Databases & SQL

Chapter 13

1. Introduction
2. Database Terminology
3. Creating a Database
   - Conceptual Design
   - Logical Design
   - Normalization
   - Physical Design
4. MySQL
   - Retrieving Data
   - Modifying Data
   - Joins
5. MySQL & PHP

- Storing data in flat files is fine for very small applications.
- Most applications require a better way to manage the data.
- This is not a database course. Take CS361 for much more comprehensive coverage.
- Our interest in databases is setting up a simple database and connecting to it from our web pages.

- **Database** - structured collection of data. Most common type is a **relational database**.
- Relational Database - data is stored in **tables** consisting of rows and columns. Relations connect pieces of data.
- **Tuple** - a data item or object, also called a row or record.
- **Field** - an attribute of an object, also called a column.
- We will be using MySQL in this course. It is already installed on the web server (LA**M**P).
- A database contains tables, tables contain tuples.

- Large databases can contain millions and millions of records.
- It isn't feasible to retreive all records and loop through them all looking for some specific piece of data.
- A **query** is sent to the database to retrieve all records that match some criteria.
    - Give me all players who played in 2010.
    - Give me the top five pitchers by ERA in 2011.
    - Give me the top five home run hitters of 2010 who played for Milwaukee.

Motivation

- Databases provide the ability to ask more complex questions that would be harder to program.
- Databases are created to be fast. You do not have to keep data sorted nor do you have to do any searching.
- Databases protect the programmer against errors. Imagine writing a program to transfer $1,000 from one account to another. Let's say the source account is debited first. Before the destination account is credited, the program crashes. Databases provide **transactions**.
- Databases allow for multiple read/write accesses at once.

SQL

1. **SQL**, or Structured Query Language, provides a standard way to interact with relational databases.
2. SQL is a **declarative language**. Java, C++, JavaScript, PHP are all **imperative languages**.
3. We will be mostly interested in queries and data manipulation.
4. SQL also provides transactions, data definition and data control.
5. SQL is just a sequence of statements.

- **Database Design** occurs in three stages.
  1. **Conceptual Design** - determine what data needs to be stored in your database and how the data relates to other data.
  2. **Logical Design** - determine how to logically structure your data into tables and how those tables should be structured. Includes primary and foreign keys and avoiding redundant data.
  3. **Physical Design** - the final step is actually creating the database by choosing appropriate data type.
- Once your database is designed, we use SQL's data definition language to create the database and tables.

- **Database Schema** - description of the tables in a database and how they relate to one another.
- A schema contains:
  - names of all of the tables.
  - columns belonging to each table.
  - the data type of each column.
  - whether or not a column can be NULL.
  - primary key(s) of each table.
  - any foreign keys referencing other tables.

Example

- We want to design a database schema that stores current information about different countries in the world.
- For each country, we want the name, some unique three letter code, the surface area and continent.
- We also want to know population, average income, life expectancy.
- The average income comes from GNP and population.
- We want to know the languages spoken, what percentage speaks it and whether it is official or not
- Finally, we want information about major cities: name, region, population, whether or not it's the capital.

- Our first step is to identify entities, attributes and relationships.
- **Entity** - person, place, event or concept. An entity will map to a table.
- **Attribute** - characteristic of an entity. An attribute will map to columns of a table.
- **Relationship** - how two entities are connected. Relationships are represented by columns of a table or an entirely new table.
- There is generally no <u>right</u> answer but some answers are better than others.
- One common mistake to avoid is putting too much data in one table. Try to avoid repeated data.

Example

What should be our entities, attributes and relationships?
Entity-Relationship (ER) diagrams are often used to represent the conceptual view of a database.

- We want to design a database schema that stores current information about different countries in the world.
- For each country, we want the name, some unique three letter code, the surface area and continent.
- We also want to know population, average income, life expectancy.
- The average income comes from GNP and population.
- We want to know the languages spoken, what percentage speaks it and whether it is official or not
- Finally, we want information about major cities: name, region, population, whether or not it's the capital.

- Once we have our conceptual design, we start creating our schema. We first map the entities and attributes.
- **Table**: Countries.
  Columns: name, code, area, continent, gnp, population, life_expectancy, gov, leader.
- **Table**: Languages.
  Columns: name.
- **Table**: Cities.
  Columns: name, region, population.

