

---

# An Introduction to Database Normalization

(2000-11-27) - Contributed by W.J. Gilmore

A database can be great fun, right? Yes, of course! There are though, a couple things that can ruin all that hard work and effort you put into your efficient little database. Today we discuss how to keep that beloved bin of data from going bad on you: database normalization.

Call me a nerd, but I'll never forget the elation I felt several years back when I first succeeded in connecting a database to a Web page. At the time a newcomer to the world of database administration, I happily began creating all kinds of databases to store my valuable information. However, several problems soon arose, due in large part to the techniques that I had employed when designing the tables that were employed to store my data. The tables were becoming increasingly difficult to maintain, with some of the data even getting unwittingly deleted or modified! Thoroughly disgusted with the thought of losing control of my data, I set out to learn more about the mechanics of efficient database administration, concentrating upon a particularly important aspect named database normalization. Database normalization can essentially be defined as the practice of optimizing table structures. Optimization is accomplished as a result of a thorough investigation of the various pieces of data that will be stored within the database, in particular concentrating upon how this data is interrelated. An analysis of this data and its corresponding relationships is advantageous because it can result both in a substantial improvement in the speed in which the tables are queried, and in decreasing the chance that the database integrity could be compromised due to tedious maintenance procedures. Before delving further into the subject of db normalization, allow me to introduce a few terms that frequently arise when discussing this subject. To better illustrate the meaning of the respective terms, I'll allude to a hypothetical database which contains information about a school scheduling system. {mospagebreak title=Preliminary Definitions} In this section I introduce several definitions that are common jargon in the world of database administration and normalization.

**entity:** The word 'entity' as it relates to databases can simply be defined as the general name for the information that is to be stored within a single table. For example, if I were interested in storing information about the school's students, then 'student' would be the entity. The student entity would likely be composed of several pieces of information, for example: student identification number, name, and email address. These pieces of information are better known as **attributes**.

**primary key:** A primary key uniquely identifies a row of data found within a table. Referring to the school system, the student identification number would be the primary key for the student table since an ID would uniquely identify each student. Note that a primary key might not necessarily correspond to one specific attribute. In fact, it could be the result of a combination of several components of the entity. For example, while a location could not be a primary key for a class, since there might be several classes held there throughout the day, the combined time and location would make a satisfactory primary key, since no two classes could be held at the same time in the same location. When multiple attributes are used to derive a primary key, this key is known as a **concatenated primary key**.

**relationship:** Understanding of the various relationships both between the data items forming the various entities and between the entities themselves forms the crux of database normalization. There are three types of data relationships that you should be aware of:

- **one-to-one (1:1)** - A one-to-one relationship signifies that each instance of a given entity relates to exactly one instance of another entity. For example, each student would have exactly one grade record, and each grade record would be specific to one student.
- **one-to-many (1:M)** - A one-to-many relationship signifies that each instance of a given entity relates to one or more instances of another entity. For example, one professor entity could be found teaching several classes, and each class could in turn be mapped to one professor.
- **many-to-many (M:N)** - A many-to-many relationship signifies that many instances of a given entity relate to many instances of another entity. To

---

illustrate, a schedule could be comprised of many classes, and a class could be found within many schedules. foreign key: A foreign key forms the basis of a 1:M relationship between two tables. The foreign key can be found within the M table, and maps to the primary key found in the 1 table. To illustrate, the primary key in the professor table (probably a unique identification number) would be introduced as the foreign key within the classes entity, since it would be necessary to map a particular professor to several classes. Entity-relationship diagram (ERD): An ERD is essentially a graphical representation of the database structure. These diagrams, regardless of whether they are built using the latest design software or scrawled on a napkin with a crayon, are immensely useful towards attaining a better understanding of the dynamics of the various database relationships. Click here to examine a sample ERD diagram which illustrates the relational structure that might be found in our school system database.

{mospagebreak title=So Why Normalize?} Thus far the only thing that I have really stated about database normalization is that it provides for table optimization through the investigation of entity relationships. But why is this necessary? In this section, I'll elaborate a bit upon why normalization is necessary when creating commercial database applications. Essentially, table optimization is accomplished through the elimination of all instances of data redundancy and unforeseen scalability issues.

RedundancyData redundancy is exactly what you think it is; the repetition of data. One obvious drawback of data repetition is that it consumes more space and resources than is necessary. Consider the following table:

student_id
class_name
time
location
professor_id
999-40-9876
Math 148
MWF 11:30
Rm. 432
prof145
999-43-0987
Physics 113
TR 1:30
Rm. 12
prof143
999-42-9842
Botany 42

---

F 12:45

Rm. 9

prof167

999-41-9832

Matj 148

MWF 11:30

Rm. 432

prof145 Basically this table is a mapping of various students to the classes found within their schedule. Seems logical enough, right? Actually, there are some serious issues with the choice to store data in this format. First of all, assuming that the only intention of this table is to create student-class mappings, then there really is no need to repeatedly store the class time and professor ID. Just think that if there are 30 students to a class, then the class information would be repeated 30 times over! Moreover, redundancy introduces the possibility for error. You might have noticed the name of the class found in the final row in the table (Matj 148). Given the name of the class found in the first row, chances are that Matj 148 should actually be Math 148! While this error is easily identifiable when just four rows are present in the table, imagine finding this error within the rows representing the 60,000 enrolled students at my alma mater, The Ohio State University. Chances that you'll find these errors are unlikely, at best. And the cost of even attempting to find them will always be high.

