wrangling data

unix terminal and filesystem

Grab data-examples.zip from top of lecture 4 notes and upload to main directory on c9.io. (No need to unzip yet.)

Now go to bash and type the command ls to list files

The unix command unzip <filename> unzips a file.

```
devinbalkcom:~/workspace $ unzip data-examples.zip
Archive:  data-examples.zip
   creating: data-examples/
  inflating: data-examples/countdown.php
  inflating: data-examples/countdown.py
  inflating: data-examples/db
  inflating: data-examples/myusers
  inflating: data-examples/README-SQL.txt
  inflating: data-examples/submit.php
  inflating: data-examples/world.csv
  inflating: data-examples/world.sqlite3
```

```
devinbalkcom:~/workspace $ ls
README.md  data-examples/  data-examples.zip  hello-world.php  php.ini  provided-01.zip  test/
```

unix terminal and filesystem

cd <directoryname> changes the working directory.

```
devinbalkcom:~/workspace $ cd data-examples
devinbalkcom:~/workspace/data-examples $ ls
README-SQL.txt  countdown.php  countdown.py  db  myusers  submit.php  world.csv  world.sqlite3
devinbalkcom:~/workspace/data-examples $
```

side note. Why all the typing? Why are so many nerds using mac or linux? What is linux/unix anyway?

Ok, I'll be nice, use the mouse and c9.io interface to look at the contents of countdown.php.

unix terminal and filesystem

php <filename> runs php code.

```
devinbalkcom:~/workspace/data-examples $ php countdown.php
10
9
8
7
6
5
4
3
2
1
Blast off!
devinbalkcom:~/workspace/data-examples $
```

python <filename> runs python code.

```
devinbalkcom:~/workspace/data-examples $ python countdown.py
```

unix terminal and filesystem

command	what it does
`pwd`	print current directory
`ls`	list files in current directory
`cd dirname`	change to directory
`cd ..`	change to directory above this one
`mkdir dirname`	make a new directory
`touch filename`	make an empty directory
`mv filename1 filename2`	rename/ move a file
`cp filename1 filename2`	copy a file
`rm filename`	remove a file. (-r for directories)

Exercise: create a new directory **tempdir**, and a file **myfile.txt** within.

outline

1. encoding data for transmission or storage (bits)
2. encryption and compression
3. compound data: objects and records
4. storing data in a file system: csv files
5. the unix filesystem
6. storing data in a relational database
7. selecting data using relations (SQL query)

databases

firstName	lastName	email	phone
Devin	Balkcom	devin.balkcom@dartmouth.edu	6-0272
Hany	Farid	hany.farid@dartmouth.edu	6-2761

A **table** is a collection of **rows**. Each row can be thought of as a record: a collection of related information.

Each **column** contains a field: data of a particular type.

relational database query: A search for values over one or more columns returns a set of rows.

creating a database with sqlite3

To create a database, sqlite3 <filename>.
Create a new database db.sqlite3 in data-examples now.

```
devinbalkcom:~/workspace/data-examples $ pwd
/home/ubuntu/workspace/data-examples
devinbalkcom:~/workspace/data-examples $ sqlite3 db.sqlite3
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

Type:

```
CREATE TABLE professors (firstName text,
 lastName text, email text, phone text);
```

creating a database with sqlite3

```
CREATE TABLE professors (firstName text,
 lastName text, email text, phone text);
```

- Notice that a file was just created: db.sqlite3
- You have created a table professors with four columns
- Each column is of type text
- Commands in sqlite end with a semicolon
- By convention, commands are capitalized.

Type `.tables` to list tables. You should have one: professors.

```
Enter SQL statements terminated with a ";"
sqlite> .tables
sqlite> CREATE TABLE professors (firstName text, lastName text, email text, phone text);
sqlite> .tables
professors
sqlite>
```

creating a database with sqlite3

Add a row with `INSERT INTO <table_name> VALUES()`

```
INSERT INTO professors VALUES ('Devin',
'Balkcom', 'devin.balkcom@dartmouth.edu',
'6-0272');
```

Verify that it worked:

```
SELECT * FROM professors;
```

```
sqlite> SELECT * FROM professors
   ...> ;
Devin|Balkcom|devin.balkcom@dartmouth.edu|6-0272
sqlite>
```

Add Hany's info now:
Hany|Farid|hany.farid@dartmouth.edu|6-2761

.commands in sqlite3

command	what it does
`.help`	Lists the .commands
`.quit`	c ya!
`.tables`	list tables
`.schema tablename`	list column names, types for table
`.open filename`	opens the database in filename

Notice, no save command. Saves immediately!

sqlite3 data types

Each value in an SQLite3 database has one of the following *storage classes*.

- NULL. The value has no value
- INTEGER. The value is a signed integer
- REAL. The value is a floating point value
- TEXT. The value is a text string
- BLOB. The value is a blob of data, stored exactly as input (image or music data is not text).

