



In this lab, you will again use the PHPMyAdmin web interface to your database. You will need to install a database table via the IMPORT tab. This table will not even be in 1NF. Your task will be to bring this table successively to 1NF, 2NF and 3NF.

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: 30 minutes.

A) Keeping on top of your MP3 files

To start this exercise, import the file mp3collection.sql into your database via the Import tab of the PHPMyAdmin web interface. Now have a look at the **MP3** table created by this script: It contains four fields. The first describes the artist, the second the album, and the third gives a list of tracks from that album, and the fourth column tells you on which harddrive you have stored the associated MP3 file. The primary key is defined on the unique combination of artist, album, and harddrive.

So far, so good, it's all there. However, say you wanted to search for a particular song by title. Then you need to go record by record through the **tracks** fields of the table, parse them and see whether the song is in there:

```
SELECT * FROM MP3 WHERE tracks LIKE '%Nobody does it later%';
```

This is a little awkward, especially if we then also want to find out how long the song is going to play.

Formally speaking, this is because the table isn't in first normal form (1NF). It violates the requirement of 1NF that fields should never contain more than one data item from a particular domain. In this case, the **tracks** field should really be a **track** field, and the time for each song should be recorded in a separate field.

So you'd be looking at an alternative CREATE TABLE like this (don't try this out just yet):

```
CREATE TABLE MP3 (  
  artist varchar(100),  
  album varchar(100),  
  track text,  
  duration varchar(5),  
  disk varchar(11),  
  PRIMARY KEY(artist,album,disk)  
);
```

But now you have a different problem: You can't use (**artist,album,disk**) as a primary key any longer because it is no longer unique: Any two tracks from the same album stored on the same disk will have the same key value. So maybe try (**artist,track,disk**) as a primary key – this covers even the case of having the same song twice on two different disks:

```
CREATE TABLE MP3 (  
  artist varchar(100),  
  album varchar(100),  
  track text,  
  duration varchar(5),  
  disk varchar(11),  
  PRIMARY KEY(artist,track,disk)  
);
```

The table is now in 1NF. Rewrite the SQL script so that it reflects this table.

B) But is it 2NF?

No. Remember that the criterion for a table to be 2NF is that none of its non-key fields must be dependent on part of the key only. Moreover, this must apply to all "candidate keys", i.e., all combinations of fields that might yield a unique key:

These are:

- **(artist, track, disk)** –currently in use as primary key
- **(album, track, disk)** – assuming that no two artists will have an album of the same name and simultaneously a track of the same name on it

However: In the first case, the duration of a track is determined by the artist or album and track alone, not by the disk. So the table clearly isn't in 2NF.

The cleanest way out is usually to split tables at this point and used dedicated primary keys: a table of albums, artists, disks each with its own dedicated primary key. Then have a table that links albums to artists, another that links tracks to albums and records how long the tracks are, and one that links tracks to disks. So you might end up with something like this:

```
CREATE TABLE Album (
    album_id int unsigned not null auto_increment,
    album varchar(100),
    artist varchar(100),
    PRIMARY KEY (album_id)
);

CREATE TABLE Track (
    track_id int unsigned not null auto_increment,
    track varchar(100),
    album_id int unsigned not null,
    duration varchar(5),
    PRIMARY KEY (track_id)
);

CREATE TABLE Disk (
    disk_id int unsigned not null auto_increment,
    disk varchar(11),
    PRIMARY KEY (disk_id)
);

CREATE TABLE Disktrack (
    disktrack_id int unsigned not null auto_increment,
    disk_id int unsigned not null,
    track_id int unsigned not null,
    PRIMARY KEY (disktrack_id)
);
```

Now, all the tables are in 2NF. Can you think of other solutions?

C) Getting to 3NF

Presuming we've picked the solution above, are we 3NF? Remember the criterion for this is that any field in a table must depend on the primary key only, i.e., if we update the field, this shouldn't normally require us to update any other field.

Reality check:

Album table:	Is 3NF	Isn't 3NF	If not, why:
Track table:	Is 3NF	Isn't 3NF	If not, why:
Disk table:	Is 3NF	Isn't 3NF	If not, why:
Disktrack table:	Is 3NF	Isn't 3NF	If not, why:

Exercise: Change the DB to make all tables involved 3NF.