Database Theory (in a hurry...)

The assumption is that you have installed MySQL on your chosen operating system.

The Theory in 60 Seconds ...

A **database** is a collection of *related* **tables**.

A **table** is a collection of data arranged as one or more **rows** of **columned** data. (You need at least one row with one column to call it a "table"[1]).

Each data item is often referred to as a **field**. (For example, the "date field", the "last name field", and so on). Data within fields have *type information* associated with it – for example: textual data, date data, binary data.

The information that describes how a database, tables, rows and columns relate to each other is known as the **schema**.

Some aspects of database theory derive from **Set Theory**, so some of the "language" used with databases will be familiar: set, union, intersection, etc.

Designing a Database and Tables - Part 1

Designing a **database** is easy – give it a name and create it:

```
create database gregs_list;
```

Check it exists:

```
show databases;
```

Use it (always a good idea):

```
use gregs_list;
```

Destroy the database[2] (and ***EVERYTHING*** it contains):

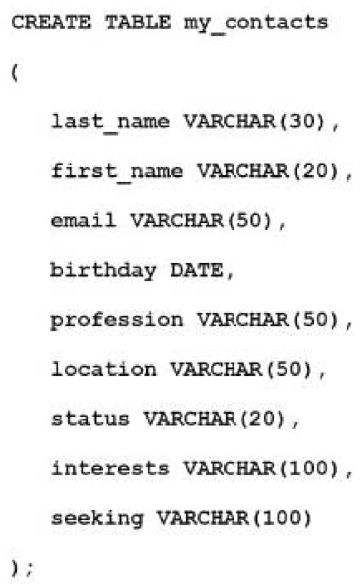
```
drop database gregs_list;
```

1 It would be a pretty useless table, but it's a table all the same.
2 Obviously, don't do this now ... I'm just letting you know you can.

Designing a Database and Tables – Part 2

When designing a **table**, you need to look at the data you want to store then try to match up your data type requirements with the data types available within MySQL – there's a lot of choice: search the web for “MySQL data types” to see what's possible within MySQL[3].

We can create a table to hold Greg's contact information:



```
CREATE TABLE my_contacts
(
   last_name VARCHAR(30),
   first_name VARCHAR(20),
   email VARCHAR(50),
   birthday DATE,
   profession VARCHAR(50),
   location VARCHAR(50),
   status VARCHAR(20),
   interests VARCHAR(100),
   seeking VARCHAR(100)
);
```

If you make a mistake you can, of course, use:

```
drop table my_contacts;
```

to remove the table and start again, or you could alter the table:

```
alter table my_contacts add column gender varchar (1);
```

which is a little easier on your fingers (not to mention your data). It is also possible to alter existing columns and remove them (by dropping individual columns). The on-line MySQL documentation is your friend.

To see what a table looks like (to MySQL), describe it:

```
describe my_contacts;
```

As well as the name and the type, you have information on whether or not NULL values are allowed in a field, whether or not the field is part of a key, a default value for the field and some extra stuff (which is nice).

Now, with the database and table created, all you need is some data ...

3 Other technologies with have other, but similar, types.

Getting data into a table

The general form of the INSERT statement is this:

```
insert into table_name
( ... list of field names ... )
values
( ... list of data values ... );
```

Don't forget those semi-colons! For example:

```
insert into my_contacts
( last_name, first_name )
values
( 'Bloggs', 'Joseph' );
```

The list of field names after the INSERT INTO line is optional but, if you leave them out, be prepared to use *positional listing* of the data values within the second set of parentheses[4].

Getting data out of the table

The SELECT statement is your *friend*. To get everything in a table:

```
select * from my_contacts;
```

which can look pretty awful on some systems (if you have a lot of fields), so be a little more specific about what you want to see in the output:

```
select first_name, last_name from my_contacts;
```

which should look a little nicer. More on using SELECT later, other than to say that it is all but impossible to damage your data with a SELECT statement (unlike something like DROP), so feel free to experiment.

More specific table creations

If you know a lot about your data when creating a table, use your knowledge to more specifically define your table, for example:

```
create table doughnut_list
(
      name  varchar(10) not null,
      type  varchar(6),
      cost  dec(3,2) not null default 1.50
)
```

4 In order words, be prepared to think more!

SELECT can do no harm!

No matter how hard you try, the SELECT statement can not harm your data, as all it ever does is read from the tables within the database.

The same CANNOT be said for UPDATE and DELETE ...

Doing updates with UPDATE

The general form of the UPDATE statement is:

```
UPDATE some_table
SET some_field = 'some_value'
WHERE some_condition;
```

Things to note: UPDATE will not ask for confirmation before it does it stuff – it'll just go ahead and apply the update. If the update can change more than one row, then it will.

General rule: *only perform an UPDATE when you are sure you have found what it is you want to update. In order words, SELECT what you want to update FIRST, then perform the UPDATE.*

Wave "Bye Bye" to your data with DELETE

The general form of the DELETE statement is:

```
DELETE FROM some_table
WHERE some_condition;
```

Like with UPDATE, the DELETE statement assumes you know what you are doing[5], and performs the deletion without asking for confirmation. Once it's gone, it's gone for good ... so be careful.

General rule: *only perform a DELETE when you are sure you have found what it is you want to remove from the table. In order words, SELECT what you want to delete FIRST, then perform the DELETE.*

A little database design goes a long way

When it comes to creating the schema for tables, *design is important*.

It is generally a very bad idea to put more than one piece of data into a single field – better to split the individual pieces of data into individual fields. Not only will this make the data easier on the eye, but it'll make the data easier to work with. This is especially true when it comes to searching your data.

5 I know ... scary.

Getting to the First Normal Form (1-N-F)

You need two things: **atomic data** and a **primary key**. Data is atomic if it conforms to these two rules:

- A column with atomic data cannot have several values of the same type of data in that column.
- A table with atomic data cannot have multiple columns with the same type of data.

So this type of deal is out (as it's a nightmare to search for a specific ingredient ... "*which food uses eggs?*"):

Food	**Ingredients**
Bread	flour, eggs, milk
Salad	lettuce, tomato, eggs

As is this (as it makes adding a new "student" a nightmare as well as searching a pain ... "*who lectures Harry?*"):

Lecturer	**Student1**	**Student2**
Paul	Harry	Mary
Joe	Pat	Harry

Tables that are designed to be atomic will contain less duplicated data and will be much easier (and efficient) to search. But to be "normal", the table must also contain a **primary key**, which can be used to uniquely identify each row in the table.

It is generally a bad idea to rely on the existing data in the existing columns of a table to "create" a primary key ... much better to create a new field to act solely for this purpose. This is easy when creating a table from scratch:

```
CREATE TABLE person
(
      id    INT         NOT NULL AUTO_INCREMENT,
      last  VARCHAR(30) NOT NULL,
      first VARCHAR(40) NOT NULL,
      PRIMARY KEY (id)
);
```

It's only a little more work to add a primary key to an existing table (but not much more work):

```
ALTER TABLE my_contacts
      ADD COLUMN id INT NOT NULL AUTO_INCREMENT FIRST,
      ADD PRIMARY KEY (id);
```

Data is said to be in *First Normal Form* when (a) each row of data contains *atomic data* and (b) each row of data has a *unique identifier* (a primary key).