
```

Making a database to experiment with

Let's make a database to work with using XAMPP's MySQL.

http://cs.arizona.edu/classes/cs337/fall13/files/plants.sql is a file with data for three tables. Copy it into the current directory.

```
Z:\whm\337\sql> c:\xampp\mysql\bin\mysql -u root
mysql> create database plants;
Query OK, 1 row affected (0.00 sec)

mysql> use plants;
Database changed

mysql> source plants.sql;
Query OK, 0 rows affected [...lots more...]

mysql> show tables;
+-------------------+
| Tables_in_plants |
+-------------------+
| installation      |
| plant             |
| residence         |
+-------------------+
3 rows in set (0.00 sec)
```

Using the plants database with XAMPP

The previous slide showed how to make a database named plants and populate it from a file. That only needs to be done once. (To repeat it, do "drop database plants;" first.)

Once created, start mysql like this, naming `plants` on the command line:

```
Z:>c:\xampp\mysql\bin\mysql -u root plants
mysql> show tables;
+-------------------+
| Tables_in_plants |
+-------------------+
| installation      |
| plant             |
| residence         |
+-------------------+
3 rows in set (0.00 sec)
```

`mysql` on lectura

Here's how you can access the same tables on lectura:

```
% mysql -h mysql -p -u cs337f13 whm_cs337f13
Enter password:  (tednelson)
mysql> show tables;
+-------------------------+
| Tables_in_whm_cs337f13 |
+-------------------------+
| installation            |
| plant                   |
| residence               |
+-------------------------+
3 rows in set (0.00 sec)
```

Note that `-h mysql` is referencing the host mysql.cs.arizona.edu.

MySQL and SQL resources

MySQL Reference manual
dev.mysql.com/doc/refman/5.5/en/

Learning PHP, MySQL, JavaScript, and CSS, 2nd edition,
by Nixon

Head First PHP & MySQL, by Beighley and Morrison

Sams *Teach Yourself SQL™ in 10 Minutes*, 4th ed.,
by Ben Forta. (On Safari!)

A "deep" book, for future reference:
Relational Theory for Computer Professionals: What Relational Databases Are Really All About, by C.J. Date
(Date worked with E.F. Codd.)

Examining rows with `select`

The `select` statement

SQL's `select` statement is used to display sets of rows:

```
% mysql -u root plants
mysql> select * from plant;
+----+------------------+-------+------------+----------+
| id | plantname        | price | maxheight  | exposure |
+----+------------------+-------+------------+----------+
|  1 | Glossy Abelia    |     6 |          6 | full sun |
|  2 | Grand Fir        |    25 |         25 | full sun |
|  3 | Spanish Fir      |    25 |         15 | full sun |
|  4 | Blue Spanish Fir |    25 |         15 | full sun |
|  5 | Common Boxwood   |     6 |          7 | full sun |
|  6 | Incense Cedar    |    25 |         18 | full sun |
...
22 rows in set (0.00 sec)
```

The query above asks for all columns for all rows in the table `plant`. We're using the database named `plant`**S**.

The rows are considered to be a set. There is no guarantee of the order in which they'll be produced.

`select`, continued

`select * ...` means select all columns but we can specify only certain columns in a certain order:

```
mysql> select maxheight, plantname from plant;
+-----------+-------------------+
| maxheight | plantname         |
+-----------+-------------------+
|         6 | Glossy Abelia     |
|        25 | Grand Fir         |
|        15 | Spanish Fir       |
|        15 | Blue Spanish Fir  |
|         7 | Common Boxwood    |
|        18 | Incense Cedar     |
...
```

`select`, continued

We can perform computations on columns, and give columns new labels:

```
mysql> select
         cast(maxheight*12*2.54 as decimal(8,1))
            "Max height in cm",
         plantname "Name"
         from plant;
+--------------------+--------------------+
| Max height in cm | Name               |
+--------------------+--------------------+
|              182.9 | Glossy Abelia      |
|              762.0 | Grand Fir          |
|              457.2 | Spanish Fir        |
|              457.2 | Blue Spanish Fir   |
|              213.4 | Common Boxwood     |
|              548.6 | Incense Cedar      |
|               30.5 | Harebell           |
...
```

`select`, continued

We can add an `order by` clause to show the rows in the set in sorted order. `desc` indicates descending.

```
mysql> select plantname, maxheight from plant
         order by maxheight desc;
+-----------------------------------+-------------+
| plantname                         | maxheight   |
+-----------------------------------+-------------+
| Grand Fir                         |          25 |
| Himalayan White Pine              |          18 |
| Incense Cedar                     |          18 |
| Spanish Fir                       |          15 |
| Blue Spanish Fir                  |          15 |
| Blue-Needled Japanese White Pine  |          12 |
...
| Harebell                          |           1 |
| Black Mondo Grass                 |         0.5 |
+-----------------------------------+-------------+
22 rows in set (0.00 sec)
```

`select`, continued

A `where` clause can be used to select a subset of the rows:

```
mysql> select plantname, maxheight, price
from plant where exposure = 'indoor';
+----------------------+------------+--------+
| plantname            | maxheight  | price  |
+----------------------+------------+--------+
| Makinoi's Holly Fern |          2 |      4 |
| Sword Fern           |          3 |      4 |
| Asian Saber Fern     |          2 |      4 |
| Japanese Tassel Fern |          2 |      4 |
| Black Mondo Grass    |        0.5 |      3 |
| Hybrid Sweet Olive   |          5 |      6 |
| Sweet Olive          |          5 |      6 |
+----------------------+------------+--------+
7 rows in set (0.00 sec)
```

`select`, continued

We can use `and` and `or` to create arbitrarily complex conditions for `where`:

