

MySQL: Learning the Database Design Process

http://www.informit.com/isapi/product_id~%7BBE2D83E8-E7A1-4B31-BCD6-32BAE1C8C334%7D/st~%7BDB8A27E8-F4F6-483E-8F13-C4A8BDAE0F14%7D/session_id~%7B214C7DB9-9787-4F3B-9D18-5FC6F1E2C8B2%7D/content/index.asp

Introduction

In this hour, you'll learn the thought processes behind designing a relational database. This will be the last theory-focused hour; you'll soon be ready to jump headlong into creating MySQL databases for use in your own applications.

Topics covered in this hour are:

- Some advantages to good database design
- Three types of table relationships
- How to normalize your database
- How to implement a good database design process

The Importance of Good Database Design

Good database design is crucial for a high performance application, just like an aerodynamic body is important to a racecar. If the racecar doesn't have smooth lines, it will produce drag and go slower. The same holds true for databases. If a database doesn't have optimized relationships—normalization—it won't be able to perform as efficiently as possible.

Beyond performance is the issue of maintenance. Your database should be easy to maintain. This includes storing a limited amount (if any) of repetitive data. If you have a lot of repetitive data and one instance of that data undergoes a change (such as a name change), that change has to be made for all occurrences of the data. To eliminate duplication and enhance your ability to maintain the data, you would create a table of possible values and use a key to refer to the value. That way, if the value changes names, the change occurs only once—in the master table. The reference remains the same throughout other tables.

For example, suppose you are responsible for maintaining a database of students and the classes in which they're enrolled. If thirty-five of these students are in the same class, called "Advanced Math," this class name would appear thirty-five times in the table. Now, if the instructor decides to change the name of the class to "Mathematics IV," you must change thirty-five records to reflect the new name of the class. If the database were

designed so that class names appeared in one table and just the class ID number was stored with the student record, you would only have to change one record—not thirty-five—in order to update the name change. The benefits of a well-planned and designed database are numerous, and it stands to reason that the more work you do up front, the less you'll have to do later. A really bad time for a database redesign is after the public launch of the application using it—although it does happen, and the results are costly. So, before you even start coding your application, spend a lot of time designing your database. Throughout the rest of this hour, you'll learn more about relationships and normalization, two important pieces to the design puzzle.

Types of Table Relationships

In Hour 2, you learned that keys are used to tie tables together. These relationships come in several forms:

- One-to-one relationships
- One-to-many relationships
- Many-to-many relationships

For example, suppose you have a master `employees` table for employees, containing their Social Security number, name, and the department in which they work. You also have a separate table containing the list of all available departments, made up of a Department ID and a name. In the `employees` table, the Department ID field will match an ID found in the `departments` table. As you've learned by now, this is a type of relationship, which you can see in [Figure 3.1](#). The "PK" next to the field name stands for *primary key*, which you'll learn more about during this lesson.

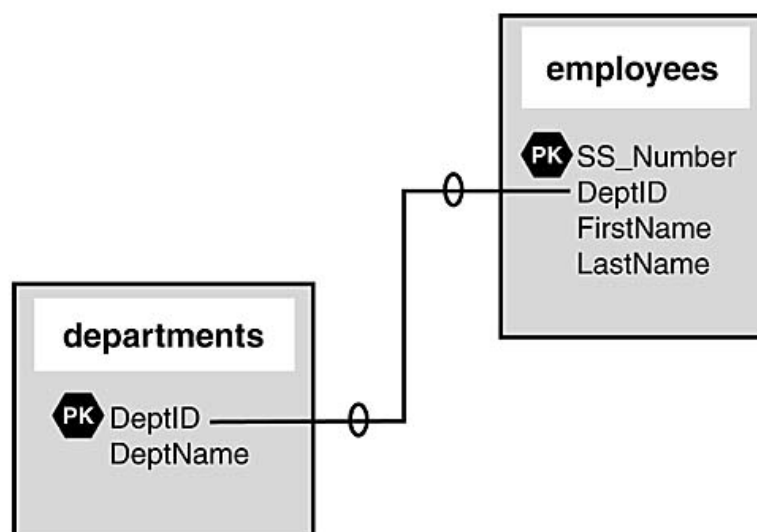


Figure 3.1 The employees and departments tables are related through the DeptID.

