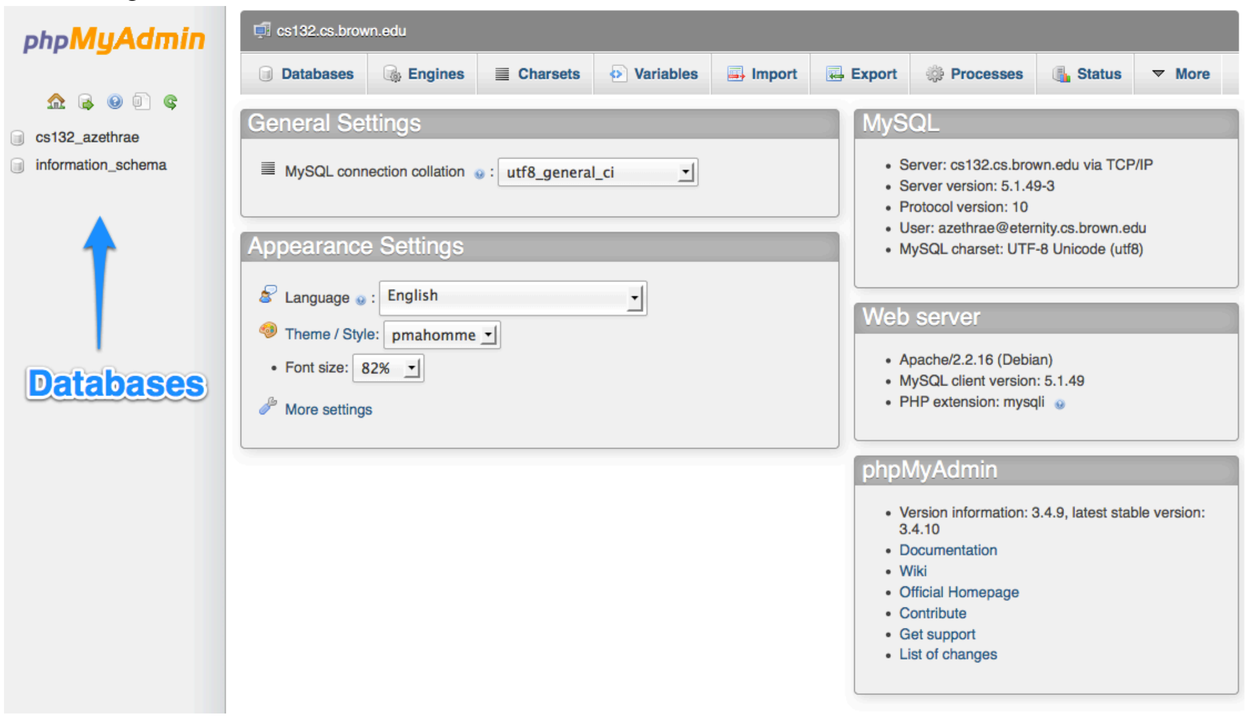
SQL/LAMP

During this lab, you'll be using your MySQL install on your web space. If anything appears to be acting up it might be because you missed a command when you initialized your web space; talk to a TA and we'll have you up and running in a few seconds.

Part 1: How to use phpMyAdmin

phpMyAdmin is an admin panel for your MySQL database which can make getting a good overview of your data, tables, and schema nice and easy. It allows you to run queries straight through a web interface without having to set up a database connection yourself. This makes it perfect for playing around with when learning SQL. Go to your web folder, you should see a 'phpmyadmin' folder link in the directory listing. Click on it and you're in!
You'll be greeted with a screen like this:



MySQL connection collation: utf8_general_ci

Language: English
Theme / Style: pmahomme
Font size: 82%

- Server: cs132.cs.brown.edu via TCP/IP
- Server version: 5.1.49-3
- Protocol version: 10
- User: azethrae@eternity.cs.brown.edu
- MySQL charset: UTF-8 Unicode (utf8)