- Once we have our initial schema, we determine the primary keys. A key is a unique column(s) for all tuples in a table.
- **Table**: Countries.
  Columns: name, code, area, continent, gnp, population, life_expectancy, gov, leader.
- **Table**: Languages.
  Columns: name.
- **Table**: Cities.
  Columns: name, region, population.

- Determining relationships is generally the hardest part of the mapping process.
- If entity X is related to only one other entity Y, then simply add a column (or columns) to X.
  - **Table**: Countries.
    Columns: name, code, area, continent, gnp, population, life_expectancy, gov, leader, capital_name, capital_region.
  - **Table**: Languages.
    Columns: name.
  - **Table**: Cities.
    Columns: name, region, population, country_code.

- A “many-to-many” relationship is best solved by using a separate mapping table. For example, a country can speak many languages and a language can be spoken in many countries.
  - **Table**: Countries.
    Columns: name, code, area, continent, gnp, population, life_expectancy, gov, leader, capital_name, capital_region.
  - **Table**: Languages.
    Columns: name.
  - **Table**: Cities.
    Columns: name, region, population, country_code.
  - **Table**: CountryLanguages.
    Columns: country_code, language, official, percentage.

- Lastly, we want to get rid of useless data or confusing data.
  - **Table**: Countries.
    Columns: name, code, area, continent, gnp, population, life_expectancy, gov, leader, capital_name, capital_region.
  - ~~**Table**: Languages~~.
    ~~**Columns**: name~~.
  - **Table**: Cities.
    Columns: name, region, population, country_code.
  - **Table**: CountryLanguages.
    Columns: country_code, language, official, percentage.

- Lastly, we want to get rid of useless data or confusing data.
  - **Table**: Countries.
    Columns: name, code, area, continent, gnp, population, life_expectancy, gov, leader, capital_name, capital_region.
  - **Table**: Cities.
    Columns: id, name, region, population, country_code.
  - **Table**: CountryLanguages.
    Columns: country_code, language, official, percentage.

- Once we have our tables setup, the way to minimize duplicate data and inconsistent data is to use a technique called **normalization**.
- Normalization is splitting up tables to improve structure and remove redundancies.
- As of now, there are 9 levels of normal form. We will briefly look at the first 3 levels.

First Normal Form

- A table is in first normal form if there are no repeated rows and there are no multi-valued columns.
- Columns are not meant to store lists. Only 1 data item should be in each column.
- Rows must also be unique. If two rows are identical, then why have both of them?

Second Normal Form

- A table is in second normal form if it is in first normal form and the primary key determines all non-key columns.
- In other words, if I know the code for some country, then I should be able to determine all other columns in that tuple. If I can't, then it isn't a primary key.

Third Normal Form

- A table is in third normal form if it is in second normal form and all columns are directly dependent on the primary key.
- For example, consider a table storing a person with attributes of a name and all details of an address (street number, street name, city, state, zip). The city and state don't really depend on the person key. It is probably best to move city, state and zip to a new table and only store the zip in the person table.

- Once we have our tables setup, we need to choose data types for each column.
- Datatypes are unfortunately RDBMS specific.
- MySQL allows for: INT, BIGINT, FLOAT, DATE, DATETIME, CHAR, VARCHAR, BLOB, TEXT, ENUM among many others. See http://www.w3schools.com/sql/sql_datatypes.asp
  for more details (be sure to look at the MySQL part).

- Let's consider our Cities table.
- **Table**: Cities.
  Columns: id, name, region, population, country_code.
- We must determine the types of each and how big each will be.
- A full explanation on how to create a table is at http://dev.mysql.com/doc/refman/5.7/en/create-table.html

- First login to the web server and start mysql.
- Your database should already be created for you. It is just your username.
- To use the database, type in: USE username;
- Once the database is selected, then we can create our tables.

```
CREATE TABLE ‘Cities‘ (
    ‘id‘ int(11) NOT NULL auto_increment,
    ‘name‘ varchar(40) NOT NULL default ”,
    ‘country_code‘ varchar(3) NOT NULL default ”,
    ‘region‘ varchar(20) NOT NULL default ”,
    ‘population‘ int(11) NOT NULL default 0,
    PRIMARY KEY (‘id‘)
);
```

- If you prefer a GUI, you should log into http://webdev.cs.uwosh.edu/phpmyadmin/
- It is very self-explanatory and easy to use.
- It is also a good place to go to test out your SQL commands to make sure you are creating them correctly.

