Intro to Databases

ACM Webmonkeys 2011

Motivation

- Computer programs that deal with the real world often need to store a large amount of data. E.g.:
  - Weather in US cities by month for the past 10 years
  - List of customers, list of products, etc.
- These data files need to be persistent, modifiable, and searchable.
- So how do we store large amounts of data in an accessible manner?

Simplest way: raw data

- The obvious solution is simply to list the data, row-by-row, in a text file. This is the basis of the CSV (comma-separated value) format:
    - City, Month, Year, Low, Hi
    - Chicago, August, 2008, 69, 80
    - Chicago, September, 2008, 70, 92
- However, searching through this takes a long time.
    - Need to look at every single row
- And we can't keep the whole file in memory, generally, so we need to find a way to manage accessing only portions of it at a time.

Database Management System (DBMS)

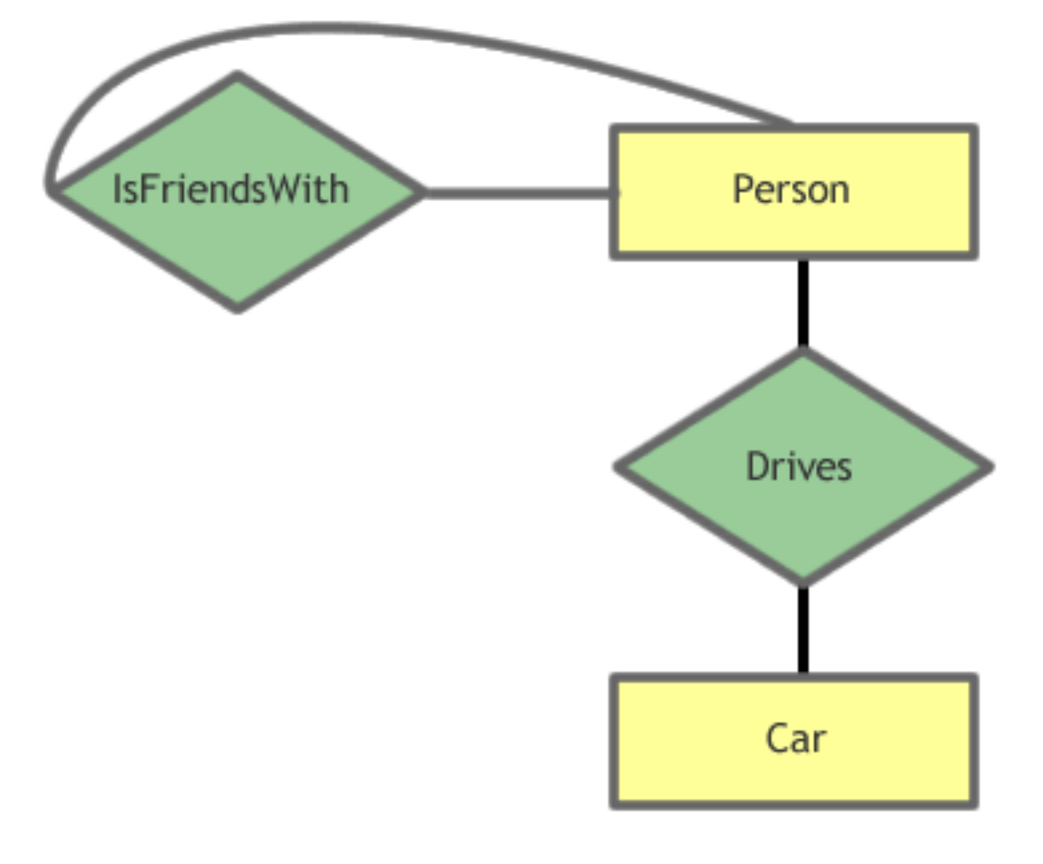
- A database management system, or DBMS, handles all aspects of your database:
  - Storing data
  - Caching
  - Handling queries
- We'll look more into how a DBMS works later.
- Popular DBMSes:
  - MySQL
  - PostgreSQL
  - MSSQL
  - Oracle

Another look at data

- It's easy to think how databases represent database-oriented things, like scientific data.
- But we can use databases to represent much more abstract things.
- To do this, let's first define the term entity to refer to some object or thing that we want to represent.
  - We might have a Person entity, for example.
- The key point is that entities aren't just freestanding, independent things. They're related to each other. So, let's define relationships as how two entities are related.
  - Person is related to itself by a "Is Friends With" relation, and Person might be related to a Car entity with a "Drives" relation.