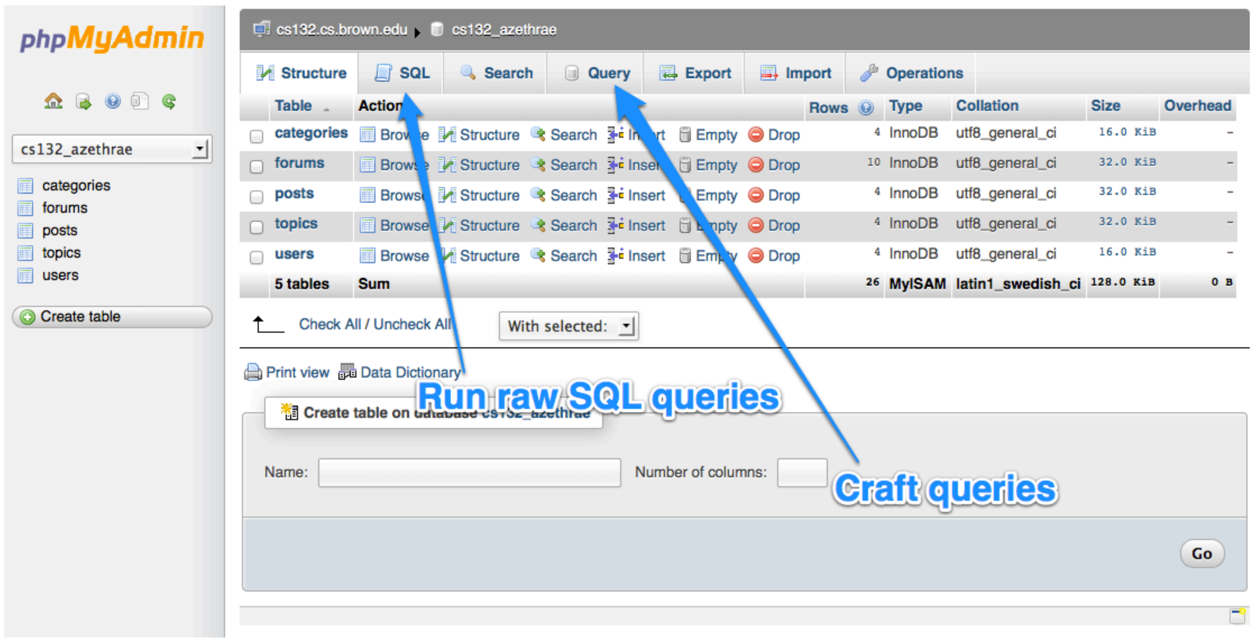
- Apache/2.2.16 (Debian)
- MySQL client version: 5.1.49
- PHP extension: mysqli

- Version information: 3.4.9, latest stable version: 3.4.10

Your databases are listed on the left. You'll want to work in cs132_<your login>; information_schema is used internally by phpMyAdmin.

If you click on your database, you'll see a list of all of the tables. Tables are the core of a SQL database, they're composed of columns (each containing a type of data, for example a time or a URL) and rows (each containing one logical entry).

In this page you can also click through to SQL to run a written out SQL query. You'll be doing a lot of that in this lab. There's a 'craft queries' section too. Avoid it for this lab, it's not very flexible.



cs132.cs.brown.edu ▸ cs132_azethrae

cs132_azethrae: categories, forums, posts, topics, users

	Table	Action	Rows	Type	Collation	Size	Overhead
☐	categories	Browse Structure Search Insert Empty Drop	4	InnoDB	utf8_general_ci	16.0 KiB	-
☐	forums	Browse Structure Search Insert Empty Drop	10	InnoDB	utf8_general_ci	32.0 KiB	-
☐	posts	Browse Structure Search Insert Empty Drop	4	InnoDB	utf8_general_ci	32.0 KiB	-
☐	topics	Browse Structure Search Insert Empty Drop	4	InnoDB	utf8_general_ci	32.0 KiB	-
☐	users	Browse Structure Search Insert Empty Drop	4	InnoDB	utf8_general_ci	16.0 KiB	-
	5 tables	Sum	26	MyISAM	latin1_swedish_ci	128.0 KiB	0 B

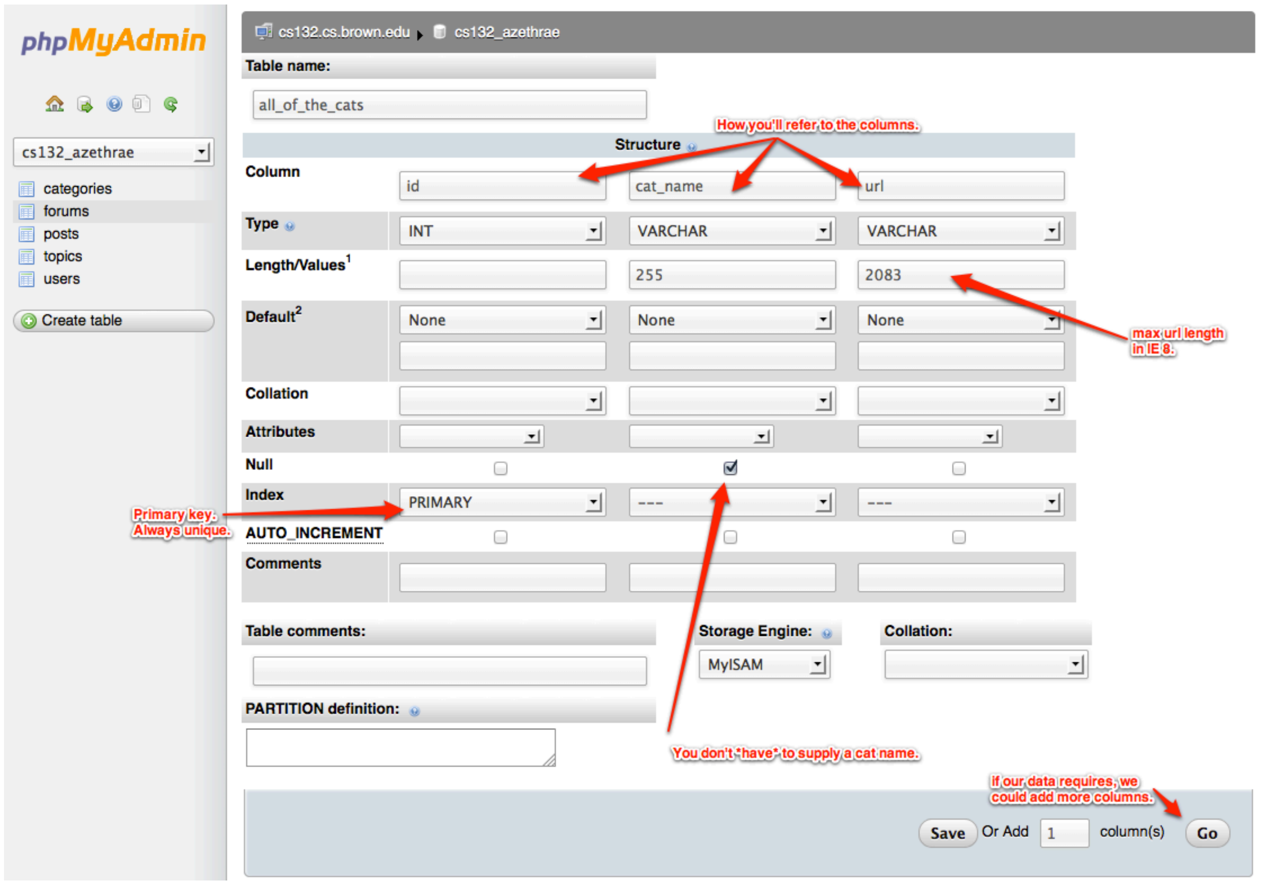
Check All / Uncheck All   With selected:

Print view   Data Dictionary

Create table on database cs132_azethrae

Name: ______   Number of columns: ______

You'll be making lots of tables yourself with straight up SQL (in the SQL tab). If you were to click 'create table' in the left bar however, you'd be presented with an interface as below.


phpMyAdmin
Create table
How you'll refer to the columns.
max url length in IE 8.
Primary key. Always unique.
You don't *have* to supply a cat name.
If our data requires, we could add more columns.
Save
Go

cs132.cs.brown.edu ▸ cs132_azethrae

Table name: all_of_the_cats

Structure			
Column	id	cat_name	url
Type	INT	VARCHAR	VARCHAR
Length/Values[1]		255	2083
Default[2]	None	None	None
Collation			
Attributes			
Null	☐	✓	☐
Index	PRIMARY	---	---
AUTO_INCREMENT	☐	☐	☐
Comments			

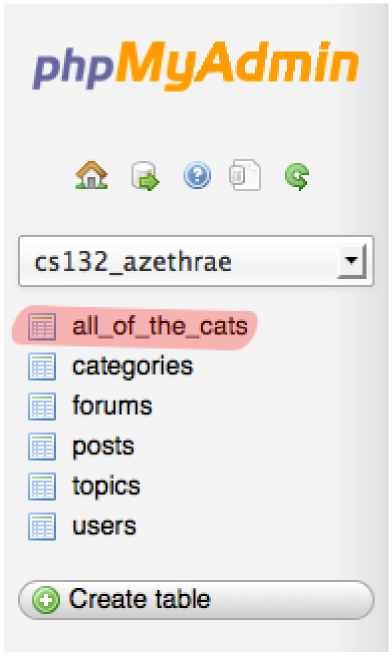
Table comments: Storage Engine: MyISAM   Collation:

PARTITION definition:

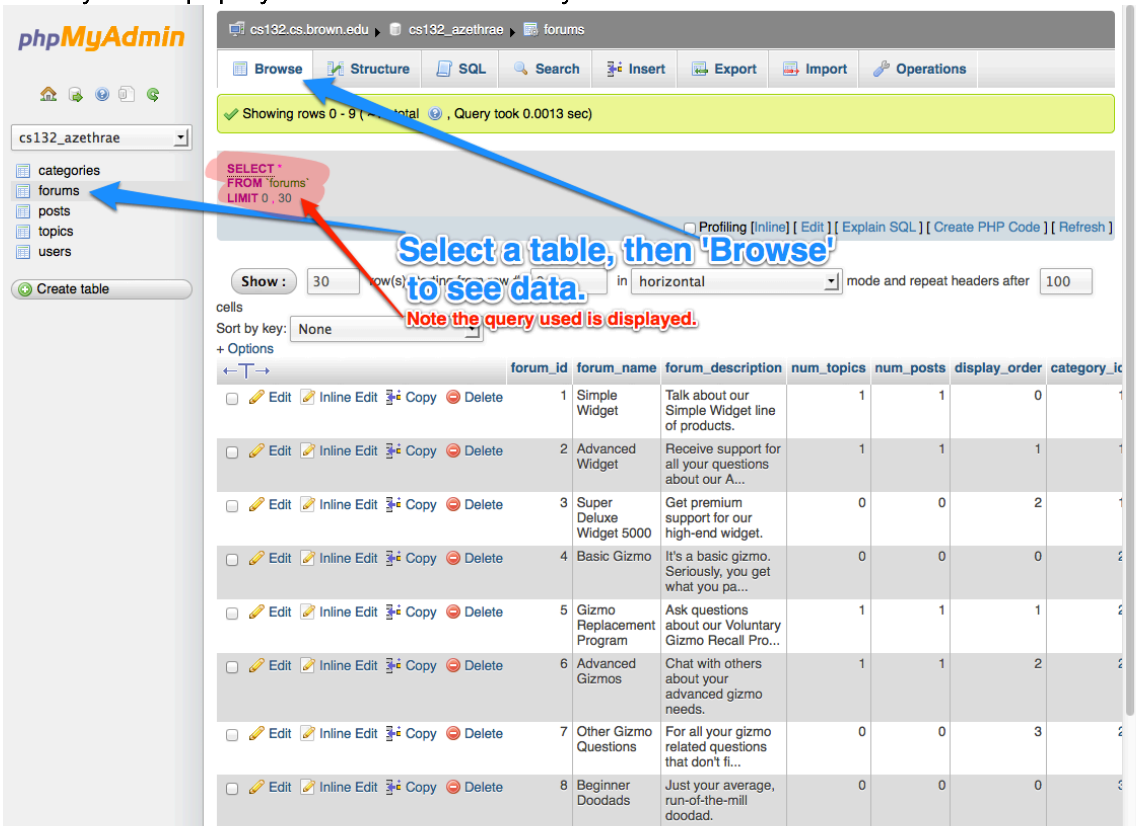
Save   Or Add 1 column(s)   Go

We've created a table here called 'all_of_the_cats'. It's great because it holds cat names and

URLs to pictures of cats. Each entry also has a primary key which is always unique. This enforces uniqueness between rows in the table.



After making a new table, it will appear in the left sidebar.
It's often useful to have a way of manually inspecting the data in your database, or changing the data in it by hand. phpMyAdmin makes this easy:



Showing rows 0 - 9 (~ total , Query took 0.0013 sec)

```
SELECT *
FROM `forums`
LIMIT 0 , 30
```

Profiling [Inline] [Edit] [Explain SQL] [Create PHP Code] [Refresh]

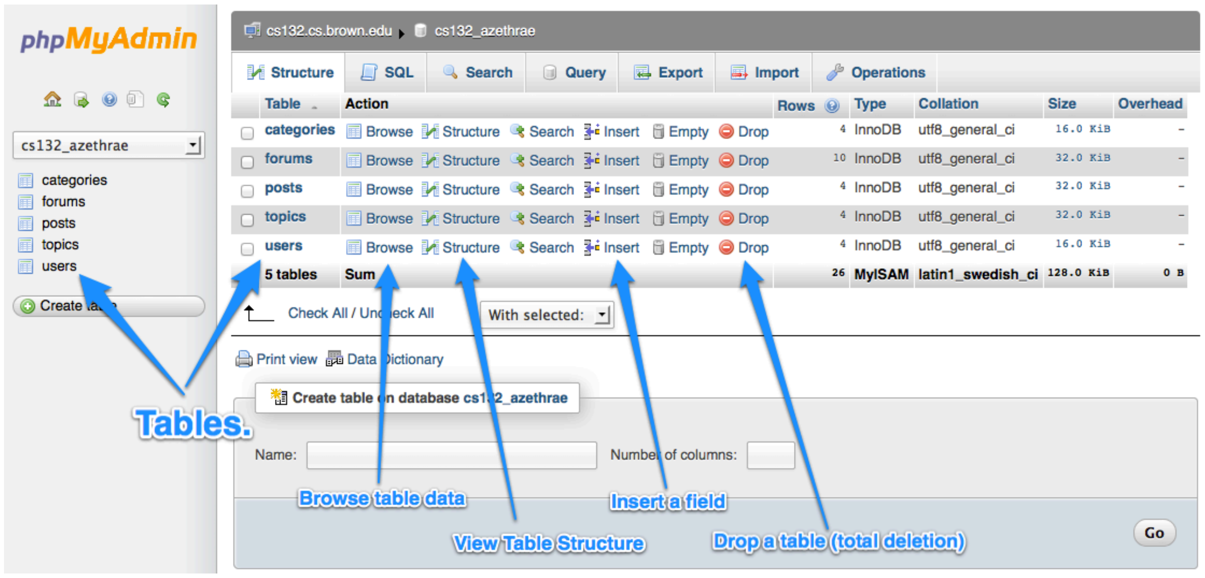
Show : 30 row(s) starting from row # in horizontal mode and repeat headers after 100 cells

Sort by key: None

+ Options

	forum_id	forum_name	forum_description	num_topics	num_posts	display_order	category_i
Edit Inline Edit Copy Delete	1	Simple Widget	Talk about our Simple Widget line of products.	1	1	0	1
Edit Inline Edit Copy Delete	2	Advanced Widget	Receive support for all your questions about our A...	1	1	1	1
Edit Inline Edit Copy Delete	3	Super Deluxe Widget 5000	Get premium support for our high-end widget.	0	0	2	1
Edit Inline Edit Copy Delete	4	Basic Gizmo	It's a basic gizmo. Seriously, you get what you pa...	0	0	0	2
Edit Inline Edit Copy Delete	5	Gizmo Replacement Program	Ask questions about our Voluntary Gizmo Recall Pro...	1	1	1	2
Edit Inline Edit Copy Delete	6	Advanced Gizmos	Chat with others about your advanced gizmo needs.	1	1	2	2
Edit Inline Edit Copy Delete	7	Other Gizmo Questions	For all your gizmo related questions that don't fi...	0	0	3	2
Edit Inline Edit Copy Delete	8	Beginner Doodads	Just your average, run-of-the-mill doodad.	0	0	0	3

