Lecture 5: Another Simple Example

- There is also a nice example in the text – let's talk about this one a bit
- Four tables:
    - Corvettes – stores basic information about the cars
    - States – associates an id with each state name
    - Equipment – associates an id with each equipment value
    - Corvettes_Equipment – "helper" table to connect Corvettes to Equipment
        - Corvettes.Vette_id is one to many with Corvettes_Equipment. Equip
- See text files

Lecture 5: Another Simple Example

- Corvettes_Equipment.Vette_id is many to one with Equipment.Equip_id
  - Together they effectively make Corvettes.Vette_id and Equipment.Equip_id many to many
    - States is not really necessary
      - > Just "clean" up Corvettes table
  - The setup.php script creates these tables
    - Look at it (see comments especially)

- Once these tables are set up, we need to extract information via queries
  - These often involve a "join" of two or more tables
    - Relating data in one table to data in another to get a new set of entities
- Relational databases allow for many variations of joins:
  - Ex: INNER JOIN, CROSS JOIN, NATURAL JOIN, LEFT JOIN, RIGHT JOIN
  - We will focus on just a few of the possibilities

- **INNER JOIN** (or cartesian product, or CROSS JOIN)
  - Associates each row in one table with each row in the other
    - > Note: Technically, an INNER JOIN is not equivalent to a CROSS JOIN – the INNER JOIN requires the tables to have some matching data -- however, in MySQL they are equivalent
  - If table T_1 had R_1 rows and table T_2 had R_2 rows, the cross join will have $R_1 x R_2$ rows
    - For example, a join of the Corvettes and States tables
  - This is the default join in MySQL, and syntactically can be done in several different ways

- Implicit vs explicit
- Explicit:
    - SELECT * FROM Corvettes INNER JOIN States
    - SELECT * FROM Corvettes JOIN States
- Implicit
    - SELECT * FROM Corvettes, States
- Clearly, in most cases this is not what we really want
    - Rather, we want certain rows in one table that relate to certain rows in the other table in a certain way

- We can achieve this with an **equi-join**
  - In an equi-join, we add equality testing conditions to the join that only some of the rows in the full join table meet
    - For example in the Corvettes and States tables
    - We can either match fields in one table to those in another or we can match fields in either table to a specific value. For example:

SELECT * FROM Corvettes, States WHERE Corvettes. State = States.State_id

SELECT * FROM Corvettes, States WHERE Corvettes. State = States.State_id AND States.State = 'California '

    - We can also use the other syntax variations

- Technically, an equi-join requires all conditions to be equality tests
  - Allows for some efficiency improvements
- However, the restrictions given in the WHERE clause can be arbitrary
  - Any relational expression that we'd like to use when comparing data in the tables
- The more restrictions we have, the fewer records will be returned
  - Note, however, that having fewer returned records does not necessarily imply that the query took less time to be done!

- Generally a MySQL join is specified in the select command (this version is somewhat simplified):

SELECT *items* FROM *tables*

[WHERE condition]

[GROUP BY group_type]

[HAVING where_definition]

[ORDER BY order_type]

[LIMIT limit criteria]

  - [] values are optional
  - More than one table gives us a join
  - The WHERE condition restricts the join

- We do these queries using MySQL commands
  - Our access can be very general if we'd like
    - See carsdata.html and access_cars.php
    - BE CAREFUL with these files on your accounts
      - > Allow user to make arbitrary changes to DB
      - > This type of script would not be available on a production server
  - Let's try a few queries:
    - Find all cars since 1995 (inclusive) that have automatic transmission, also showing their state
    - Find out how many cars are from each state in the list
    - Find all cars from California, sorted by mileage

- More often the access is restricted, based on the needs of the site that we are developing
    - User of the site doesn't even know that MySQL is being used
    - Simply enters information into a form as we'd expect and the processing of the form is done via a MySQL database
    - Queries are designed based on what information is needed
    - Access is similar in nature to that used to set up DB

- For example:
- User logs into a website by typing id and password
    - Ids and passwords are stored in a DB and a query is done to determine if the user is valid
    - New users are added to the DB
- Let's talk a bit about this from a security standpoint
    - Assume DB X is storing ids and passwords for System Y
    - Assume DB X becomes compromised

- We'd like this to NOT also compromise System Y
- How can we handle this?
- Rather than storing passwords in plaintext, we instead store a **hash**ed version of the password
    - The function hash() allows several different hash algorithms to be used
- These are **one way mappings of the password** using sophisticated hashing techniques
    - Since they are one way, we cannot (easily) reproduce the password from the hashed versions
- But don't we need the actual password?
    - Not usually

- Ex:
- When logging into System Y user types password, P'
- Hashed version of actual password P, hash(P) is stored in the DB
- System Y retrieves hash(P) from the DB and compares it with hash(P')
- If hash(P) == hash(P’), the password is valid; otherwise it is not
- Note that obtaining hash(P) by breaking into the DB will not give away actual passwords
- Since login system requires user to enter P, not hash(P)
- Q: How can we tell if hashed passwords are being used on a given system?

- Let's look at a detailed example:
    - setquotes.php, ex17.php, ex17b.php, ex17c.php
- Script ex17.php is the "initial script"
    - Session variables are used to prevent users from going directly to ex17b.php or ex17c.php
    - Idea is that access is granted each step of the way and if you start in the middle it will be detected
- Take a look at the files and read the comments very carefully
    - There is a lot of information in there!

- We have mentioned security a few times now
- The most important thing to realize as a Web developer / administrator is that hackers are always trying / finding new / unusual ways to attack you
- Ex: SQL Injection (http://en.wikipedia.org/wiki/SQL_injection)
    - One common use of a DB is to search based on input from the user
        - Ex: Enter name of person you wish to find
    - A dangerous assumption is that the user will always enter a legitimate search string

- SQL syntax is such the user can actually enter SQL code into a search string thereby circumventing the original intent of the query
    - In some cases, the user can actually be destructive, changing the DB itself
- Ex: Consider the following MySQL query in a PHP script:

    $query = "select * from Movies where Movie_name = '$movie'";

    - Assume $movie was extracted from a POST submission
    - The intent is that the user will enter a movie name
    - What if the user enters SQL syntax?
        - Let's try it!

- In some systems the problem can be much worse, due to allowance of cascading SQL statements in a query
    - Ex user input:

    ' or '1'; drop table Movies; select * from Movies where '1' = '1

    - Oh no!
        - Note that this depends on the user knowing the table name – but it can often be guessed
        - Also it depends on system allowing multiple statements in a query – MySQL does not but some do
    - See ex18.php