The Entity-Relational Model

- The Entity-Relational Model is a way of looking at data in this way. To visualize entities and their relationships, we use an ER diagram:



Converting ER to database format

- Databases, however, still work in terms of tables (think spreadsheets). You can only list data in rows.
- However, we can turn an ER model into a database schema by applying pretty straight-forward rules:
  - Each entity becomes its own table
  - Many-to-many relationships become their own table
  - One-to-many relationships have the one's ID stored in the table of the many
  - One-to-one relationships store each other's ID in each's table
- Take CS411 to learn what the types of relationships are and how to do this.

Our example, in database form

- If we convert our example to a database-compatible format, we'll have four tables:
  - Person(PersonID, Name, Age, Address)
  - IsFriendsWith(PersonID, OtherPersonID)
  - Car(CarID, Make, Model, Year)
  - Drives(PersonID, CarID)
- Note that each entity table has an ID field. This is called a primary key, and each table should have one (for the other two tables, the IDs they join define a primary key).
  - Even if other fields uniquely define a row in your table, having an integral primary key is generally a good idea.

Recap up to this point

- So, we have data that we want to store organized into a format based solely around tables.
- How do we get this into an actual database and start using the data?
- Let's pick a DBMS to work with.
  - MySQL is (sort of) the easiest, so we'll use it for this tutorial

MySQL

- MySQL is an open-source, Sun-developed DBMS that is commonly used in conjunction with PHP.
- You can install MySQL directly onto your machine.
  - If you have WAMP or a similar package already installed, then MySQL is already on your machine.
- To begin with, open the mysql prompt, log in, and create a new database:
  - CREATE DATABASE mytestdb;
- Then, set it as our current database:
  - USE mytestdb;

What language are we using?

- The two commands that we just entered are part of the SQL language (at least, MySQL's version of SQL).
- SQL stands for Structured Query Language, and is commonly used in most relational databases.
- The purpose of SQL can be divided into two categories:
  - Data definition
    - e.g., Defining tables
    - CREATE TABLE t;
    - ALTER TABLE t ADD f INTEGER(32);
  - Data manipulation
    - e.g., Retrieving data, modifying data
    - SELECT * FROM t;
    - INSERT INTO t VALUES (1, 2, 3);

CREATE TABLE

- The syntax to create a new table is straight-forward:

```
CREATE TABLE tablename (
    field     TYPE <any attributes>,
    field2    TYPE,
    etc.
)
```

- MySQL has a number of types, but the basic ones are:
  - INTEGER, DECIMAL, VARCHAR (variable-size charstring), TEXT, and DATE
  - VARCHAR and DECIMAL take arguments to specify their sizes. E.g., VARCHAR(255) = 255-byte string.

Creating the person table

- Note that table creation happens only once, so it's common to use GUI interfaces to create tables instead of writing SQL queries. However, for now, we'll use SQL.
- Our Person table has four attributes: PersonID, Name, Age, Address.
- So:

```
CREATE TABLE Person (
    PersonID    INTEGER PRIMARY KEY AUTO_INCREMENT,
    Name        VARCHAR(64),
    Age         INTEGER,
    Address     VARCHAR(255)
);
```

Inserting data

- Let's put some data into our new table. In SQL, the query to insert records into a table is the INSERT command; most commonly, you'll be using it like:
    - INSERT INTO table (field1, field2) VALUES (val1, val);
- So for our Person table, we can do:
    - INSERT INTO Person (Name, Age, Address) VALUES ("Robert", 21, "123 Fake Street");
    - INSERT INTO Person (Name, Age, Address) VALUES ("Bill", 22, "321 Other Street");
    - etc.