You can use the 'structure' tab to make structural changes to existing tables. Or Drop them.



phpMyAdmin

cs132_azethrae

categories
forums
posts
topics
users

Create table

cs132.cs.brown.edu ▸ cs132_azethrae

Structure | SQL | Search | Query | Export | Import | Operations

Table	Action						Rows	Type	Collation	Size	Overhead
categories	Browse	Structure	Search	Insert	Empty	Drop	4	InnoDB	utf8_general_ci	16.0 KiB	-
forums	Browse	Structure	Search	Insert	Empty	Drop	10	InnoDB	utf8_general_ci	32.0 KiB	-
posts	Browse	Structure	Search	Insert	Empty	Drop	4	InnoDB	utf8_general_ci	32.0 KiB	-
topics	Browse	Structure	Search	Insert	Empty	Drop	4	InnoDB	utf8_general_ci	32.0 KiB	-
users	Browse	Structure	Search	Insert	Empty	Drop	4	InnoDB	utf8_general_ci	16.0 KiB	-
5 tables	Sum						26	MyISAM	latin1_swedish_ci	128.0 KiB	0 B

Check All / Uncheck All With selected:

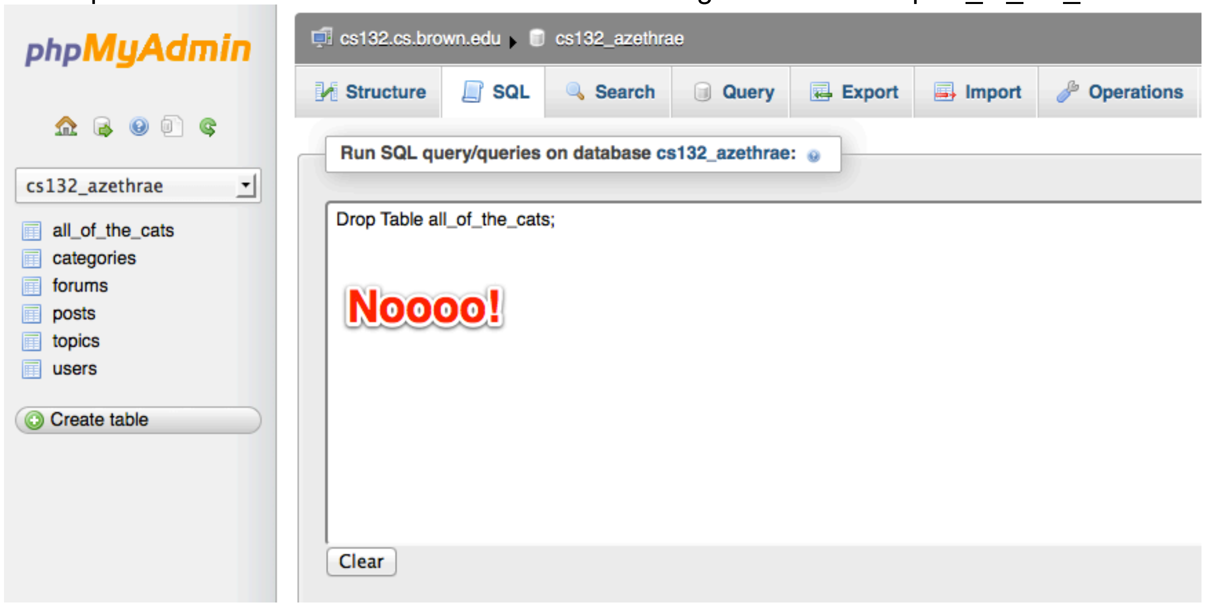
Print view Data Dictionary

Create table on database cs132_azethrae

Name: Number of columns:

Go

Drop is a euphemism for 'delete forever'. It's easy to drop a table in phpMyAdmin, so be wary. Dropping a table in SQL is just as easy. Here we've opened up the SQL tab in order to write queries and commands directly. As mentioned, this page is where you'll be doing most of the work for the rest of the lab.
In this particular case we've made the heart-wrenching decision to drop all_of_the_cats.



phpMyAdmin

cs132_azethrae

all_of_the_cats
categories
forums
posts
topics
users

Create table

cs132.cs.brown.edu ▸ cs132_azethrae

Structure | SQL | Search | Query | Export | Import | Operations

Run SQL query/queries on database cs132_azethrae:

```
Drop Table all_of_the_cats;
```

Clear

If there's anything about phpMyAdmin's interface which seems counter-intuitive, wave down a TA!

Part 2: SQL Table Definition

The big picture:

```
CREATE TABLE <table-name>(
      [column-definition]*
      [constraints]*
)
```

where [column-definition] contains a comma separated list of column names with types;
`[constraints] contains Primary key, foreign key, and other meta-data attributes of columns`

We'll learn this by example.