- Before we can use PHP to connect to our database, we need to understand basic SQL commands.
- Before we can use basic SQL commands, we need to create our database.
- Everyone should have access to MySQL on the webdev.cs.uwosh.edu server.
  1. Connect to the server using SSH.
  2. Start mysql -p.
  3. Enter in your mysql password.
- Several key database statements include: SHOW DATABASES; USE database; SHOW TABLES; DESCRIBE table;
- Keywords do not need to be capitalized but most programmers use this convention. Each SQL command much end with a semi-colon.

- The **SELECT** statement is a way to retrieve information from a specified table and returns the result in another table. The basic syntax of SELECT is:
  SELECT col1, col2, . . . , coln FROM someTable;
- SELECT * FROM someTable; means select all columns from that table.
- Be careful, **database and table names are case-sensitive**.

Example

Let's say we want to get all teams in our database. First we want to know what the table looks like:

```
DESCRIBE TeamsFranchises;
```

Once we know the fields and types, we can make our query:

```
SELECT franchName, active FROM TeamFranchises;
```

- The **DISTINCT** modifier prevents duplicates in the returned table.
- SELECT franchName FROM TeamFranchises; returns 120 results.
- SELECT DISTINCT franchName FROM TeamFranchises; returns 99 results.

- Our queries, up until this point, are returning all rows in the table.
- We almost always want to filter our results. What if we want a list of currently active teams?
- The **WHERE** clause is made up of some number of **predicates** that specify conditions to limit which rows are returned.
- http://dev.mysql.com/doc/refman/5.0/en/non-typed-operators.html contains a complete list of MySQL operators.

Example

Let's say we want to get all active teams in our database. First we want to know what the table looks like:

SELECT franchName FROM TeamFranchises WHERE active="Y";

What does this do?

SELECT playerID FROM Salaries WHERE yearID=2012 AND salary>23000000;

- **BETWEEN** is used to check if a value is within some range of values. For example: SELECT playerID FROM Salaries WHERE yearID=2012 and salaray BETWEEN 20000000 AND 25000000;
- **IN** is used to check if a value is in some set of values. For example: SELECT playerID FROM Salaries WHERE year IN (2011, 1994, 1997);
- **LIKE** is used to filter strings. LIKE is not case-sensitive.
  - LIKE 'text%' - starts with text
  - LIKE '%text' - ends with text
  - LIKE '%text%' - contains text
  - LIKE '___' - matches exactly 3 characters

  SELECT playerID from Salaries WHERE playerID LIKE 'jones%';
  SELECT nameFirst, nameLast FROM Master WHERE nameFirst LIKE '__';

- The database will always return exactly what you want but it may not be in the order you want.
- **ORDER BY** can order the tuples in ascending or descending order.
  SELECT playerID, salary FROM Salaries WHERE yearID=2012 AND salary>23000000 ORDER BY salary DESC;
- By default, ORDER BY is ascending order. You can order by multiple columns if the first column is the same:
  SELECT nameFirst, nameLast FROM Master ORDER BY nameLast, nameFirst DESC;
- This orders them by nameLast in ascending order first. If there are ties, it orders them in descending order by nameFirst.

- Sometimes you only want the top 3 or top 10.
- You can use **LIMIT** to accomplish this.
  SELECT playerID, salary FROM Salaries WHERE yearID=2012 ORDER BY salary LIMIT 5;

- There are special **aggregate functions** built into SQL that perform summary computations.
- Functions include: AVG, COUNT, MAX, MIN, STD (STDDEV), SUM, VARIANCE.
  SELECT AVG(salary) FROM Salaries WHERE yearID=2012;
  SELECT COUNT(*) FROM Salaries WHERE yearID=2012 AND salary>15000000;
- **GROUP BY** modifies the behavior of aggregate functions. For instance, the following prints out the average salary of all players from 1990 to 1999:
  SELECT yearID, avg(salary) FROM Salaries WHERE yearID BETWEEN 1990 AND 1999 GROUP BY yearID;

- Consider the last query:
  SELECT yearID, avg(salary) FROM Salaries WHERE yearID BETWEEN 1990 AND 1999 GROUP BY yearID;
- How could we sort them by the average salary? Something like this:
  SELECT yearID, avg(salary) FROM Salaries WHERE yearID BETWEEN 1990 AND 1999 GROUP BY yearID ORDER BY avg(salary);
- It works, but it's a bit ugly. Instead, we can give avg(salary) a name so we can reuse it later:
  SELECT yearID, avg(salary) average FROM Salaries WHERE yearID BETWEEN 1990 AND 1999 GROUP BY yearID ORDER BY average;