The next few sections will take a closer look at each of the relationship types.

One-to-One Relationships

In a one-to-one relationship, a key will appear only once in a related table. The example of the employees and departments tables is not a one-to-one relationship, as many employees will undoubtedly belong to the same department. An example of a one-to-one relationship is if each employee is assigned one computer within a company. [Figure 3.2](#) shows the one-to-one relationship of employees to computers.

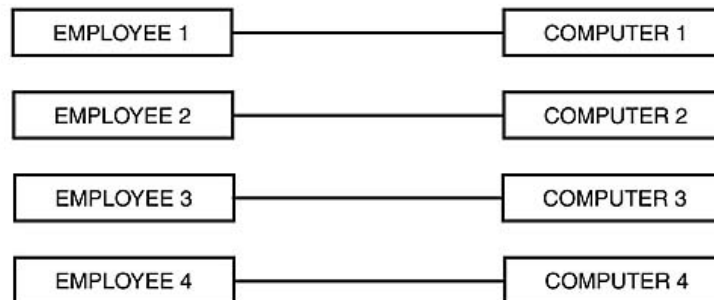


Figure 3.2 One computer is assigned to each employee.

The employees and computers tables in your database would look something like [Figure 3.3](#), which represents a one-to-one relationship.

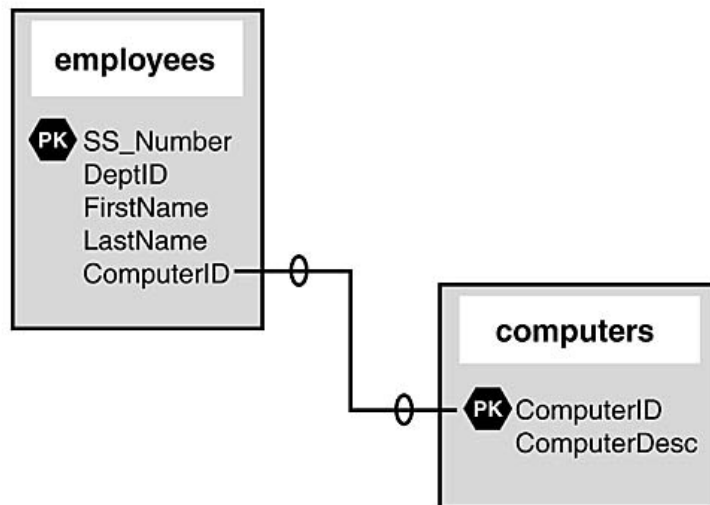


Figure 3.3 One-to-one relationship in the data model.

One-to-Many Relationships

In a one-to-many relationship, keys from one table will appear multiple times in a related table. The example shown in [Figure 3.1](#), indicating a connection between employees and departments, is an example of a one-to-many relationship. A real-world example would be an organizational chart of the department, shown in [Figure 3.4](#).

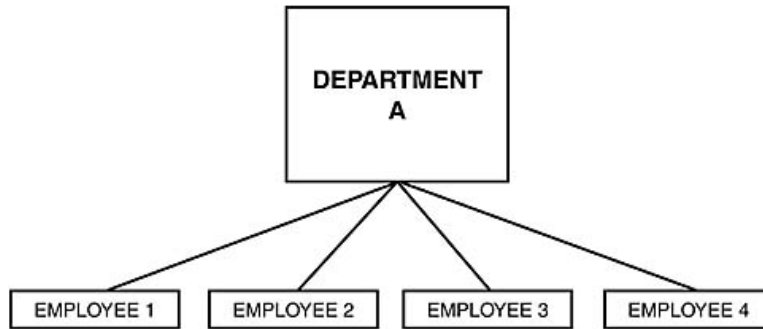


Figure 3.4 One department contains many employees.

The one-to-many relationship is the most common type of relationship. Another practical example is the use of a state abbreviation in an address database; each state has a unique identifier (CA for California, PA for Pennsylvania, and so on) and each address in the United States has a state associated with it.

If you have eight friends in California and five in Pennsylvania, you will use only two distinct abbreviations in your table. One abbreviation represents a one-to-eight relationship (CA), and the other represents a one-to-five (PA) relationship.