TODO In SQL box, enter the following code:

```
CREATE TABLE banks (
    bank_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    bank_name VARCHAR(200) NOT NULL,
    bank_city VARCHAR(200) NOT NULL,
    PRIMARY KEY(bank_id)
) ENGINE=InnoDB CHARSET=UTF8;
```

This creates a table called "banks" with 3 columns: "bank_id", "bank_name" and "bank_city".

```
bank_id INT UNSIGNED NOT NULL AUTO_INCREMENT
```

says the bank_id is unsigned integer that can't be null. AUTO_INCREMENT tells SQL to automatically assign sequence numbers for new rows. This is a MySQL-specific construct: other database systems use different mechanisms to generate sequence numbers.

`VARCHAR(200) represents a variable string of at most length 200.`
Other data types: SMALLINT, FLOAT, CHAR(N) (VARCHAR is NULL-terminated, CHAR takes up N bytes all the time)
Other type attributes: UNIQUE (no two rows can have the same value for this column)

`PRIMARY KEY(bank_id) makes "bank_id" the primary key of the table. A PRIMARY KEY is a unique index where all key columns must be defined as NOT NULL. A table can have only one  PRIMARY KEY.`
Another common constraint is FOREIGN KEY. A FOREIGN KEY in one table points to a PRIMARY KEY in another table. It is used to prevent actions that would destroy links between tables, as well as prevents that invalid data form being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

TODO: Create more tables using phpMyAdmin
Create a "people" table with 3 columns: "person_id ", "person_name" and "birthday". With the following constraint:

- All three columns shouldn't accept NULL values;
- "person_id" should be unsigned integer that increments automatically;
- "person_name" should be a variable string of length 200;
- "birthday" should be type "DATETIME"

Create a "accounts" table with columns: "account_id", "bank_id", "person_id" and "total".

- All columns should be unsigned integer that does not accept NULL values;

- “account_id” should be the primary key of this table and increment automatically;
- “total” should have default value: 0 (HINT: DEFAULT 0 serves the purpose);
- “bank_id” should reference “bank_id” in table “bank”
  Todo this, you add “FOREIGN KEY(bank_id) REFERENCES banks(bank_id),” to constraints.
- “person_id” should reference “person_id” in table “people”.

Part 3: More SQL: INSERT, SELECT, UPDATE and DELETE

INSERT:

Once you have the table created, insert data into it is fairly easy. All you need to do is to provide target table, columns, and values:

```
INSERT INTO banks (bank_name, bank_city)
VALUES (‘Bank Of America’, ‘Providence’);
```

Note: since the bank_id is set to be AUTO_INCREMENT, you don’t have to give it a value.

TODO:
1. Use SQL box in phpMyAdmin to insert at least 2 banks and 2 people into the “banks” table and “people” table respectively.

Now go to the Browse tab, navigate to the tables you just created, the data you inserted should be there. Pay specially attention to “bank_id” and “person_id”. Even though you didn’t enter the id explicitly, auto_increment did the job for you.

2. Insert at least 3 accounts in “accounts” table use the similar syntax. Make sure you use the “bank_id” and “person_id” as it is in “banks” and “people” table to fulfill the foreign key constraints. Note: If you don’t insert a total value, total will default to 0 (why?).

SELECT:

SELECT is the most often used SQL statement. It retrieves data from the database based on parameters specified by the user. The general syntax for SELECT is as follows:

```
SELECT column_name(s) FROM table_name [WHERE]
```

It is pretty straight forward and does exactly what it says. Some extra things you can do with simple SELECT … FROM:

- Use “DISTINCT” to eliminates duplicates
  ```
  SELECT DISTINCT bank_city FROM banks
  ```
- Use * to get all columns of a table.
  SELECT * FROM accounts

The WHERE clause is optional, it allows you to filter out rows that are returned. This is done via comparison of attributes or constants using operators =, ≠, <, >, ≤, and ≥. Comparisons can be joined using AND and OR. Other logical operators you can use: NOT, IN, etc.

Try these examples (You might want to insert more data to make the output interesting) :

- `SELECT bank_city FROM banks`
- `SELECT bank_name, bank_city FROM banks`
- `SELECT * FROM accounts WHERE bank_id = 1`
- `SELECT * FROM accounts WHERE total > 20 AND person_id < 3`
- `SELECT bank_name FROM banks WHERE bank_city IN (‘Providence’, ‘New York’)`

Aggregate Functions

To calculate and summarize data, you can use te aggregate functions: `SUM, AVG, COUNT, MIN, MAX and COUNT.`
Try these:

- `SELECT SUM(total) FROM accounts`
- `SELECT COUNT(account_id), AVG(total) FROM accounts`

Note: COUNT(expression) finds the number of not-NULL values in expression. But COUNT(*) returns the number of rows in the specified table including the null values, and without eliminating duplicates.