```
mysql> select plantname, maxheight, price
       from plant where exposure = 'indoor'
       and price < 5
       and maxheight >= 3;
+-------------+------------+--------+
| plantname   | maxheight  | price  |
+-------------+------------+--------+
| Sword Fern  |          3 |      4 |
+-------------+------------+--------+
1 row in set (0.00 sec)
```

`select`, continued

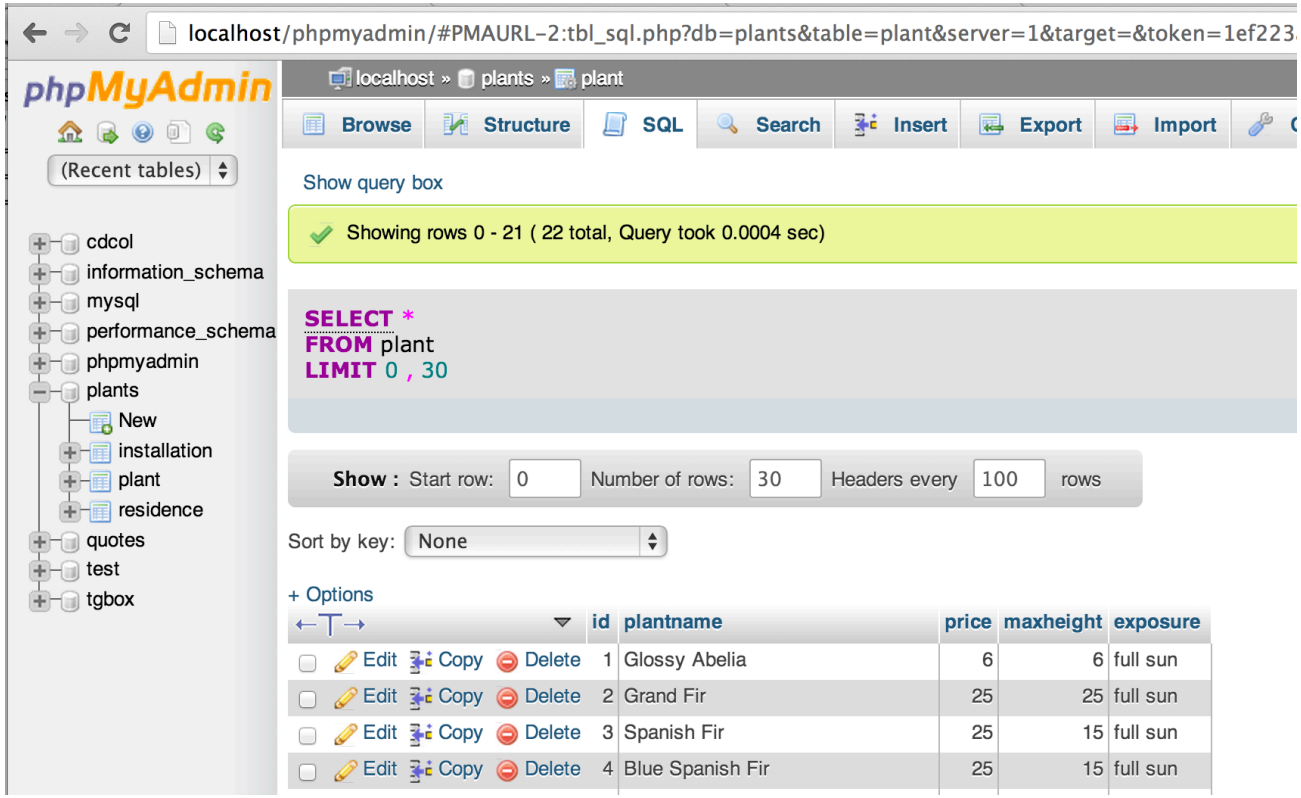
The various clauses of `select` that we've seen must be in this order:

`select`
columns/expressions
`from` *table* (we'll soon see multiple tables)
`where` *condition*
`order by` *specs*

Full details:
http://dev.mysql.com/doc/refman/5.6/en/select.html

GUIs for MySQL (and others)

There are a variety of GUIs for working with MySQL and other relational databases. XAMPP includes phpMyAdmin.



Others are MySQL Workbench, DbVisualizer, and Navicat. IDEs often include database tools, too.

PDO: PHP Data Objects

PDO basics

PHP has several *database abstraction layers* and vendor-specific database extensions.

(see php.net/manual/en/refs.database.php)

We'll be using PHP Data Objects, known as PDO.

There are PDO "drivers" that interface with MySQL, SQL Server, Oracle, SQLite, PostgreSQL, and other databases.

PDO provides an object-based access mechanism. Instead of calling functions we'll make instances of PDO classes and invoke methods on them.

Creating a database connection

pdo1.php creates a connection to the plants database on localhost (i.e., the plants db in XAMPP's MySQL) and asks the server for its version number.

```
<?php
$dsn = "mysql:host=localhost;dbname=plants";
$conn = new PDO($dsn, "root", "");
                // Data source name, user, password
$version = $conn->getattribute(4);  // server version
echo "Server version: $version";
```

Notes:

(1) `new PDO(...)` created an instance of the PDO class.

(2) `$conn->getattribute(4)` invokes the `getattribute` method on that object. The integer 4 happens to mean "server version".

Creating a connection, continued

newconn.php holds a function that decides which database to connect to based on where it's running (as indicated by gethostname()) and what connection parameters to use.

```
<?php
function newconn()
{
   if (gethostname() === "cgi-vm.cs.arizona.edu") {
      $dbname = "whm_cs337f13";
      $user = "cs337f13"; $pw = "tednelson";
      $host = "mysql.cs.arizona.edu";
   } else {
      $dbname = "plants";
      $user = "root"; $pw = ""; $host = "localhost";
   }

   $dsn = "mysql:host=$host;dbname=$dbname"; // Data source name
   $conn = new PDO($dsn, $user, $pw);
   return $conn;
}
```

We'll use this function to make our examples with the plants database work on both XAMPP and on lectura.

Doing a `select` with PDO

First, get a database connection:

```
include("newconn.php"); $conn = newconn();
```

Invoke the connection's `query` method, passing it an SQL query in a string:

```
$plants = $conn->query(
    "select * from plant order by maxheight, plantname");
```

The query method returns an instance of PDOStatement. Its `fetchAll` method returns an array of the results!