The CRUD tasks

- There are four basic types of "normal" SQL queries, of which we've seen one. They handle the four basic data-related tasks:
  - Create
  - Retrieve
  - Update
  - Delete
- These correlate to the following SQL commands:
  - INSERT
  - SELECT
  - UPDATE
  - DELETE
- Remember those four keywords, and you can always look up the proper syntax.

Retrieving data: SELECT

- The most versatile and useful command in SQL is probably the SELECT command, because SELECT is the only way we can get data out of the database.
- The basic format of SELECT is (with considerable variation possible):
  - SELECT expression FROM table;
- You can add several modifiers onto this, the most common being WHERE, i.e.:
  - SELECT expression FROM table WHERE x=y AND z>1;
- Note that expression is usually just a field name (and you can select multiple expressions, just separate them with commas).
  - If you want to get all data, use SELECT * FROM table;

Getting Person data

- To get all the data from our Person table, we can just do:
  - SELECT * FROM Person;
- What if we want only people older than 20?
  - SELECT * FROM Person WHERE Age > 20;
- We can also order our results alphabetically.
  - SELECT * FROM Person ORDER BY Name;
- What if we only want 10 results?
  - SELECT * FROM Person ORDER BY Name LIMIT 10;
- This next query uses an aggregate function, which runs on all the returned data and returns only a single row. What does it do?
  - SELECT MAX(Age) FROM Person;

Selecting from multiple tables

- Selecting from one table is pretty easy. But what if we want to select across multiple tables, taking into account our relationships?
- Let's add ourselves a Cars table, and a Drives table to represent our relation.

```
CREATE TABLE Car ( CarID INTEGER PRIMARY KEY AUTO_INCREMENT,
Make VARCHAR(64), Model VARCHAR(64));
INSERT INTO Car VALUES (1, "Mazda", "Protege");
INSERT INTO Car VALUES (2, "Toyota", "Camry");
CREATE TABLE Drives ( PersonID INTEGER, CarID INTEGER);
INSERT INTO Drives VALUES (1, 1);
INSERT INTO Drives VALUES (2, 1);
```

Joining tables

- SQL can only return as its result a single table. So, the only way to write a query across multiple tables is to somehow join those tables into a single one, then return rows from that single table.
- For example, here we'll want to join the Person table to the Drives table, to get a table where each Person row also has a CarID on the end, and then join it to the Car table.
  - The result will be one table with both person and car information.
- This is not a particularly easy concept.

Finding which car each person drives

```
SELECT * FROM
    (Person JOIN Drives
        ON Person.PersonID = Drives.PersonID)
    JOIN Car ON Drives.CarID = Car.CarID;
```