```
CREATE TABLE professors (firstName text,
 lastName text, email text, phone text);
```

sqlite3 primary key

The **primary key** is a column that uniquely identifies each row.

Last name is not a good primary key. SSN is, but private.

```
CREATE TABLE professors2 (id integer PRIMARY KEY
AUTOINCREMENT, firstName text, lastName text, email text,
phone text);
```

To insert:

```
INSERT INTO professors2 VALUES (NULL, 'Devin',
'Balkcom', 'devin.balkcom@dartmouth.edu', '6-0272')
```

Add Hany to professors2 and select all.

a bigger example: world.sqlite3

```
sqlite> .open world.sqlite3
sqlite> .tables
world
sqlite>
```

(To see how world.sqlite3 was created, go to README-SQL.txt)

a bigger example: world.sqlite3

> *CAIN: I'm ready for the 'gotcha' questions and they're already starting to come. And* ***when they ask me who is the president of Ubeki-beki-beki-beki-stan-stan*** *I'm going to say, you know,* ***I don't know. Do you know?***

Let's see what we've got:

```
sqlite> .schema
CREATE TABLE world (country text, abbrv text, gdp real, population real);
sqlite>
```

To get the whole table:

```
SELECT * FROM world;
```

To select a few columns

```
SELECT country, gdp FROM world;
```

```
Uzbekistan|62643953022.0
Vanuatu|814954307.0
Vietnam|186205000000.0
West Bank and Gaza|12737613125.0
Zambia|27066230009.0
Zimbabwe|14196912535.0
sqlite>
```

selecting rows with WHERE

```
SELECT column FROM table WHERE condition;
```

Example:

```
SELECT population FROM world
 WHERE country = FRANCE;
```

Multiple columns and rows:

```
SELECT country, gdp FROM world
 WHERE population < 500000;
```

conditions in SQL

```
SELECT column FROM table WHERE condition;
```

We can use BETWEEN x AND y to narrow further:

```
SELECT country, gdp FROM world WHERE
  population BETWEEN 500000 AND 1000000;
```

Logical operators AND and OR work:

```
SELECT country FROM world WHERE
  population < 500000 AND gdp > 10000000;
```

Exercise: Tuvalu

Write an SQLite command that searches the table world for all countries with a gdp between 10 and 100 million. Your search should return the matching countries' 3-letter abbreviation.

```
SELECT abbrv FROM world WHERE gdp BETWEEN
  10000000 AND 100000000;
```

Nesting searches

Extract names of countries with population larger than France's:

First way: find the population of France with search (66206930), and then use that number (not nested):

```
SELECT country FROM world WHERE population >
66206930;
```

Second way: nest the searches

```
SELECT country FROM world WHERE population >
(SELECT population FROM world WHERE
country='France');
```

Aggregate (calculations on search results)

```
SELECT <command> FROM <table>;
```

Command is usually of the form FUNCTION(column).

```
SELECT SUM(population) FROM world;

SELECT MAX(gdp) FROM world;

SELECT COUNT(*) FROM world;
```

Exercise: wealthy countries

Write a nested search that lists the names and gdp's of all countries with above-average gdp. (Hint -- try something simple first, like computing average gdp.)

```
SELECT country,gdp FROM world WHERE gdp >
(SELECT AVG(gdp) FROM world);
```

```
sqlite> SELECT country,gdp FROM world WHERE gdp > (SELECT AVG(gdp) FROM world);
Arab World|2845790000000.0
Australia|1454680000000.0
Brazil|2346080000000.0
Canada|1785390000000.0
Central Europe and the Baltics|1457320000000.0
Euro area|13410200000000.0
European Union|18514200000000.0
France|2829190000000.0
Germany|3868290000000.0
Indonesia|888538000000.0
Italy|2141160000000.0
Japan|4601460000000.0
Korea|1410380000000.0
Mexico|1294690000000.0
Netherlands|879319000000.0
North America|19210100000000.0
Russian Federation|1860600000000.0
Spain|1381340000000.0
United Kingdom|2988890000000.0
United States|17419000000000.0
sqlite>
```

LIKE (loose matches with wildcards)

We can use =, >, < to filter rows.
We can use LIKE to filter text based on wildcards.

```
SELECT country FROM world
  WHERE country LIKE "F%";
```

```
sqlite> select country from world where country like "F%";
Finland
France
sqlite>
```

mini-exercise: which countries contain ‘ee’?

```
sqlite> select country from world where country like "%EE%";
Greece
sqlite>
```

deleting rows with DELETE

DELETE works the same as SELECT, but destroys the rows.