Many-to-Many Relationships

The many-to-many relationship often causes problems in practical examples of normalized databases, so much so that it is common to simply break many-to-many relationships into a series of one-to-many relationships. In a many-to-many relationship, the key value of one table can appear many times in a related table. So far, it sounds like a one-to-many relationship, but here's the curveball: the opposite is also true, meaning that the primary key from that second table can also appear many times in the first table.

Think of it this way, using the example of students and classes. A student has an ID and a name. A class has an ID and a name. A student will usually take more than one class at a time, and a class will always contain more than one student, as you can see in [Figure 3.5](#).

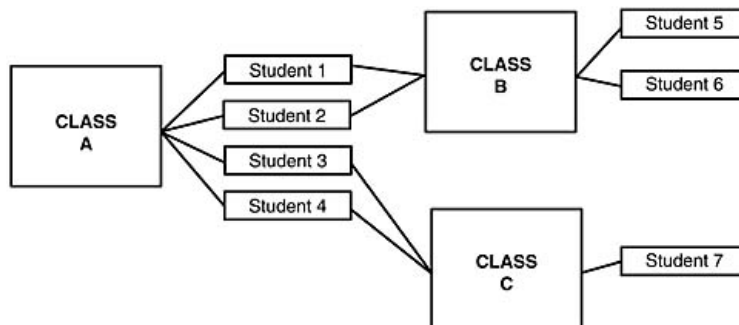


Figure 3.5 Students take classes, classes contain students.

As you can see, this sort of relationship doesn't present an easy method for relating tables. Your tables could look like [Figure 3.6](#), seemingly unrelated.

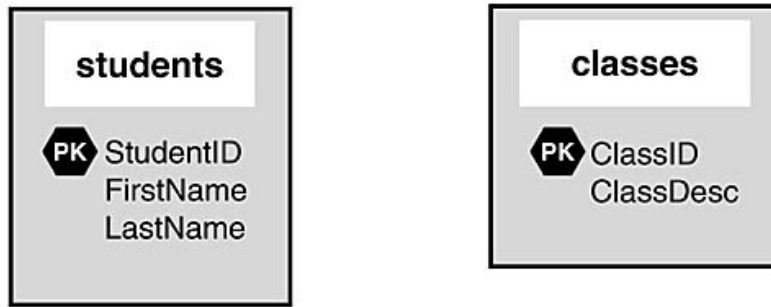


Figure 3.6 The students table and the classes table, unrelated.

In order to make the theoretical many-to-many relationship, you would create an intermediate table, one that sits between the two tables and essentially maps them together. You might build one similar to that in [Figure 3.7](#).

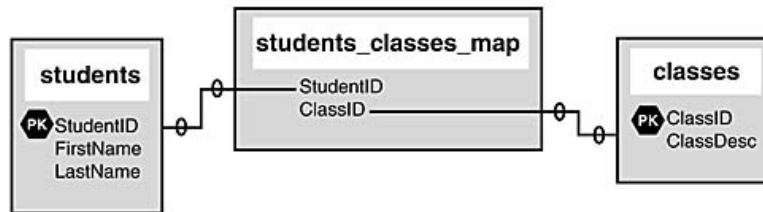


Figure 3.7 The table students_classes_map acts as an intermediary.

If you take the information in Figure 3.5 and put it into the intermediate table, you would have something like [Figure 3.8](#).

STUDENTID	CLASSID
STUDENT 1	CLASS A
STUDENT 2	CLASS A
STUDENT 3	CLASS A
STUDENT 4	CLASS A
STUDENT 5	CLASS B
STUDENT 6	CLASS B
STUDENT 7	CLASS C
STUDENT 1	CLASS B
STUDENT 2	CLASS B
STUDENT 3	CLASS C
STUDENT 4	CLASS C

Figure 3.8 The table students_classes_map populated with data.

As you can see, many students and many classes happily co-exist within the students_classes_map table.

With this introduction to the types of relationships, learning about normalization should be a snap!

Understanding Normalization

Normalization is simply a set of rules that will ultimately make your life easier when you're wearing your database administrator hat. It's the art of organizing your database in such a way that your tables are related where appropriate and flexible for future growth.

The sets of rules used in normalization are called *normal forms*. If your database design follows the first set of rules, it's considered in the *first normal form*. If the first three sets of rules of normalization are followed, your database is said to be in the *third normal form*.