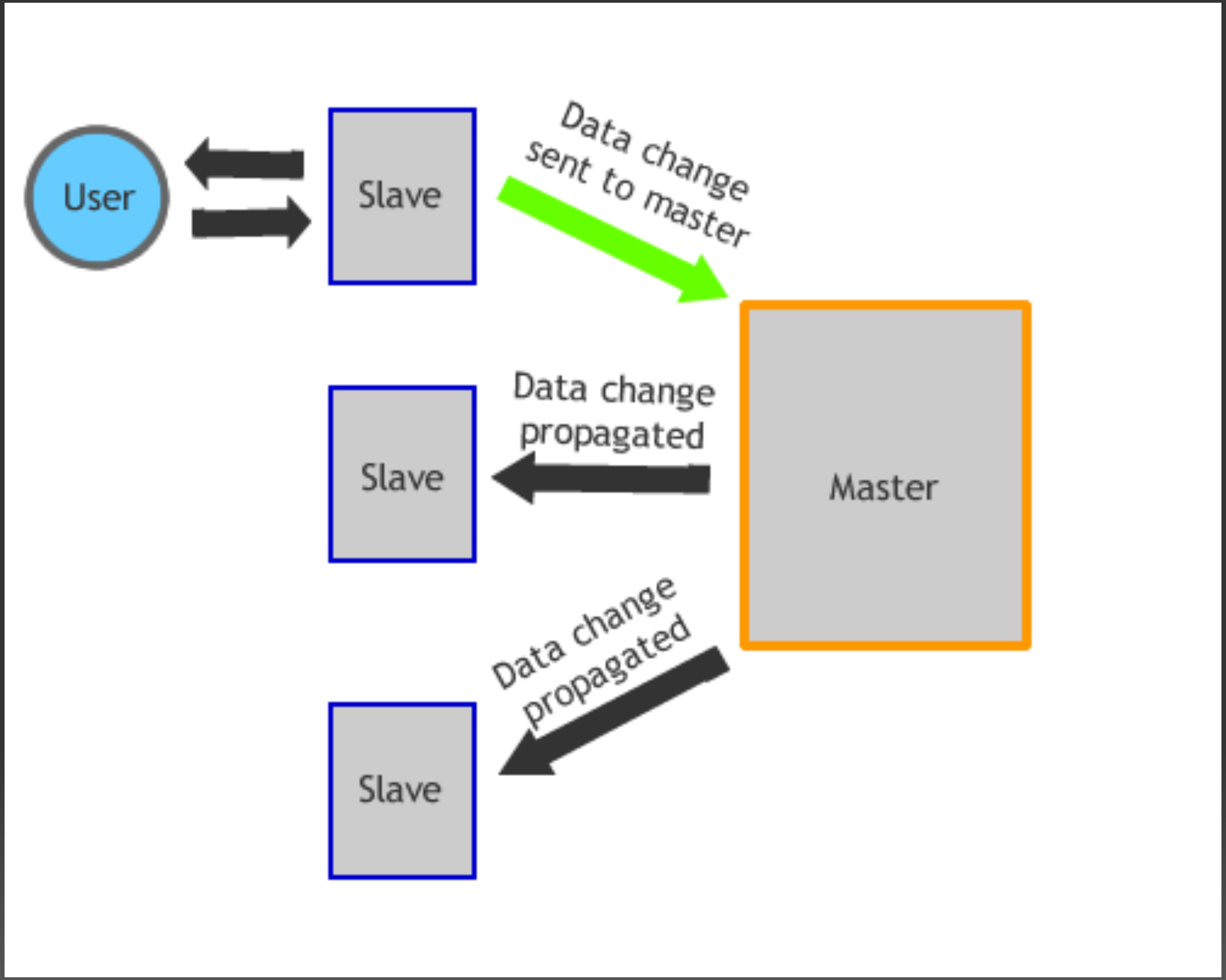
Other database considerations

- As going over all of SQL isn't particular useful, let's move on to some other aspects of modern databases.
- One of the most important features of a database is data consistency
  - The database can crash (this is unavoidable), but the data must not be corrupted
- Maintaining this consistency requires continuous logging.
- Note that, no matter how advanced your database is, it can't survive the loss of the hard disk its data lives on. If you rely on a database, make sure to make and test regular off-site backups.

Scalability through replication

- A single database server can only serve so many requests (maybe 100/second). So how do large web sites with thousands of hits per second scale their databases up?
- The most common solution is replication, which is essentially having the same database files replicated to multiple machines.
  - This makes maintaining consistency difficult -- a user is only dealing with one database server at a time, so changes written to one db need to be propagated to the rest
  - The master database stores and updates the master copy of the data, and sends out changes periodically to update on the slaves.

Replication diagram



Summary

- There's a lot more to modeling data, DBMSes, and SQL than I could cover in this tutorial, but hopefully it's provided a general overview of the topic.
- If you're really interested in databases, CS411 is a class dedicated to them.