```
DELETE FROM world WHERE country="France";
DELETE FROM world WHERE country LIKE "S%";
```

You can delete the whole table with DROP TABLE

```
DROP TABLE <table>
```

As Yoda would say, "Drop or drop not -- **there is no undo!**"

a bigger example: northwind.sqlite3

Grab northwind.sqlite3 from lecture 5 notes, and
upload to data-examples on c9.io.
cd to data-examples, and open the database with sqlite3.

```
.tables
.schema
```

```
sqlite> .tables
Alphabetical list of products   Orders
Categories                      Orders Qry
Current Product List            Products
Customer and Suppliers by City  Products Above Average Price
CustomerCustomerDemo            Products by Category
CustomerDemographics            Region
Customers                       Shippers
EmployeeTerritories             Summary of Sales by Quarter
Employees                       Summary of Sales by Year
Order Details                   Suppliers
Order Details Extended          Territories
Order Subtotals
sqlite>
```

exercise: counting customers

Display all of the rows of the table Customers and write an sql command that counts the total number of rows.

```
sqlite> SELECT COUNT(*) FROM CUSTOMERS;
91
sqlite>
```

CUSTOMERS should be lower-case, right?

selecting unique values with DISTINCT

```
SELECT City from Customers;
```

will gives some repeat cities. Try:

```
SELECT DISTINCT city FROM customers
```

mini-exercise: how many cities do customers live in?

```
sqlite> SELECT COUNT(DISTINCT city) FROM customers;
69
```

ordering results with ORDER BY

```
SELECT * FROM Customers ORDER BY Country ASC;
SELECT * FROM Customers ORDER BY Country DESC;

SELECT * FROM Customers
  ORDER BY Country ASC, City ASC;
```

```
sqlite> SELECT ContactName, Country, Phone FROM Customers ORDER BY Country ASC;
Patricio Simpson|Argentina|(1) 135-5555
Yvonne Moncada|Argentina|(1) 135-5333
Sergio Gutirez|Argentina|(1) 123-5555
Roland Mendel|Austria|7675-3425
Georg Pipps|Austria|6562-9722
Catherine Dewey|Belgium|(02) 201 24 67
Pascale Cartrain|Belgium|(071) 23 67 22 20
Pedro Afonso|Brazil|(11) 555-7647
Aria Cruz|Brazil|(11) 555-9857
AndrFonseca|Brazil|(11) 555-9482
Mario Pontes|Brazil|(21) 555-0091
```

other useful commands

Read about in lecture 5 notes:

UPDATE
DATE
JOIN

php, meet sql

evolution of HTTP communication

Client: gimme a page.
Server: ok (html)

C: gimme a page.
S: let me build you one based on my current data (php)

C: here's some data. Gimme a page.
S: building you a page based on our data.

C: here's some data. Gimme a page.
S: building you a page based on our data. I'll save your data in a file.

C: here's some data. Gimme a page.
S: building you a page based on our data. I'll save your data in a database.

anatomy of a modern web application

1. **Client**: requests an html page from the server.
2. **Server**: apache sends html or creates (php, python + django or flask, ruby + rails, java + ?, node.js + express)
3. **Client**: browser renders that page and requests additional resources from various servers (images, css, external javascript).
4. **Client**: user clicks or enters data (callback functions).
5. **Client**: javascript reacts (DOM manipulation, react.js, backbone, angular) and/or a request is sent to the server (ajax or form).
6. **Server** records data received (php; file or sql/nosql database, can be processed later with Python, etc); go back to step 2.

anatomy of a modern web application

1. **Client**: requests an html page from the server.
2. **Server**: apache sends html or creates (php, python + django or flask, ruby + rails, java + ?, node.js + express)
3. **Client**: browser renders that page and requests additional resources from various servers (images, css, external javascript).
4. **Client**: user clicks or enters data (callback functions).
5. **Client**: javascript reacts (DOM manipulation, react.js, backbone, angular) and/or a request is sent to the server (ajax or form).
6. **Server** records data received (php; file or sql/nosql database, can be processed later with Python, etc); go back to step 2.

the form (example_form-submit.html)

```
<b>Sign up to be notified:</b>
<form action="submit.php" method="POST">
  name: <input type="text" name="name"> <br>
  email: <input type="text" name="email"> <br>
  <button type="submit">sign up</button>
</form>
```

what happens when ‘sign up’ is clicked?

1. Browser creates (and encrypts?) a ‘post request’ and sends to server:

```
POST submit.php HTTP/1.1
Host: www.cs.dartmouth.edu
name=devin&email=devin@cs.dartmouth.edu
```

2. Server runs submit.php script with $_POST dictionary containing name, value pairs, and sends back the results of the script as html.

Today, we want to write submit.php to take $_POST dictionary and save values as a row in a database table.

what should go in our database?

1. From client, we have name and e-mail.
2. From server, we have date (or we could let sqlite3 do it).
3. We could use a primary key. Make sqlite3 do it.

but first, we need to create a database

On c9.io:

```
sqlite3 myusers.sqlite3
sqlite> CREATE TABLE users (id integer
PRIMARY KEY AUTOINCREMENT, name text, email
text, date text);
sqlite> .exit
```

what does submit.php look like?

First, grab the name and e-mail from the $_POST dictionary:

```
$name  = $_POST["name"];
$email = $_POST["email"];
```

Then grab the date using php:

```
$date  = date("F j, Y, g:i a");
```

accessing the database from php

```
$name  = $_POST["name"];
$email = $_POST["email"];
$date  = date("F j, Y, g:i a”);

$db    = new SQLite3(‘myusers.sqlite3');

$db->query( 'INSERT INTO users VALUES( NULL,"' . $name
. '", "' . $email . '", "' . $date . '")' );
```

The extra quotes are needed to build a string that contains strings. (Note single vs. double quotes.) We want to build something like:

```
INSERT INTO users VALUES( NULL,"Hany Farid",
"farid@cs.dartmouth.edu", "March 3, 2016, 7:51 pm")
```

(You could clean up with string substitution tricks.)

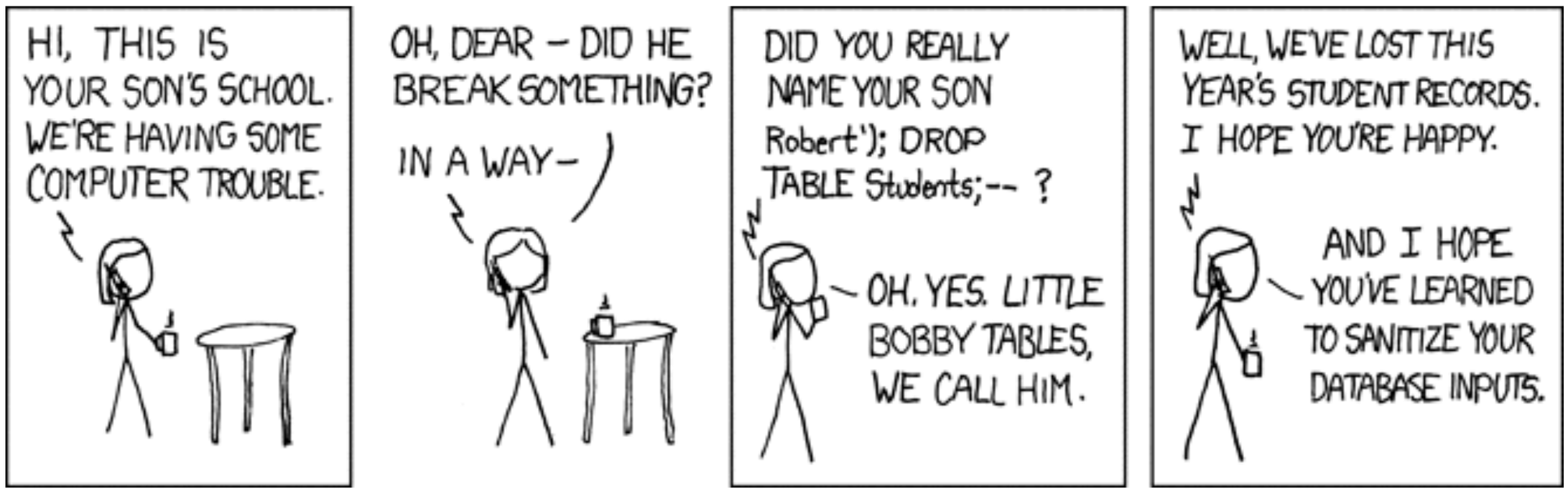
fulfilling the http request

The client asked for an html page.

```
$name  = $_POST["name"];
$email = $_POST["email"];
$date  = date("F j, Y, g:i a”);

$db    = new SQLite3(‘myusers.sqlite3');

$db->query( 'INSERT INTO users VALUES( NULL,"' . $name
. '", "' . $email . '", "' . $date . '")' );

print $name . ", thanks for signing up on " . $date;
```

security note: code injection attacks

The php script is building a command for sql to execute. Clever choices of values could look like code: a code injection attack.



discussion

1. strengths and weaknesses of relational databases: flexible query language. Costs of flexibility: security, run-time, rigidity of row structure. nosql vs sql.
2. server-side frameworks (python + django/flask, ruby on rails, js + node.js/ express, java + ?, scala + lift, …)
3. client-side frameworks with ajax (react.js, backbone, angular)