Throughout this hour, you'll learn about each rule in the first, second, and third normal forms and hopefully will follow them as you create your own applications. You'll be using an example set of tables for a students and courses database and taking it to the third normal form.

Problems with the Flat Table

Before launching into the first normal form, you have to start with something that needs to be fixed. In the case of a database, it's the *flat table*. A flat table is like a spreadsheet—many, many columns. There are no relationships between multiple tables; all the data you could possibly want is right there in that flat table. This scenario is inefficient and consumes more physical space on your hard drive than a normalized database.

In your students and courses database, assume you have the following fields in your flat table:

- StudentName—The name of the student.
- CourseID1—The ID of the first course taken by the student.
- CourseDescription1—The description of the first course taken by the student.
- CourseInstructor1—The instructor of the first course taken by the student.
- CourseID2—The ID of the second course taken by the student.
- CourseDescription2—The description of the second course taken by the student.
- CourseInstructor2—The instructor of the second course taken by the student.

- Repeat CourseID, CourseDescription, and CourseInstructor columns many more times to account for all the classes a student can take during their academic career.

With what you've learned so far, you should be able to identify the first problem area: CourseID, CourseDescription, and CourseInstructor columns are repeated groups.

Eliminating redundancy is the first step in normalization, so next you'll take this flat table to first normal form. If your table remained in its flat format, you could have a lot of unclaimed space, and a lot of space being used unnecessarily—not an efficient table design!

First Normal Form

The rules for the first normal form include

- Eliminate repeating information.
- Create separate tables for related data.

If you think about the flat table design, with many repeated sets of fields for the student and courses database, you can identify two distinct topics: students and courses. Taking your student and courses database to the first normal form would mean that you create two tables: one for students and one for courses, shown in [Figure 3.9](#).

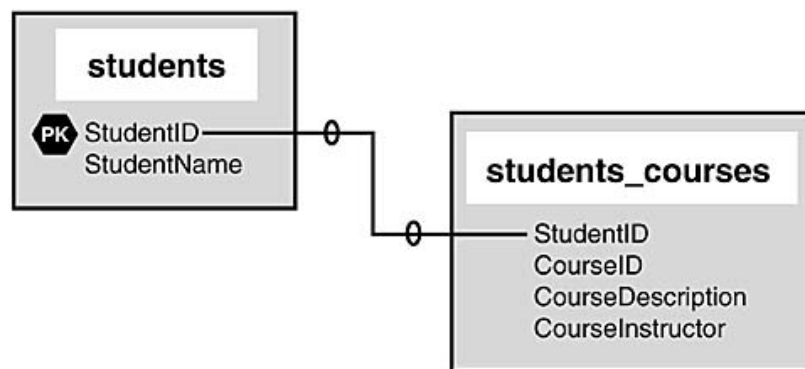


Figure 3.9 Breaking the flat table into two tables.

Your two tables now represent a one-to-many relationship of one student to many courses. Students can take as many courses as they wish and are not limited to the number of CourseID/CourseDescription/CourseInstructor groupings that existed in the flat table.

The next step is to put the tables into second normal form.

Second Normal Form

The rule for the second normal form is

- No non-key attributes depend on a portion of the primary key.

In plain English, this means that if fields in your table are not entirely related to a primary key, you have more work to do. In the students and courses example, it means breaking out the courses into their own table, and modifying the students_courses table.

CourseID, CourseDesc, and CourseInstructor can become a table called courses with a primary key of CourseID. The students_courses table should then just contain two fields: StudentID and CourseID. You can see this new design in [Figure 3.10](#).

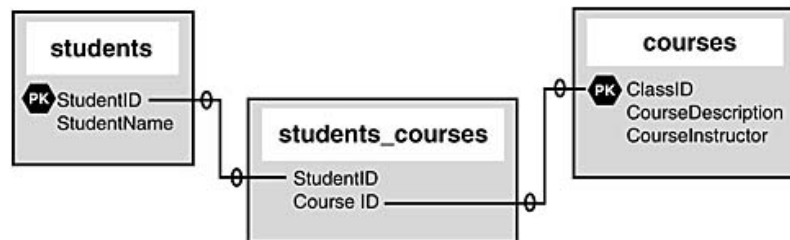


Figure 3.10 Taking your tables to second normal form.

This structure should look familiar to you as a many-to-many relationship using an intermediary mapping table. The third normal form is the last form we'll look at, and you'll find it's just as simple to understand as the first two.