```
var_dump($plants->fetchAll(PDO::FETCH_ASSOC));
```

The above is in plants1.php. Let's hit it!

Doing a `select` with PDO, continued

Here's the first part of the output:

```
array(22) {
  [0]=> array(5) {
    ["id"]=> string(2) "17"
    ["plantname"]=> string(17) "Black Mondo Grass"
    ["price"]=> string(1) "3"
    ["maxheight"]=> string(3) "0.5"
    ["exposure"]=> string(6) "indoor"
  }
  [1]=> array(5) {
    ["id"]=> string(1) "7"
    ["plantname"]=> string(8) "Harebell"
    ["price"]=> string(1) "3"
    ["maxheight"]=> string(1) "1"
    ["exposure"]=> string(8) "full sun"
  }
 ...
```

Try fetchAll() with PDO::FETCH_NUM and no arguments, too!

What do we see? What would be an interesting thing to display?

Doing a `select` with PDO, continued

This is plants2.php. What does it do? Run it!

```
include("newconn.php"); $conn = newconn();

$plants = $conn->query(
   "select * from plant order by maxheight, plantname");

foreach ($plants->fetchAll(PDO::FETCH_ASSOC) as $row)
{
   echo bar_for_length($row['maxheight']);
   echo " {$row['plantname']}<br>";
}

function bar_for_length($len)
{
  $len *= 3;
  return "<div class=bar style='width:{$len}em'>$len</div>";
}
```

Our blog with a database

If we wanted our blog entries stored in a database, we might use a table like this:

```
create table entry    /* from blog.sql */
(
   id          bigint not null auto_increment primary key,
   title       varchar(100) not null,
   posted      date,
   text        varchar(2000) not null
   /* no tags yet... */
);
```

What limitations on blog entries does the above imply?

What would need to be changed to display entries from a database instead of blog.txt?

Our blog with a database, continued

Here is load_entries_sql.php. Hit tle_sql.php.

```
function load_entries_sql()
{
   $entries = array();
   $conn = newconn(); // assumes params for blog db

   $stmt = $conn->query("select posted, title, text from entry");

   foreach ($stmt->fetchAll(PDO::FETCH_ASSOC) as $row) {
      $date = $row["posted"];
      $title = $row["title"];
      $text = $row["text"];
      $tags = array(); /* no tags yet.. */

      $entries[] = array("title" => $title, "date" => $date,
                         "text" => $text, "tags" => $tags);
      }

   return $entries;
}
```

The `insert` statement

The previous example assumed some rows in blog. They were created with these lines in blog.sql:

```
insert into entry(title, posted, text)
    values('I\'ve started a blog!',
            '2013-08-28', 'Line 1\nLine 2');

insert into entry(title, posted, text)
    values('Phone trouble...',
            '2013-09-19', 'First (and last) line');
```

The general form is:

insert into *TABLE*(*COL1*, *COL2*, ..., *COLN*)
    values(*VALUE1*, *VALUE2*, ..., *VALUEN*);

<u>Using only what we've seen</u>, how could we get all our blog.txt data into the database?

`insert`, continued

This program <u>almost</u> works:

```
include("load_entries.php");

foreach (load_entries("blog.txt") as $row) {
   echo "insert into entry(title, posted, text)
      values('{$row['title']}', '{$row['date']}',
                  '{$row['text']}');\n";
   }
```

Execution:

```
 % php gen_blog_inserts.php
insert into entry(title, posted, text)
      values('I've started a blog!', '2013-08-26', 'Ligula ...');
insert into entry(title, posted, text)
      values('New pictures!', '2013-08-28', 'Donec et ...');
...
```

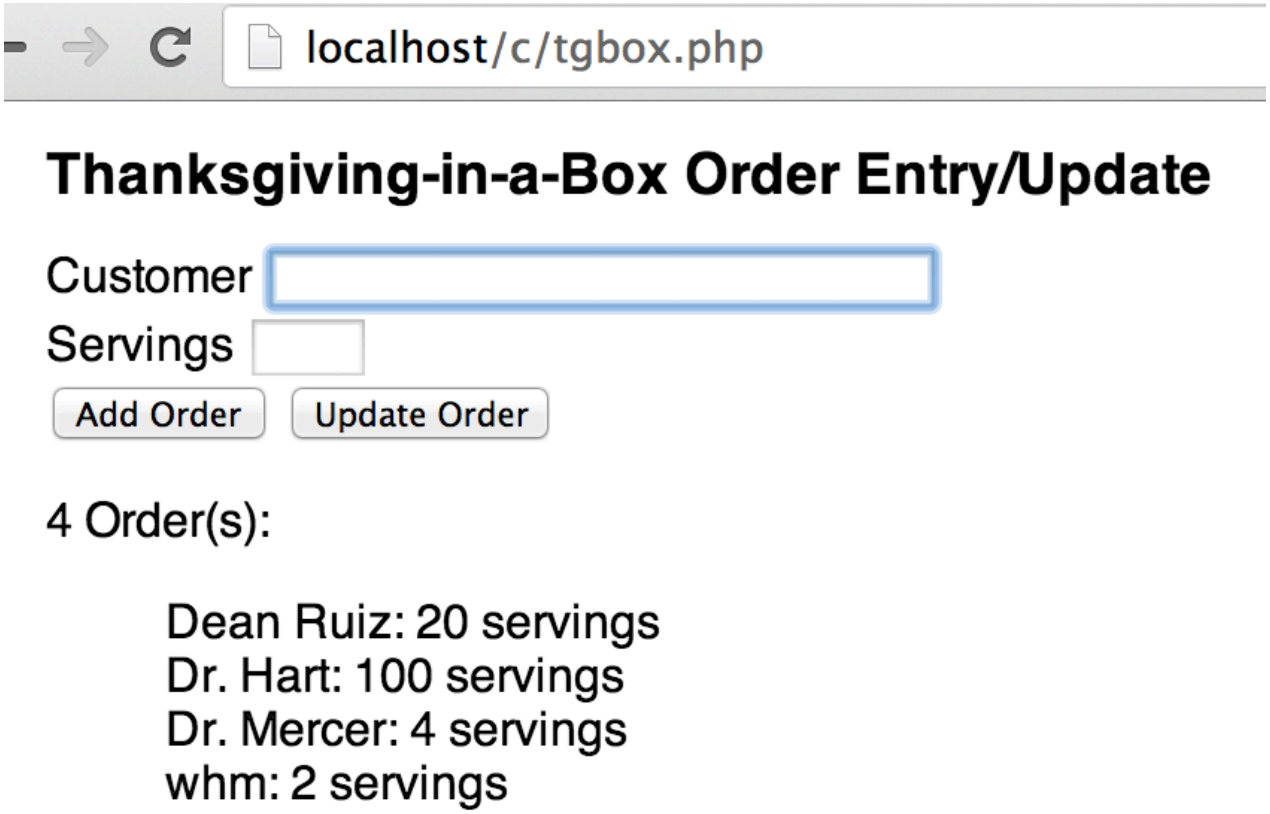
Questions:

(1) Why "almost works"?

(2) This just prints text. How would we use it?

Example: Thanksgiving-in-a-Box

Thanksgiving-in-a-Box eliminates holiday stress by delivering dinners in a box! Clerks at TGB use this app:



Where do we start?

TGB, continued

User Stories (all are for clerk):

I can see what orders have been placed.

I can add an order for some number of servings for a customer. A customer can only have one order.

I can change the number of servings for an existing order.

What information do we need to store? How could it be represented in a table?

TGB, continued

Here's a table definition: (tgbox.sql)