More about SELECT: output control

There are other optional clauses you can include after SELECT … FROM ... to make the output interesting and useful. They are all optional, but have to be included in the following order:

```
      SELECT column_name(s) FROM table_name [WHERE][GROUP BY] [HAVING] [ORDER
BY] [LIMIT]
```

- The GROUP BY clause allows you to perform aggregate operations on the returned data (for example, figuring out how many accounts can be found at each bank). A GROUP BY clause is just the words “GROUP BY” followed by a comma delineated list of one or more columns to group by.
- The HAVING clause is exactly the same as the WHERE clause, but it filters after a GROUP BY has been applied. This allows you to filter based on the result of aggregation.
- The ORDER BY clause allows you to change the order that rows appear in. An ORDER BY clause consists of the words “ORDER BY” followed by a comma delineated list of one or more columns. Each column name can be followed by the keyword ASC or DESC to specify whether the sort is done in ascending or descending order.
- The LIMIT clause allows you to restrict the total number of rows that are returned. LIMIT 25 says to return the first 25 rows: LIMIT 10, 15 says to return 15 rows after the first 10 rows.

TODO: Try the following examples and make sure you understand what they do.

- `SELECT * FROM accounts WHERE bank_id = 1 ORDER BY total`

- `SELECT * FROM accounts ORDER BY total LIMIT 3`
- `SELECT SUM(total) FROM accounts GROUP BY person_id`
- `SELECT person_id, SUM(total) FROM accounts GROUP BY person_id`
- `SELECT person_id, SUM(total) FROM accounts GROUP (compare the output with the previous query)`
- `SELECT bank_id, count(account_id) FROM account HAVING count(account_id) > 5`

JOIN the tables

Crucially (to your next assignment and otherwise), SQL can be used do queries on multiple tables simultaneously. We can do this by first joining the tables on a set of matching values, then we can run the previously described queries to manipulate the resulting table on its own. For example, both your 'accounts' table and your 'people' table have a column called 'person_id'. We've named them as such to make it clear that the values in the columns can be used to link a person to their account.

An **'inner join'** can be used to match up two tables with columns which refer to each other:

```
SELECT *
FROM people INNER JOIN accounts
ON accounts.person_id=people.person_id;
```

We've now created a new table where every row in 'accounts' and every row in 'people', where the value of the two 'person_id' columns are the same, are joined. Here's an illustration:

accounts:

account_id	bank_id	person_id	total
1	1	23	5000
2	1	66	1000
3	2	1	1900
4	2	30	6000
5	1	31	2000

people:

person_id	person_name	birthday
23	Jason Gorelick	1990-03-26 00:00:00
30	Neal Poole	1989-11-24 00:00:00
35	Adam Zethraeus	1990-02-06 00:00:00

result of inner join:

person_id	person_name	birthday	account_id	bank_id	total
23	Jason Gorelick	1990-03-26 00:00:00	1	1	5000
30	Neal Poole	1989-11-24 00:00:00	4	2	6000

Notice that the resulting table only contains the rows from each source table which have a match in the person_id columns. That's to say, because Adam's person_id (35) was not found in the accounts table, his information does not appear in a row in the resulting table.

You can use a '**left join**' to keep rows which don't have a match in your query. The syntax is predictably similar to an inner join, *but notice the order of the tables*. We'll illustrate by using the same example tables as in the previous example. Try to predict what the resulting table will look like.

```
SELECT *
FROM people LEFT JOIN accounts
ON accounts.person_id=people.person_id;
```

(accounts table and people table as above)

unnamed tables resulting from left join:

person_id	person_name	birthday	account_id	bank_id	total
23	Jason Gorelick	1990-03-26 00:00:00	1	1	5000
30	Neal Poole	1989-11-24 00:00:00	4	2	6000
35	Adam Zethraeus	1990-02-06 00:00:00	NULL	NULL	NULL

In the case of a left join, the order of the tables in the query matters; rows from the first are included in the output even if they don't have matching values in the columns used to join the tables. Note, columns that had been defined as NOT NULL in their source tables still ended up NULL in the result of the join when there was no row to join to.

The '**AS**' keyword can be used to rename columns when creating result tables. For example, the previous query could be written to rename the 'total' column to 'total_money' as such:

```
SELECT people.person_id,
            person_name,
            birthday,
            account_id,
            bank_id,
            total AS total_money
FROM people LEFT JOIN accounts ON accounts.person_id=people.person_id;
```

In this example, as we are naming each column we want to select, instead of selecting all, we must be explicit about which person_id column to display. We do so by prefixing the column name with the table name. If we were to do further manipulation of the resulting table, we would refer to the former ‘total’ column as ‘total_money’ - aside from that, as we manually selected all of the columns, the returned table is identical to the one returned by selecting ‘ * ’.

TODO:

- Populate your tables with data which will cause an inner join and a left join to return different results. Show a TA the sources, queries, and results.

Value NULL
If a column is not set to be NOT NULL, it will default to NULL if you don’t give it a value during insertion.
Special things about NULL:
- “xxx IS NULL” is used when doing comparisons with NULL. Note: “xxx=NULL” does not work.
- Anything op null = null (op is any arithmetic operation)

String Operation
LIKE is used for string matching expressions. SQL includes a string-matching operator
– “%” Matches any substring (including empty).
– “_” Match any one charactee
You can do the string operations like: UPPER(...), SUBSTRING(...) in both output and predicates.

