



In this lab, you will again use the PHPMyAdmin web interface to your database. You will need to install the tables of the social networking database into your database via the IMPORT tab. Then use the SQL tab for direct query entry. The lab covers master-detail relationships via foreign keys and SELECTs on multiple tables.

Note again that these lab sheets are not assessed, but they may be discussed in tutorials and their content is examinable!

Estimated time to complete this lab: 45 minutes.

### **A) Save your cities!**

In this lab, we will create a table called **City**. This may be incompatible with your own existing **city** table (if you're operating your own MySQL DB under Windows) and in either case is going to cause a bit of confusion. Moreover, your table already contains a lot of valuable inserts, which we don't want to lose. So let's make sure your table is safe by renaming it:

```
ALTER TABLE city RENAME TO mycity;
```

This renames your own table to `mycity`.

### **B) Installing the tables for the social networking database**

You will find an SQL script *social\_network.sql* for this purpose on the class website with the lecture notes for lecture 5. Upload this script via the Import tab. This creates the social networking database (which I'll call from here SNDB) as shown in the UML diagram on slide 81 of the notes. It also populates it with some data. You will also want to then upload the script *city.sql* from the class website via the Import tab. This contains the code that creates and populates the new **City** table (which, incidentally, has the same structure as the existing **city** table – at least in the version we have assumed so far). You are encouraged to open the two SQL scripts in a text editor and have a look at them. At least the SQL statements in the *city.sql* script should by now be familiar ones.

### **C) Repetition is boring (and takes up a lot of space)**

Use an appropriate SELECT query to have a look at the User table of the SNDB. Notice something? There is a lot of repetition in the city and country fields of the table. That's because we often have multiple users per city and even more so by country. Yet at present, we're storing the full country name for each user. That's not a big problem now, but what about when we're overtaking Facebook and have a gazillion of them? Then the countries and cities will take up a lot of storage space.

First observation: The city implies the country. Each city is in only one country. So if we know which city a user lives in, we do in principle know the user's country, right? For example, we could look that up in the **City** table. So this means we don't really need to store the same country a million times in the User table – one copy for each city in the **City** table is enough.

All we then need in the **User** table is a reference (a foreign key) into the **City** table. Let's get that organised. First, we need to create an appropriate field for storage of the key in **User**. The **cityId** is an **int NOT NULL**, so the field that we store it in in **User** needs to be an **int NOT NULL** as well. Finally, where in **User** should it go? Since it's meant to replace the **City** and **Country** fields in **User**, we'll put it into the same spot just after the **Address** field and call it **idCity**:

```
ALTER TABLE User ADD idCity int NOT NULL AFTER Address;
```

Now we need to store the actual **cityId** values in **idCity**. Since the user records already exist, we need to UPDATE them for this purpose. The multiple-table UPDATE syntax (see lecture 6) comes in handy here:

```
UPDATE User, City
SET User.idCity = City.cityId
WHERE User.City = City.name AND User.Country = City.country;
```

Note that the only table that actually gets modified here is **User**, because there is no field from the **City** table that is modified in the SET clause of the statement.

Now, we can get rid of the **City** and **Country** fields in the **User** table:

```
ALTER TABLE User DROP City;
ALTER TABLE User DROP Country;
```

Do a **SELECT** on the **User** table and you'll see that it has become a bit slimmer – but also less readable for humans. Fear not, a simple **SELECT** gives us the full picture again:

```
SELECT * FROM User, City WHERE idCity = cityId;
```

#### **D) Enforcing the foreign key relationship**

So far, so good. At present, there's a danger though: What if we insert a new user and specify an **idCity** value that has no corresponding **cityId** value? Or, just as bad, what if we delete a city from the **City** table for which we still have users? Clearly, that's undesirable. The solution is to turn **idCity** officially into a foreign key.

For this, we must first turn the **idCity** field into an index (key) into the **User** table:

```
ALTER TABLE User ADD KEY (idCity);
```

Then we can enable the foreign key relationship. However, in doing so, we must temporarily prevent the database from enforcing this relationship – otherwise we fall victim to a known MySQL bug:

```
SET foreign_key_checks = 0;
ALTER TABLE User ADD FOREIGN KEY (idCity) REFERENCES City (cityId);
SET foreign_key_checks = 1;
```

Done. Try deleting an entry from the **City** table that's in use, e.g., the one for Auckland:

```
DELETE FROM City WHERE cityId = 1;
```

This now produces an error because there are records in the **User** table whose **idCity** is 1. The try inserting a **User** record with a non-existent **idCity** value and you will get an error message as well – that's just what we want!

#### **E) Exercise: Do we really need to store all of these country names all over again for each city?**

Of course not. Create a **Country** table with an auto-incrementing primary key **countryId** and an appropriate varchar field **name** for each country. Let's stop and think for a moment: We're trying to save space, right? So we'd like to get away with the smallest integer possible for the primary key. How many countries are there in the world? Just under 200, in fact, and the records aren't going to change that often. So a **tinyint unsigned** with a range of 0 to 255 should do just fine for the **countryId**. To allow the use of the foreign key later, you'll also need to add **ENGINE=InnoDB** to the CREATE TABLE definition. It goes after the closing parenthesis of the statement but before the terminating semicolon!

Then use an **INSERT...SELECT** statement to copy the country names across from the **City** table to the **Country** table:

```
INSERT INTO Country (name)
SELECT DISTINCT country FROM City;
```

This is another form of the **INSERT** statement where multiple records can be inserted into the specified field(s) (in this case **name**) of a table (in this case **Country**) at the same time. These records are given by the **SELECT** query that follows as the second part of the **INSERT** statement. In this case, we need a **SELECT DISTINCT** because some countries turn up more than once in the **City** table.

The next step is to reference the countries in the **City** table by means of their **countryId** value from the **Country** table. For this, you will need an extra **tinyint unsigned** field, ideally just before the **country** field in the **City** table. In Section C), you updated the **User** table with an id from the **City** table, and the next step here is to update the new field in the extended **City** table with the key values from the **Country** table. Once that is done, you can drop the **country** column in the **City** table.

Finally, ensure that there is a proper foreign key relationship in place: Export the database tables via PHPMyAdmin as SQL and unclick "Save as file". In the response, scroll down: The foreign key constraints are listed right at the bottom of the export SQL statements, and there should be one mentioning each foreign key in the DB. If yours isn't there, then something hasn't worked (note: don't always expect an error message!).

Note: Typical reasons for why setting up a foreign key relationship may fail silently in MySQL are:

- One of the tables doesn't run with the InnoDB engine
- The fields in the two tables have different types (they must be the same type, both signed or both unsigned, and both must be NOT NULL)
- The field in the table that uses the foreign key is not declared as a key or index
- You've tried to put the foreign key in place before having the other things above sorted.

Another (more dangerous) way of checking whether this has worked is to check that you are now prevented from accidentally deleting countries from the **Country** table for which there is still a city in the **City** table, and that prevents you from adding cities for which there is no corresponding country. Use appropriate DELETE and INSERT queries to verify that this does indeed work.

Once you have it all going, use an appropriate SELECT query to get a dataset of users with their city and country as plain text names.