- Unfortunately, we can't filter out aggregate columns in the WHERE clause. Let's say we only want averages above 1 million, we might try something like this:
  SELECT yearID, avg(salary) average FROM Salaries WHERE average>1000000 AND yearID BETWEEN 1990 AND 1999 GROUP BY yearID ORDER BY average;
- This error comes up: Unknown column 'average' in 'where clause'
- **HAVING** is the clause that solves this problem:
  SELECT yearID, avg(salary) average FROM Salaries WHERE yearID BETWEEN 1990 AND 1999 GROUP BY yearID HAVING average>1000000 ORDER BY average;

- Now that we know the clauses, the order of the query is important.
- The accepted order for a query is this: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY/LIMIT.

- SQL can be used to insert, delete and update records.
- **INSERT** is used to add tuples to the database. Syntax looks like:

  INSERT INTO someTable [(col1, col2, . . . , coln)]
  VALUES (value1, value2, . . . , valuen);
- If columns are omitted, the values are inserted in order.

  INSERT INTO Salaries
  VALUES ('1984', 'MIL', 'NL', 'glavito02', 7000000);
- The database will not allow you to insert if the primary key already exists. For example, if we already ran the above statement, the following statement would fail:

  INSERT INTO Salaries
  VALUES ('1984', 'MIL', 'NL', 'glavito02', 3000000);

- You may have a primary key that is numeric and doesn't really mean anything.
- If a tuple is inserted, it is just assigned the next number.
- We could solve this by selecting the maximum value from the primary key field and increment it.
- A better option is to set the column to auto_increment, in other words: CREATE TABLE ‘Cities‘ (‘id‘ int(11) NOT NULL auto_increment, . . .
- When adding to the Cities table, we do this:
  INSERT INTO Cities VALUES (”, ’Oshkosh’, ’USA’, ’WI’, 65000);

- If a field needs to be updated, be very careful to update the correct field and the correct tuple(s).
- The WHERE clause is optional but it is rarely omitted.
- Let's assume the id for Oshkosh is 300. If we wanted to update the population, we would write:
  ```
  UPDATE Cities SET population=67000 WHERE id=300;
  ```
- Deletion is very similar. WHERE is optional but is rarely omitted. If we want to delete Oshkosh:
  ```
  DELETE FROM Cities WHERE id=300;
  ```
- Be very careful with UPDATE and DELETE as they are permanent. There are no confirmation boxes nor are there undo's.

- Since we normalized our database, many of the questions we wish to answer are not available by a simple SELECT query using 1 table.
- Considering our baseball database, how do we find out the names of the top 5 paid players of 2010?

- **JOIN** connects two or more tables into a larger combined table.
- A join is a subset of the **Cartesion Product** of the tables. Records are connected if they meet certain conditions, specified by **ON**.
- If we want to see the name of each player along with the salary:
  SELECT * FROM Salaries JOIN Master ON Salaries.playerID = Master.playerID;
- We can also use the WHERE clause to filter it down:
  SELECT * FROM Salaries JOIN Master ON Salaries.playerID = Master.playerID WHERE teamID = 'MIL' AND yearID = 2010;
- Why didn't I need to say Salaries.yearID?

- The syntax for joins can get quite large:
  SELECT * FROM Salaries JOIN Master ON Salaries.playerID = Master.playerID WHERE teamID = 'MIL' AND yearID = 2010;
- One way to shorten them is to give names to the tables:
  SELECT * FROM Salaries s JOIN Master m ON s.playerID = m.playerID WHERE teamID = 'MIL' AND yearID = 2010;

- Return the names of all of the players born on New Years Eve.
- Return the name and maximum salary from all players born in Wisconsin.
- Return the top 3 earners of all time, display their names and salaries.
- Return the top five intentionally walked players in their career.
- Return the top 10 people with the highest batting average over their career. In other words, the number of hits divided by the number of at bats. To filter our data more, we are looking for people who debuted after January 1, 1980 and had at least 100 at bats.

