Design and Normalization Techniques

Database Design

Now that you have a basic understanding of databases, SQL, and MySQL, this topic begins the process of taking that knowledge deeper. The focus in this chapter, as the title states, is real-world database design. Like the work we did last week, much of the effort this week requires paper and pen—and serious thinking about what your applications will need to do.

We begin this week with thorough coverage of database normalization, a vital approach to the design process. After that, we focus on design-related concepts specific to MySQL: working with indexes, table types, language support, times, and foreign key constraints.

You'll explore steps involved in proper database design and how to make the most of MySQL. You'll also plan a couple of multi-table databases. Next week you'll learn more advanced SQL and MySQL, and put these concepts to use.

Normalization

Whenever you are working with a relational database management system such as MySQL, the first step in creating and using a database is to establish the database's structure (also called the database schema). Database design, also known as data modeling, is crucial for successful long-term management of information. Using a process called normalization, you carefully eliminate redundancies and other problems that would undermine the integrity of your database.

The techniques you will learn this week will help ensure the viability, usefulness, and reliability of your databases. The primary example we will discuss is that of a forum where users can post messages, but the principles of normalization apply to any database you might create.

Normalization was developed by an IBM researcher named E. F. Codd in the early 1970s (he also invented the relational database). A relational database is merely a collection of data, organized in a particular manner, and Dr. Codd created a series of rules called normal forms that help define that organization. Below sections discusses the first three of the normal forms, which are sufficient for most database designs.

Before you begin normalizing your database, you must define the role of the application being developed. Whether it means that you thoroughly discuss the subject with a client or figure it out for yourself, understanding how the information will be accessed dictates the modeling. Thus, this process will require paper and pen rather than the MySQL software itself (although database design is applicable to any relational database, not just MySQL).

In this example, I want to create a message board where users can post messages and other users can reply. I imagine that users will need to register, and then log in with an email address/password combination to post messages. I also expect that there could be multiple forums for different subjects. I have listed a sample row of data in the following table. The database itself will be called forum.

Table-1: Display Sample Forum Data

Item	Example
username	troutster
password	mypass
actual name	Michael Clarke
user email	email@example.com
forum	MySQL
message subject	Question about normalization
message body	I have a question about…
message date	November 2, 2017 12:20 AM

Important Tips:

- One of the best ways to determine what information should be stored in a database is to think about what questions will be asked of the database and what data would be included in the answers.
- Always err on the side of storing more information than you might need. It's easy to ignore unnecessary data but impossible to later manufacture data that was never stored in the first place.
- Normalization can be hard to learn if you fixate on the little things. Each of the normal forms is defined in a very cryptic way; even when put into layman's terms, they can still be confounding. My best advice is to focus on the big picture as you follow along. Once you've gone through normalization and seen the end result, the overall process should be clear enough.

Keys

Keys are integral to normalized databases. There are two types of keys: **primary** and **foreign**. A primary key is a unique identifier that has to abide by certain rules. They must:

- **Always have a value (they cannot be NULL)**
- Have a value that remains the same (never changes)
- Have a unique value for each record in a table

A good real-world example of a primary key is the U.S. Social Security number: everyone has a unique Social Security number, and that number never changes. Just as the Social Security number is an artificial construct used to identify people, you'll frequently find creating an arbitrary primary key for each table to be the best design practice.

The second type of key is a foreign key. Foreign keys are the representation in Table B of the primary key from Table A. If you have a cinema database with a movies table and a directors table, the primary key from directors would be linked as a foreign key in movies. You'll see better how this works as the normalization process continues.

The forum database is just a simple table as it stands (above table), but before beginning the normalization process, identify at least one primary key. The foreign keys will come in later steps.

To assign a primary key:

1. Look for any fields that meet the three tests for a primary key.
   In this example (Table-1), no column fits all the criteria for a primary key. The username and email address will be unique for each forum user but will not be unique for each record in the database because the same user could post multiple messages. The same subject could be used multiple times as well. The message body will likely be unique for each message but could change (if edited), violating one of the rules of primary keys.
2. If no logical primary key exists, invent one (Table-2).

Table-2: Display Sample Forum Data

Item	Example
message ID	325
username	troutster
password	mypass
actual name	Michael Clarke
user email	email@example.com
forum	MySQL
message subject	Question about normalization
message body	I have a question about…
message date	November 2, 2017 12:20 AM

Frequently, you will need to create a primary key because no good solution presents itself. In this example, a message ID is manufactured. When you create a primary key that has no other meaning or purpose, it's called a surrogate primary key.

Important Tips:

- As a rule of thumb, I name my primary keys using at least part of the table's name (e.g., message) and the word id. Some database developers like to add the abbreviation pk to the name as well. Some developers just use id.
- MySQL allows for only one primary key per table, although you can base a primary key on multiple columns. A multiple-column primary key means the combination of those columns must be unique and never change.
- Ideally, your primary key should always be an integer, which results in better MySQL performance.