```
create table tgborder
(
    id          bigint not null auto_increment primary key,
    time        datetime not null,
    customer varchar(100) not null,
    servings   int not null
);
```

Notes:

(1) We don't actually need the id column yet.

(2) Clerks don't see the (order) time but we'll go ahead and track it.

TGB, continued

We wrap top-level control in a function named main: (tgbox.php)

```
function main()
{
   global $conn;  $conn = newconn();

   if (count($_POST)) {
      $customer = $_POST["customer"];
      $servings = $_POST["servings"];
      if (isset($_POST["add"]))
         $result = add_order($customer, $servings);
      elseif (isset($_POST["update"]))
         $result = update_order($customer, $servings);
      }

   if ($result) // show result of add/update_order
      echo "<div class=result>$result</div><br>";

   show_orders();
}
```

TGB, continued

This is a helper function that returns the number of orders that a customer has placed. We expect it to return 0 or 1.

A *prepared statement* is used for the select, to avoid a *SQL injection attack*.

```
function orders_for_customer($customer)
{
   global $conn;

   $stmt = $conn->prepare(
                        "select * from tgborder where customer=:cust");
   $stmt->bindParam(':cust', $customer);

   $result = $stmt->execute();
   if (!$result) {
      var_dump($stmt->errorInfo());
      die("error!");
      }

   return count($stmt->fetchAll());
}
```

Sidebar: SQL Injection Attack!

Problem: We need to query for orders for a given customer. We could do this:

```
$conn->query(
  "select * from tgborder where customer='$customer'");
```

What if a mischievous clerk entered the following?

```
'; drop table tgborder; select '
```

We'd execute these three commands:

```
select * from tgborder where customer='';
drop table tgborder;
select '';
```

And Poof! All our orders are gone and our app is dead!

Sidebar, continued

At hand:

```
$conn->query(
  "select * from tgborder where customer='$customer'");
```

This query makes us vulnerable to a *SQL injection attack*.

The essence of a SQL injection attack is that a user has the ability to "inject" additional SQL into a query.

In this case, whatever gets typed in the customer field goes straight into a query string.

This scenario imagines a bad clerk but with a web app that faces the whole world, anybody on the planet could mess with our database.

Sidebar, continued

If we use a "prepared statement" we eliminate the possibility of a SQL injection attack.

The prepare method is given a string with one or more tokens preceded by a colon, like ":cust".

```
$stmt = $conn->prepare(
        "select * from tgborder where customer=:cust")
```

One or more bindParam() calls supply values for the tokens. Then the statement can be executed.

```
$stmt->bindParam(':cust', $customer);
$result = $stmt->execute();
```

<u>Bottom line: Always use prepared statements to provide data values to a query.</u>

TGB, continued

```
function add_order($customer, $servings)
{
   global $conn;

   if (orders_for_customer($customer) != 0)
      return "$customer has already placed an order!";

   $stmt = $conn->prepare("insert into tgborder(time, customer, servings)
                           values(now(), :cust, :serv)");
   $stmt->bindParam(':cust', $customer);
   $stmt->bindParam(':serv', $servings);
   $result = $stmt->execute();

   if (!$result) {
      var_dump($stmt->errorInfo());
      die("error!");
      }
   else
      return "Order placed!";
}
```

TGB, continued

```
function update_order($customer, $servings)
{
   global $conn;

   if (orders_for_customer($customer) == 0)
      return "$customer has not placed an order!";

   $stmt = $conn->prepare(
            "update tgborder set servings=:serv where customer=:cust");
   $stmt->bindParam(':cust', $customer);
   $stmt->bindParam(':serv', $servings);
   $result = $stmt->execute();

   if (!$result) {
      var_dump($stmt->errorInfo());
      die("error!");
      }

   $count = $stmt->rowCount();
   if ($count == 1)
      return "Order updated!";
   else {
      return "Unexpected result: $count rows updated!  Notify a manager!";
      }
}
```

TGB, continued

```
function show_orders()
{
   global $conn;

   $rset = $conn->query(
                  "select * from tgborder order by customer");