- Older versions of PHP had functions that works for specific databases like *mysql_connect* and *pg_query*. These should be avoided.
- **PDO** is short for PHP Data Objects and allows us to connect to almost any database.
- To query a database we must:
  1. Construct a new PDO object with parameters representing the server and database.
  2. Call the PDO's query method which accepts a string representing the query.
  3. Examine and process the rows of results returned by the query.
- PDO has a number of methods. We will only use a few, the rest can be found at http://php.net/manual/en/class.pdo.php

```
<ul>
<?php
    //always connect to localhost
    $db = new PDO("mysql:dbname=krohns;host=localhost",
                  "username", "password");
    //note, no ; after 'WI' below
    //also note the single quotes surrounding a string
    $rows = $db->query("SELECT * FROM Master WHERE
                                       birthState='WI'");
    foreach($rows as $row){
?>
        <li> <?= $row["nameFirst"] ?>
        <?= $row["nameLast"] ?> </li>
<?php
    }
?>
</ul>
```

Let's assume we are prompting the user to enter a state to select players from. Code would look something like this:

```
<ul>
<?php
    $state = $_GET["state"];
    $db = new PDO("mysql:dbname=krohns;host=localhost",
                  "username", "password");
    //this works, but it's a very bad idea... why?
    $rows = $db->query("SELECT * FROM Master WHERE
                                   birthState='$state'");
    foreach($rows as $row){
?>
        <li> <?= $row["nameFirst"] ?>
        <?= $row["nameLast"] ?> </li>
<?php
    }
?>
</ul>
```

- Then code on the previous page is a security breach called a **SQL injection**.
- It is better to use a built in function so that our query is safe. Below is improved code:

```
<ul>
<?php
    //safer and single quotes are added
    $state = $db->quote($_GET["state"]);
    //db connection here
    $rows = $db->query("SELECT * FROM Master WHERE
                             birthState=$state");
    foreach($rows as $row){
?>
        <li> <?= $row["nameFirst"] ?>
        <?= $row["nameLast"] ?> </li>
<?php
    }
?>
</ul>
```

If we wish to update or delete records, we must use the exec function.

```
<?php
    $db = new PDO("mysql:dbname=krohns;host=localhost",
                  "username", "password");
    $rows = $db->exec("DELETE FROM Master WHERE
                             birthState='WI'");
?>
```

- Queries are returned as a PDOStatement and we treat it like an associative array.
- We almost always use a foreach loop to access all rows of a PDOStatement. Column names are used as the indices.

```
<ul>
<?php
    //safer and single quotes are added
    $state = $db->quote($_GET["state"]);
    //db connection here
    $rows = $db->query("SELECT * FROM Master WHERE
                                 birthState=$state");
    foreach($rows as $row){
?>
        <li> <?= $row["nameFirst"] ?>
        <?= $row["nameLast"] ?> </li>
<?php
    }
?>
</ul>
```

- Incorrect queries can be very hard to debug from PHP.
- You should always construct your query first on the console or in phpmyadmin and make sure your query is valid before trying it in PHP.
- If you get a warning about an invalid argument to the foreach function, odds are your query returned false. Verify your query isn't false before using it.
- If you get an error about an undefined index, make sure the column name is spelled correctly (and available with your query).
- To ensure your connection is closed, after your query, be sure to NULL out your database variable. On the previous slide, we should add $db = NULL at the end.

- Some debugging tools are available for you. One such option is the errorInfo method shown below.
- The array contains 3 elements, the first 2 slots are error codes and the third slot is a somewhat useful string.

```
<ul>
<?php
    $db = new PDO("mysql:dbname=krohns;host=localhost",
                  "username", "password");
    $rows = $db->query("SELECT * FROM Master WHERE
                                  birthState='WI'");
    if($rows === false){
?>
        Database Error: <?= $rows->errorInfo()[2] ?>
<?php
    } else{
        //code when it worked
    }
?>
</ul>
```

- Checking for false works often but isn't perfect. A better way is to check for exceptions.
- Exceptions by default don't print anything. We can (and should) change this setting by setting an attribute.

```
<ul>
<?php
    $db->setAttribute(PDO::ATTR_ERRMODE,
                PDO::ERRMODE_EXCEPTION);
?>
```

- Exception code are useful for debugging but don't really belong there with production code. It's better to surround the code that may crash in a try/catch block.

```
<ul>
<?php
    try{
        //statements that might throw exception
    } catch(PDOException $x){
        //handle the exception somehow
        //a useful debugging error message
        $x->getMessage();
    }
?>
```