Try the following examples

- `SELECT * FROM banks WHERE bank_name LIKE ‘%Bank%’`
- `SELECT bank_name FROM banks WHERE UPPER(bank_city) LIKE ‘PROVIDENC_’`
- `SELECT SUBSTRING(person_name, 1, 5) FROM people`

UPDATE & DELETE:

Similar to single-table SELECT statements.
Only can specify WHERE clause for DELETE:
```
DELETE FROM accounts WHERE total = 0
```
UPDATE must list what columns to update and their new values:
```
UPDATE accounts SET total = 100 WHERE person_id = 1
```

- Talk about these tables: what kind of relationship have we mapped? n-to-n.

This is a very brief intro to SQL database, if you are interested in this material, you should take CS127 next semester. Neal Pooooooooooooooole will be the HTA!!!!
http://www.cs.brown.edu/courses/cs132/staff_normal/nbpoole.png

Now make sure you understand what the following query does and alert a TA, be prepared to explain it.

```
SELECT AVG(total) AS avg_total, birthday
FROM people
      INNER JOIN accounts ON (accounts.person_id = people.person_id)
      INNER JOIN banks ON (banks.bank_id = accounts.bank_id)
WHERE UPPER(bank_name) = ‘BANK OF AMERICA’
GROUP BY birthday
HAVING avg_total > 5000
ORDER BY avg_total DESC
LIMIT 1, 10
```

Part 4: Using PHP with SQL (PDO)

Now that you’ve set up a database and put some data in it, it’s time to manipulate it with PHP. The best way of doing this is through PHP Data Objects (PDO; http://php.net/manual/en/book.pdo.php), which has many convenience functions and was designed with security in mind.

These are some PDO functions & snippets you’ll be using often:

Instantiating a PDO object:

```
try
{
        $dbh = new PDO('mysql:host=' . $_SERVER['db_host'] . ';dbname=' .
$_SERVER['db_name'], $_SERVER['db_username'], $_SERVER['db_password']);
        $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch (PDOException $e)
{
        print "Error!: " . htmlspecialchars($e->getMessage()) . "<br/>";
        exit;
}
```

This attaches to the MySQL database using $_SERVER variables defined in your .htaccess. It also sets up nice error handling, and makes sure everything works. Every database operation you’ll do from here on out will be a method of $dbh (or whatever you named it), the database handle.

Reading data from the database using a SELECT:

```
$query = $dbh->prepare('SELECT x,y,z FROM my_delicious_table WHERE x=:x’);
$query->execute(array(':x’ => ‘Hello!’));
$results = $query->fetchAll();
```

This snippet prepares a statement, executes it with a value, and fetches the results ($results is

now an array of arrays:
    $results[0][‘x’] or $results[0][0] gets the first selected value (x) from the first row of results (assuming there’s at least 1).

Sometimes it’s useful to print_r the results from the database.

There are two major advantages to using prepared statements in this way:
    1. If you execute the same query multiple times the query execution plan is cached.
    2. The prepared statement is free from SQL injection attacks which you might otherwise be subject to (if you constructed the query with string concatenation, for example).

Inserting or updating rows:

`$dbh->prepare` and `$query->execute` (described above) should be used when inserting or updating rows with user-defined values. For queries which don’t need to be executed with parameters you can use `$dbh->exec(QUERY)`, which will run the query and return the number of affected rows:

```
$dbh->exec('UPDATE my_delicious_table SET y=y+1’);
```

It’s frequently useful to have the ID of a recently inserted row, but if the ID column is autoincrement this is tricky (note that for an autoincrement ID field you shouldn’t supply a value in an INSERT, the database will choose it for you). Luckily PDO has a handy function to make this easy:

```
$new_id = $dbh->lastInsertId();
```

Part 5: Making an application

Now you’re going to tie everything together and make a little PHP application using the database you’ve set up. Here are the things your application should have:
    A form for creating a new account. This could come from either an existing customer (given their ID and their birthday) or a new customer (in which case you should collect all the info you’ll need to create a new person as well)
    A form for listing customer information. The form should take a customer name and list out all the accounts & associated information at different banks. The results should also be paginated, with an input controlling how many results to show per-page. This should involve both a JOIN and a LIMIT statement.

Support/example: adding a new bank --

```
<?php

$bank_name = isset($_POST[‘name’]) ? trim($_POST[‘name’]) : ‘’;
```

```
$bank_city = isset($_POST[‘city’]) ? trim($_POST[‘city’]) : ‘’;

if (empty($bank_name) || empty($bank_city))
{
      exit(‘Please fill in all fields’);
}

try
{
        $dbh = new PDO('mysql:host=' . $_SERVER['db_host'] . ';dbname=' .
$_SERVER['db_name'], $_SERVER['db_username'], $_SERVER['db_password']);
        $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
}
catch (PDOException $e)
{
        print "Error!: " . htmlspecialchars($e->getMessage()) . "<br/>";
        exit;
}

$query = $dbh->prepare('INSERT INTO banks (bank_name, bank_city) VALUES
(:name, :city)’);
$query->execute(array(':name’ => $bank_name, ‘:city’ => $bank_city));

$new_bank_id = $dbh->lastInsertId();
echo “New bank created with ID $new_bank_id”;
```