   $orders = $rset->fetchAll(PDO::FETCH_ASSOC);
   $count = count($orders);

   if ($count)
      echo "<p>$count Order(s):<blockquote>";
   else
      echo "<p>No orders!  Make some cold calls for hot turkey!";

   foreach ($orders as $order) {
      echo "{$order['customer']}: {$order['servings']} servings<br>\n";
      }
   echo "</blockquote>";
}
```

Sessions
(really a PHP topic)

What can you do with one request?

In some cases a user can easily derive value from a web app with a single request:

What's the current temperature in Tucson?

Show me cs337/fall13/files/a8.pdf.

What's the sum of these ten values?

Show me a map centered on Grant and Campbell.

Play a video sent in a link by a friend.

One request?, continued

Which of the following could be done with a single request to a web app?

Make a hotel reservation.

Book a flight.

Sign up for a class next semester.

Order a list of ten products using a combination of a credit card and a gift certificate.

Enter a series of numbers and be told when their sum exceeds 1000.

Conversational interfaces

Anything that requires a finite set of information can be done with a single request to a web app—all in one big form! (Ouch!)

However, users prefer a more conversational interface, perhaps submitting many data values but spread across a series of views.

What mechanisms have we learned that can be used to turn a really big form into a very conversational interface?

Sessions

As we've learned, PHP applications typically have sub-second lifetimes. Users think they've got a long-running interaction with an application but they're actually just seeing the outputs of a series of invocations of one or more applications.

A "session" is essentially a collection of data values that the server can hold for the lifetime of the application, <u>as perceived by the user</u>.

These server-maintained sessions provide a convenient mechanism to build conversational interfaces.

Sessions in PHP

A PHP application indicates that it wants the server to create a session by calling session_start().

After calling session_start(), a value can be added to the session by assigning a value to a key in $_SESSION.

Here's session1.php:

```
session_start();
$_SESSION['class'] = "CSC 337";
```

In English:

Server, please store the key/value pair class="CSC 337".

Sessions in PHP, continued

Let's hit it with curl:

```
% curl -i localhost/c/session1.php
HTTP/1.1 200 OK
Date: Fri, 15 Nov 2013 07:52:20 GMT
Set-Cookie: PHPSESSID=ag31nng3o7ct4ljtjtao1851k3; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
```

Let's look in XAMPP's temp directory, most recent first:

```
% ls -lt /Applications/XAMPP/xamppfiles/temp
total 52
-rw-------    daemon    20 Nov 15 00:52 sess_ag31nng3o7ct4ljtjtao1851k3
drwxr-xr-x _mysql      68 Nov 13 15:16 mysql
```

Let's look at that file:

```
% sudo cat sess_ag31nng3o7ct4ljtjtao1851k3
class|s:7:"CSC 337";
```

Sessions in PHP, continued

Here's what just happened:

(1) The browser said to the server:

GET /c/session1.php HTTP/1.1

(2) The server ran session1.php. When it reached the session_start() call, it generated a long random string, using it to create the file name sess_ag31nng3o7ct4ljtjtao1851k3. It also generated a Set-Cookie header for a PHPSESSID cookie with that same random string as its value.

(3) When session1.php exited PHP wrote a representation of the contents of $_SESSION (`class|s:7:"CSC 337";`) to the file. (It is *serialized*.)

(4) The browser saw the Set-Cookie header in the response and stored the PHPSESSID cookie, which references the XAMPP temp file named sess_ag31nng3o7ct4ljtjtao1851k3.

Of course, the PHPSESSID cookie will be sent to the server the next time we hit localhost.

Sessions in PHP, continued

This is session2.php:

```
var_dump($_COOKIE);
var_dump($_SESSION);
echo "calling session_start\n";
session_start();
var_dump($_SESSION);
```

Its output follows. Explain it!

```
array(1) {
  ["PHPSESSID"]=>  string(26) "ag31nng3o7ct4ljtjtao1851k3"
}
Notice:  Undefined variable: _SESSION in session2.php on line 4
NULL
calling session_start
array(1) {
  ["class"]=>  string(7) "CSC 337"
}
```

PHP sessions, continued

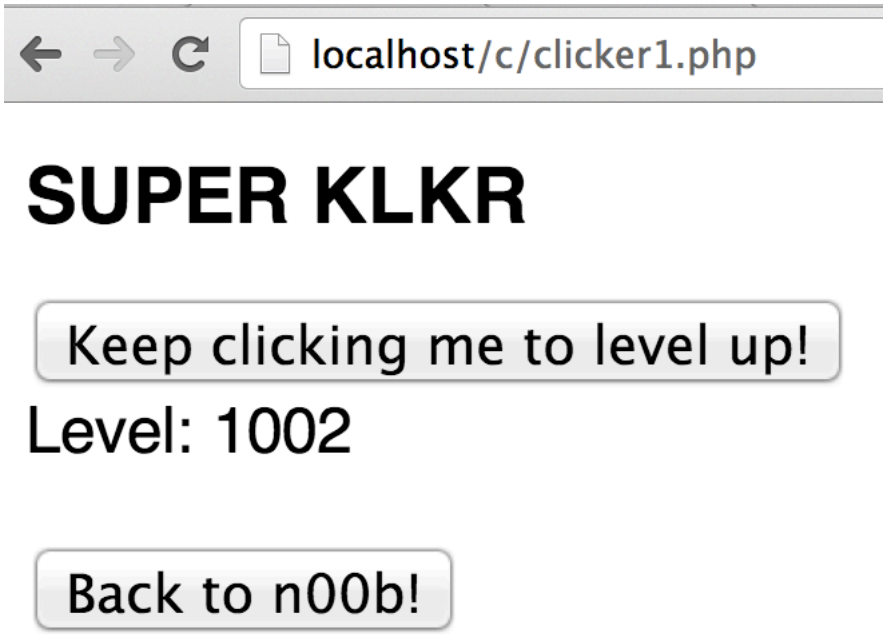
If a PHPSESSID cookie is present when session_start() is called, the data in sess_*PHPSESSID* is deserialized and used to populate $_SESSION.

The short story: If we simply call session_start() at the start of our program, we get per-user saving and loading of $_SESSION.

Hmm...What if a user has disabled cookies?

Super KLKR hacked!

Our Super KLKR game has been hacked! A user has reached level 1002, and won a new car!



How'd they do it? How can we secure Super KLKR?

Super KLKR, continued

clicker1.php: (minus the boilerplate...)