Unforeseen Scaleability Issues Unforeseen scaleability issues generally arise due to lack of forethought pertaining to just how large a database might grow. Of course, as a database grows in size, initial design decisions will continue to play a greater role in the speed of and resources allocated to this database. For example, it is typically a very bad idea to limit the potential for expansion of the information that is to be held within the db, even if there are currently no plans to expand. For example, structurally limiting the database to allot space for only three classes per student could prove deadly if next year the school board decides to permit all students to schedule three classes. This also works in the opposite direction; What if the school board subsequently decides to only allow students to schedule two classes? Have you allowed for adequate flexibility in the design so as to easily adapt to these new policies? The remedy to these problems is through the use of a process known as database normalization. A subject of continued research and debate over the years, several general rules have been formulated that layout the process one should follow in the quest to normalize a database. I'll discuss these rules in the next section, "The Three Normal Forms". {mospagebreak title=The Three Normal Forms} The process towards database normalization progressing through a series of steps, typically known as Normal Forms. For purposes of illustration, assume that a school system used a table containing these attributes to store its information. As you can see, employing this strategy results in a lookup mechanism that essentially defeats the purpose of using a database; it's just a group of records. In short, this table is in dire need of a normalization overhaul. In this section, I'll implement the rules specified by the first three Normal Form rules to reorganize this school's table structure.

First Normal Form Converting a database to the first normal form is rather simple. This first rule calls for the elimination of repeating groups of data through the creation of separate tables of related data. Obviously, the original table contains several sets of repeating groups of data, namely classID, className, classTime, classLocation, professorID, professorName. Each attribute is repeated three times, allowing for each student to take three classes. However, what if the student takes more than three classes? This, and

---

other restrictions on this table should be obvious. Therefore, let's break this mammoth table down into several smaller tables. The first table contains solely student information (Student):

studentID

studentName

Major

college

collegeLocation  
The second table contains solely class information (Class):

studentID

classID

className  
The third table contains solely professor information (Professor):

professorID

professorName

Second Normal Form  
Once you have separated the data into their respective tables, you can begin concentrating upon the rule of Second Normal Form; that is, the elimination of redundant data. Referring back to the Class table, typical data stored within might look like:

---

studentID

classID

className

134-56-7890

M148

Math 148

123-45-7894

P113

Physics 113

534-98-9009

H151

History 151

134-56-7890

H151

History 151 While this table structure is certainly improved over the original, notice that there is still room for improvement. In this case, the className attribute is being repeated. With 60,000 students stored in this table, performing an update to reflect a recent change in a course name could be somewhat of a problem. Therefore, I'll

---

create a separate table that contains classID to className mappings  
(ClassIdentity):

classID

className

M148

Math 148

P113

Physics 113

H151

History 151  
The updated Class  
table would then be simply:

studentID

classID

134-56-7890

M148

---

123-45-7894

P113

534-98-9009

H151

134-56-7890

H151 Revisiting the need to update a recently changed course name, all that it would take is the simple update of one row in the ClassIdentity table! Of course, substantial savings in disk space would also result, due to this elimination of redundancy.

Third Normal Form Continuing on the quest for complete normalization of the school system database, the next step in the process would be to satisfy the rule of the Third Normal Form. This rule seeks to eliminate all attributes from a table that are not directly dependent upon the primary key. In the case of the Student table, the college and collegeLocation attributes are less dependent upon the studentID than they are on the major attribute. Therefore, I'll create a new table that relates the major, college and collegeLocation information:

major

college

collegeLocation The revised Student table would then look like:

studentID

studentName

---

MajorAlthough for most cases these three Normal Forms sufficiently satisfy the requirements set for proper database normalization, there are still other Forms that go beyond what rules have been set thus far. However, these are out of the scope of this article. If you would be interested in learning more about these Forms, there have been a number of books written on the subject. Check out your local bookstore for more information.

{mospagebreak title=What's Next}

Taking time to properly design your database is arguably the most important step in developing database-oriented applications. This article focused upon informing you of the basic premises of efficient db design, introducing database normalization, general definitions, and the first three Normal Forms. Next article, I'll focus upon applying these principles to a MySQL database, introducing you to the syntax necessary to make database normalization possible. I'll also illustrate how SQL queries can then be used to mine the normalized tables for useful information. If you can't wait to learn more about how queries are used to retrieve data from normalized tables, I would suggest checking out the Devshed article MySQL Table Joins.