Third Normal Form

The rule for the third normal form is

- No attributes depend on other non-key attributes.

This rule simply means that you need to look at your tables and see if more fields exist that can be broken down further and that aren't dependent on a key. Think about removing repeated data and you'll find your answer—
instructors. Inevitably, an instructor will teach more than one class. However, CourseInstructor is not a key of any sort. So, if you break out this information and create a separate table purely for the sake of efficiency and maintenance (as shown in [Figure 3.11](#)), that's the third normal form.

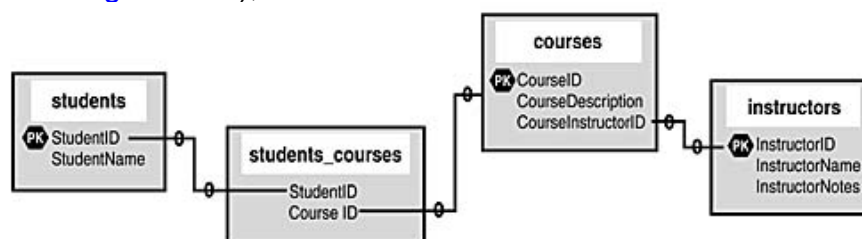


Figure 3.11 Taking your tables to third normal form.

Third normal form is usually adequate for removing redundancy and allowing for flexibility and growth. The next section will give you some pointers for the thought process of database design and where it fits in the overall design process of your application.

Following the Design Process

The greatest problem in application design is a lack of forethought. As it applies to database-driven applications, the design process must include a

thorough evaluation of your database—what it should hold, how data relates to each other, and most importantly, is it scalable?

The general steps in the design process are:

- Define the objective.
- Design the data structures (tables, fields).
- Discern relationships.
- Define and implement business rules.
- Create the application.

Creating the application is the last step—not the first! Many developers take an idea for an application, build it, then go back and try to make a set of database tables fit into it. This approach is completely backwards, inefficient, and will cost a lot of time and money.

Before starting any application design process, sit down and talk it out. If you can't describe your application—including the objectives, audience, and target market—then you're not ready to build it, let alone model the database.

Once you can describe the actions and nuances of your application to other people and have it make sense to them, you can start thinking about the tables you want to create. Start with big flat tables because, once you write them down, your newfound normalization skills will take over. You will be able to find your redundancies and visualize your relationships.

The next step is to do the normalization. Go from flat table, to first normal form, and so on, up to the third normal form if possible. Use paper, pencils, Post-it Notes, or whatever helps you to visualize the tables and relationships. There's no shame in data modeling on Post-it Notes until you're ready to create the tables themselves. Plus, they're a lot cheaper than buying software to do it for you, which range from one hundred to several thousands of dollars!

After you have a preliminary data model, look at it from the application's point of view. Or look at it from the point of view of the person using the application you're building. This is the point where you define business rules and see if your data model will break. An example of a business rule for an online registration application is, "Each user must have one e-mail address, and it must not belong to any other user." If EmailAddress weren't a unique field in your data model, then your model would be broken based on the business rule.

After your business rules have been applied to your data model, only then can application programming begin. You can rest assured that your data model is solid and you will not be programming yourself into a brick wall. The latter event is all too common.

Summary

Proper database design is the only way your application will be efficient, flexible, and easy to manage and maintain. An important aspect of database design is to use relationships between tables instead of throwing all your data into one long flat file. Types of relationships include one-to-one, one-to-many, and many-to-many.

Using relationships to properly organize your data is called normalization. There are many levels of normalization, but the primary levels are the first, second, and third normal forms. Each level has a rule or two that must be followed. Following all of the rules will help ensure that your database is well organized and flexible.

To take an idea from inception through to fruition, you should follow a design process. This process essentially says "think before you act." Discuss rules, requirements, and objectives, and then create the final version of your normalized tables.

In the next hour, you'll get underway with using MySQL through any number of interfaces.

Q&A

Q. Are there only three normal forms?

A. No, there are more than three normal forms. Additional forms are the Boyce-Codd Normal Form, Fourth Normal Form, Fifth Normal Form/Join-Projection Normal Form. These forms are not often followed because the benefits of doing so are outweighed by the cost in man-hours and database efficiency.