```
if (isset($_POST["clicks"]))
   $clicks = $_POST["clicks"];
else
   $clicks = 0;

if (isset($_POST["click"]))
   $clicks += 1;

if (isset($_POST["n00b"]))
   header("Location: {$_SERVER['PHP_SELF']}");

?>
<form method=post>
  <input type=submit name=click value="Keep clicking me to level
up!">
  <br>
  Level: <?= (int)($clicks/10)+1 ?>
  <br><br>
  <input type=submit name=n00b value="Back to n00b!">
  <input type=hidden name=clicks value=<?=$clicks?>>
</form>
```

How was this outrage committed?!

Super KLKR v2!

clicker2.php:

```
session_start();
if (!isset($_SESSION["clicks"]))
   $_SESSION["clicks"] = 0;

if (isset($_POST["click"]))
   $_SESSION["clicks"] += 1;

if (isset($_POST["n00b"])) {
   header("Location: {$_SERVER['PHP_SELF']}");
   session_destroy();
   }

?>
<form method=post>
   <input type=submit name=click value="Keep clicking me to level up!">
   <br>
   Level: <?= (int)($_SESSION["clicks"]/10)+1 ?>
   <br><br>
   <input type=submit name=n00b value="Back to n00b!">
</form>
```

Serialization
(another PHP topic)

The serialize() function

The value of $_SESSION is *serialized* into `sess_...` files maintained by the server. We can use that same serialization machinery.

The serialize() function produces a string representation of a PHP value.

```
php > var_dump(serialize(37));
string(5) "i:37;"

php > var_dump(serialize(3.7));
string(21) "d:3.7000000000000002;"

php > var_dump(serialize(array(37, 3.7)));
string(40) "a:2:{i:0;i:37;i:1;d:3.7000000000000002;}"
```

Why does the last serialization contain i:0 and i:1?

serialize(), continued

Arbitrarily complex values can be serialized:

```
php > $a = array(true, false, null);

php > $b = array("I" => 1, "V" => 5, "X" => 10);

php > $c = array("first" => $a, "second" => $b);

php > $s = serialize($c);

php > echo $s;
a:2:{s:5:"first";a:3:{i:0;b:1;i:1;b:0;i:2;N;}s:6:"second";a:3:{s:
1:"I";i:1;s:1:"V";i:5;s:1:"X";i:10;}}

php > echo strlen($s);
102
```

How are boolean values and null represented?

unserialize()

The unserialize() function is the inverse of serialize(): it takes a string produced by serialize() and reconstitutes the value that was serialized.

```
php > echo $s;
a:3:{i:0;i:10;i:1;i:20;i:2;a:3:{s:3:"one";i:1;s:3:"two";i:2;s:4:"true";b:1;}}

php > $recreated = unserialize($s);

php > var_dump($recreated);
array(3) {
  [0]=> int(10)
  [1]=> int(20)
  [2]=> array(3) {
    ["one"]=> int(1)
    ["two"]=> int(2)
    ["true"]=> bool(true)
  }
}
```

How can serialize() and unserialize() be put to use?

Storing serialized data in an RDBMS

We can create a hybrid storage system by using wide columns to store serialized data.

```
create table players (        -- serialstore.sql
   name  varchar(50),
   data   text                -- Up to 64k bytes.  longtext is 2Gb
   );
```

```
mysql> select * from players;
+--------+-------------------------------------
| name   | data
+--------+-------------------------------------
| whm    | a:3:{s:4:"wins";i:7;s:6:"losses";....
| j-than | a:3:{s:4:"wins";i:10;s:6:"losses";...
| bruce  | a:3:{s:4:"wins";i:3;s:6:"losses";....
+--------+-------------------------------------
3 rows in set (0.00 sec)
```

Serialized data in an RDBMS, continued

We might write routines to load and save **all** data for a player in one shot.

Here's code that loads player data, increments the number of wins, and saves it:

```
$who = $_GET["winner"]; // serialstore2.php

$data = get_player_data($who);

$data["wins"] += 1;

echo "{$data['wins']} wins for $who";

save_player_data($who, $data);
```

Serialized data in an RDBMS, continued

Here's the function that loads player data.

```
function load_player_data($name)
{
    $conn = newconn();
    $stmt = $conn->prepare(
        "select data from players where name=:name");
    $stmt->bindParam(":name", $name);
    $result = $stmt->execute();
    if (!$result) pdo_die($stmt);

    $rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
    assert(count($rows) == 1);

    $data = unserialize($rows[0]["data"]);

    return $data;
}
```

save_player_data(...) is similar. Both are in serialstore2.php.

Serialized data in an RDBMS, continued

In essence, we've turned a relational database into an object store.

Is this a good idea? Is it heresy?

> "A principle of RDBMS *normalization* is that at each row/column intersection there should be an *atomic* value."

What does this make easier?

What does this make harder?

Blog Overhaul

Blog overhaul

Some major revisions have been made to the blog:

- Uses database for everything except images.
- Can host blogs for any number of owners.
- Blog owner sign-up/sign-in support (using sessions).
- Simple control panel for blog owners.
- Refactored into multiple source files.
- Went from about 300 lines to over 700 lines of code.
- Hit http://cgi.cs.arizona.edu/classes/cs337/fall13/blog/
- Files in /cs/cgi/classes/cs337/fall13/blog on CS machines.

New source file structure

index.php	Sign-in/sign-up code
control.php	Control panel
blog.php	Displays entries
images.php	Image library
newentry.php	New entry dialog
load_entries.php	Loads entries from database
sql.php	Various database operations as functions
utils.php	Miscellaneous utilities

index.php—sign-in/sign-up

If we hit a URL that names a directory and index.php is present, it is run.

Sign-up is "streamlined"—just enter desired user name and password, and click Sign Up.

Errors are produced if name is already used or has disallowed characters, or if password is too short.

index.php, continued

```
include_once("sql.php");
session_start();

if ($user = @$_SESSION["blog_owner_id"]) {        // If already signed in,
   header("Location: control.php");                // go to owner's control panel.
   }

elseif (isset($_GET["who"])) {  // can hit .../fall13/blog?who=whm
   header("Location: blog.php?who={$_GET['who']}");
   }

elseif (isset($_POST["owner"]) && isset($_POST["pw"])) {
   $user = $_POST["owner"]; $pw = $_POST["pw"];
   if (isset($_POST["signin"]))
      signin($user, $pw);
   elseif (isset($_POST["signup"]))
      signup($user, $pw);
   else
      show_signin();     // shows form
   }
else show_signin();
```

The owner table

Here's the owner table. Along with a unique id, rows have the owner's username and an encrypted password.

```
mysql> select * from owner;
+----+-------+----------------------...------+
| id | owner | password                       |
+----------------------------------...------+
|  1 | test  | 04413e8e0a4558ca6a1...12354 |
|  2 | whm   | 94c07b7e6e2b129c300...19973 |
|  3 | bkm   | 34fe8cd7e2f4d3c3671...34028 |
+----------------------------------...------+
3 rows in set (0.00 sec)
```

index.php, continued

This function handles new user sign-up:

```
function signup($owner, $pw)
{
   if (sql_id_for_owner($owner))  // returns id iff owner exists
      show_signin("sorry, that's taken!");

   if (strlen($pw) < 3)  show_signin("password too short!");

   $ents = htmlentities($owner, ENT_QUOTES);
   if ($ents != $owner) show_signin("disallowed characters!");

   if (strlen($owner) > 50) show_signin("name too long!");

   $owner_id = sql_add_owner($owner, $pw);
   if ($owner_id) {
      assert(mkdir("images/$owner_id"));
      start_with_id($owner_id);
      }
}
```

index.php, continued

This function adds a new owner and password to the owner table.

```
function sql_add_owner($owner, $password) // in sql.php
{
   $salted = sha1($owner) . $password;
   $hashed = sha1($salted);

   $conn = getconn();
   $stmt = $conn->prepare(
      "insert into owner(owner, password) values(:owner, :hashed)");
   $stmt->bindParam(":owner", $owner);
   $stmt->bindParam(":hashed", $hashed);

   $result = $stmt->execute();
   $owner_id = $conn->lastInsertId();

   if (!$result) pdo_die($stmt);

   return $owner_id;
}
```

We use sha1(...) to turn a string like "secret" into a *digest* like e5e9fa1ba31ecd1ae84f75caaa474f3a663f05f4. We use some simple "salting" to make it a little harder to crack passwords, should the owner table be stolen.
<u>With PHP 5.5 we'd use password_hash(...) instead.</u>

index.php, continued

```
function signin($owner, $pw) // Called when "Sign In" is clicked
{
   $owner_id =
      sql_check_password($owner, $_POST["pw"]);
   if ($owner_id)
      start_with_id($owner_id);
   else
      show_signin("login incorrect");
}

function start_with_id($owner_id)
{
   // Set owner id in session and load control panel view
   $_SESSION["blog_owner_id"] = $owner_id;
   header("Location: control.php");
   exit();
}
```

control.php

```
$owner_id = check_signedin(); // on next slide

$owner = sql_owner_for_id($owner_id);

if (isset($_POST["new"])) {
   header("Location: newentry.php"); exit();
   }

elseif (isset($_POST["blog"])) {
   header("Location: blog.php?who=$owner");
   exit();
   }

elseif (isset($_POST["images"])) {
   header("Location: images.php"); exit();
   }

elseif (isset($_POST["signout"])) {
   session_destroy();
   header("Location: ."); exit();
   }
```

control.php, continued

KEY MECHANISM

start_with_id(...) stores the owner's id in the session
check_signedin() looks for an owner's id in the session
User is signed in iff there's an owner's id in the session!

```
function start_with_id($owner_id) // index.php
{
   $_SESSION["blog_owner_id"] = $owner_id;
   header("Location: control.php");
   exit();
}

function check_signedin() // utils.php
{
   session_start();

   if (!isset($_SESSION["blog_owner_id"])) {
      header("Location: index.php"); exit();
      }

   return $_SESSION["blog_owner_id"];
}
```

`entry` table, for blog entries

A table for blog entries was shown on slide 31. It now has an owner_id column:

```
create table entry (
   id          bigint not null auto_increment primary key,
   owner_id    bigint not null,
   title       varchar(100) not null,
   posted      date,
   text        varchar(2000) not null );
```

Some data:

```
mysql> select * from entry;
+----+----------+----------------------+------------+-----------
| id | owner_id | title                | posted     | text
+----+----------+----------------------+------------+-----------
|  1 |        1 | I've started a blog! | 2013-08-28 | Line 1 ...
|  2 |        1 | Phone trouble...     | 2013-09-19 | First (a...
|  3 |        2 | My new blog          | 2013-08-26 | Donec ...
|  4 |        3 | New pictures!        | 2013-08-28 | Convallis...
```

load_entries($owner_id)

load_entries(...) uses a where clause to select only entries by a specific owner:

```
function load_entries($owner_id)
{
   $conn = getconn();

   $entries = array();

   $stmt = $conn->prepare(
      "select id, posted, title, text from entry where owner_id=:oid");
   $stmt->bindParam(':oid', $owner_id);
   $stmt->execute();

   foreach ($stmt->fetchAll(PDO::FETCH_ASSOC) as $row) {
      $date = $row["posted"];
      ...get others, too...

      $entries[] =
            array("title" => $title, "date" => $date,  ...);
      }

   return $entries;
}
```

blog.php—display blog entries

```
session_start(); // With PHP 5.3 must precede other output

write_header();

if (isset($_GET["who"]))  // anybody can hit .../blog?who=owner-name
   show_blog($_GET["who"]);
else
   die("How'd you get here?!");

function show_blog($owner)
{
   $owner_id = sql_id_for_owner($_GET["who"]);
   if (!$owner_id)
      die("Somebody gave you a bum blog!");

   echo "<div id=main>\n";

   if (isset($_SESSION["blog_owner_id"])) {  // Add B.C. link if blog owner
      echo "<div ...><a class=bwlink href=control.php>Blog Control</a></div>";
      }

   $entries = load_entries($owner_id);
```

sql_id_for_owner($owner)

Here's one way to write sql_id_for_owner:

```
function sql_id_for_owner0($owner)
{
   $conn = getconn();

   $stmt = $conn->prepare(
      "select id from owner where owner=:owner");
   $stmt->bindParam(':owner', $owner);
   $result = $stmt->execute();

   if (!$result) pdo_die($stmt);

   $result = $stmt->fetchAll();

   assert(count($result) <= 1);  // We expect one row or no rows.
   if (count($result) == 1)
      return $result[0]["id"];
   else
      return null;
}
```

What will sql_owner_for_id($id) look like?

sql_col_for_entity(...)

Lets write a function that will do that sort of lookup for any column of any table:

```
function sql_col_for_entity($column, $table, $eqcolumn, $value) {...}
```

We'll use it like this:

```
function sql_id_for_owner($owner)
{
   return sql_col_for_entity("id", "owner", "owner", $owner);
}

function sql_id_for_tag($tag)
{
   return sql_col_for_entity("id", "tag", "tag", $tag);
}

function sql_owner_for_id($id)
{
   return sql_col_for_entity("owner", "owner", "id", $id);
}
```

sql_col_for_entity(...)

Here's the generalized function:

```
function sql_col_for_entity($column, $table, $eqcolumn, $value)
{
   $conn = getconn();

   $stmt = $conn->prepare(
      "select $column from $table where $eqcolumn=:value");
   $stmt->bindParam(':value', $value);
   $result = $stmt->execute();

   if (!$result)
      pdo_die($stmt);

   $result = $stmt->fetchAll();

   assert(count($result) <= 1);
   if (count($result) != 0)
      return $result[0][$column];
   else
      return null;
}
```

We can't use prepared statement tokens for $table and $eqcolumn so we need to be sure that no user-supplied value can end up in those variables!

The trouble with tags

A principle of RDBMS *normalization* is that at each row/column intersection there should be an "atomic" value.

We might add a `tags` column to the entry table, but it would have values like these:

UA, 337
World Series, 337, Panda Express

Those values aren't atomic—they represent multiple values.

<u>A reasonable shortcut for prototypes and 337 projects is to simply ignore the principle at hand and store values like "UA, 337" in a tags column.</u>

But lets briefly consider the right way to do it.

Tags, continued

We use two tables. One holds only tags, and an id for each:

```
create table tag
(
   id     bigint not null auto_increment primary key,
   tag    varchar(50)
);
```

The other table associates a blog entry id with a tag id:

```
create table entry_tag
(
   entry_id      bigint not null,
   tag_id        bigint not null
);
```

Each row in entry_tag indicates that a particular tag appears in a particular entry.

Tags, continued

Consider these tables:

```
mysql> select * from tag;
+----+----------------+
| id | tag            |
+----+----------------+
|  1 | 337            |
|  2 | UA             |
|  3 | Panda Express  |
+----+----------------+

mysql> select * from entry_tag;
+----------+---------+
| entry_id | tag_id  |
+----------+---------+
|        1 |       1 |
|        1 |       2 |
|        2 |       1 |
|        2 |       3 |
+----------+---------+
```

Which blog entries have which tags?
How many blog entries are there in all?

Tags, continued

tag

id	tag
1	337
2	UA
3	Panda Express

entry_tag

entry_id	tag_id
1	1
1	2
2	1
2	3

To get the tag <u>names</u> for an entry's tags in a single query, we can use a "join".

"joins" in SQL

If we specify multiple tables to select from, a *cartesian product* is formed:

```
mysql> select * from entry_tag, tag;
+----------+--------+----+----------------+
| entry_id | tag_id | id | tag            |
+----------+--------+----+----------------+
|        1 |      1 |  1 | 337            |
|        1 |      1 |  2 | UA             |
|        1 |      1 |  3 | Panda Express  |
|        1 |      2 |  1 | 337            |
|        1 |      2 |  2 | UA             |
|        1 |      2 |  3 | Panda Express  |
|        2 |      1 |  1 | 337            |
|        2 |      1 |  2 | UA             |
|        2 |      1 |  3 | Panda Express  |
|        2 |      3 |  1 | 337            |
|        2 |      3 |  2 | UA             |
|        2 |      3 |  3 | Panda Express  |
+----------+--------+----+----------------+
12 rows in set (0.00 sec)
```

All combinations of all rows from both tables are present.

"joins" in SQL, continued

Lets keep only the rows in the cartesian product where entry_id is 2 and tag_id is the id of the tag.

```
mysql> select * from entry_tag, tag
         where entry_id=2 and tag_id=id;
+----------+--------+----+----------------+
| entry_id | tag_id | id | tag            |
+----------+--------+----+----------------+
|        2 |      1 |  1 | 337            |
|        2 |      3 |  3 | Panda Express  |
+----------+--------+----+----------------+
2 rows in set (0.00 sec)
```

Of course, we don't need all the columns, only the tag itself:

```
mysql> select tag from entry_tag, tag
       where entry_id=2 and tag_id=tag.id;
+----------------+
| tag            |
+----------------+
| 337            |
| Panda Express  |
+----------------+
2 rows in set (0.00 sec)
```

tags_for_entry(...)

Let's wrap some PHP around that join:

```
function tags_for_entry($entry_id) // in sql.php
{
   $conn = getconn();

   $tags = array();

   $stmt = $conn->prepare(
      "select tag from entry_tag, tag
       where entry_id=:eid and tag.id=entry_tag.tag_id
       order by tag");
   $stmt->bindParam(':eid', $entry_id);
   $stmt->execute();

   foreach ($stmt->fetchAll(PDO::FETCH_ASSOC) as $row) {
      $tags[] = $row['tag'];
      }

   return $tags;
}
```

"Will this be on the final?"

No, joins in SQL won't be on the final!

Could we have achieved the same result without using a join?

Remember: If you use MySQL for your project, it's fine to take shortcuts like just storing stuff like tags in a comma-separated string!

p.s.
Here's a harder problem: When creating a new blog entry we want to show checkboxes only for tags created by the blog's owner. A three-table join is needed for that. See if you can figure it out! (Solution: sql_get_tags($owner_id) in sql.php.)