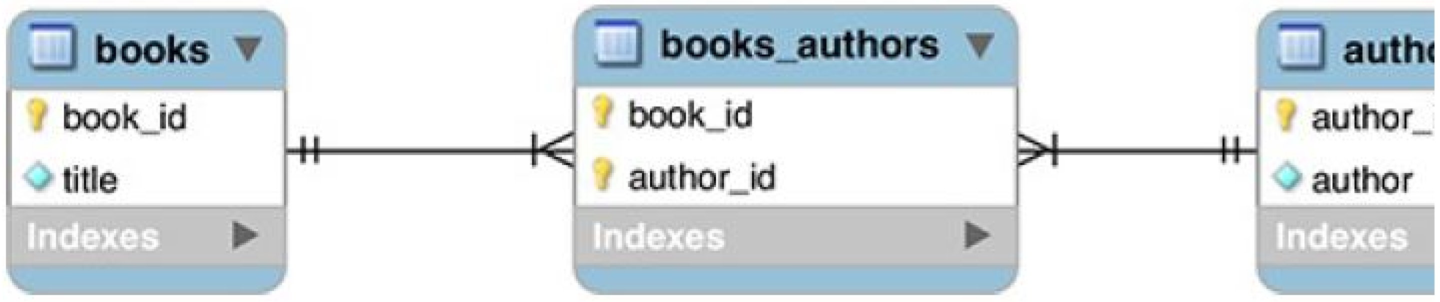
Relationships

Database relationships refer to how the data in one table relates to the data in another. There are three types of relationships between any two tables: **one-to-one**, **one-to-many**, or **many-to-many**. Two tables in a database may also be unrelated.

A relationship is one-to-one if one and only one item in Table A applies to one and only one item in Table B. For example, each U.S. citizen has only one Social Security number, and each Social Security number applies to only one U.S. citizen; no citizen can have two Social Security numbers, and no Social Security number can refer to two citizens.

A relationship is one-to-many if one item in Table A can apply to multiple items in Table B. The terms on and off will apply to many switches, but each switch can be in only one state or the other. A one-to-many relationship is the most common one between tables in normalized databases.

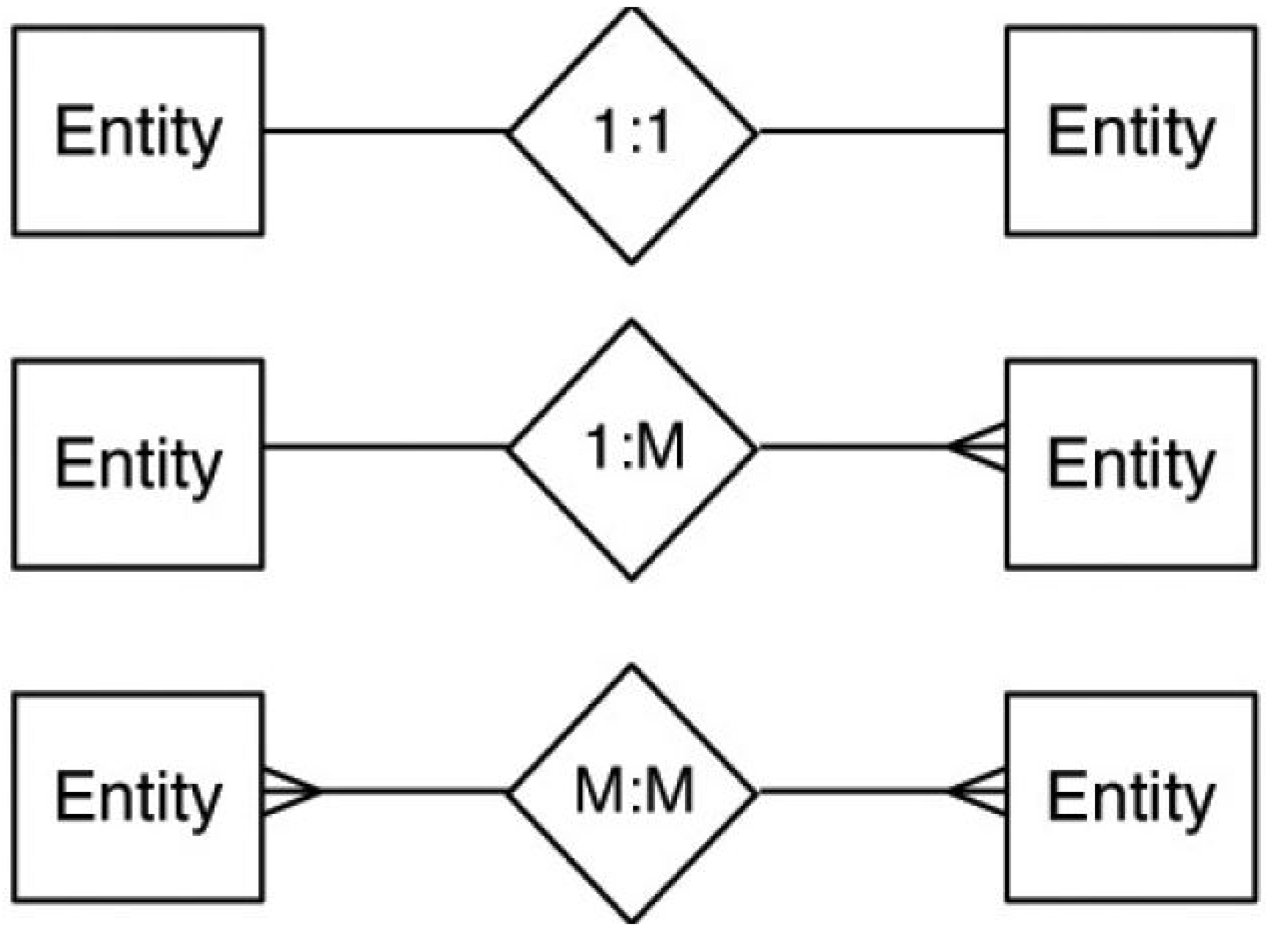
Finally, a relationship is many-to-many if multiple items in Table A can apply to multiple items in Table B. A book can be written by multiple authors, and authors can write multiple books. Although many-to-many relationships are common in the real word, you should avoid many-to-many relationships in your design because they lead to data redundancy and integrity problems. Instead of having many-to-many relationships, properly designed databases use intermediary tables that break down one many-to-many relationship into two one-to-many relationships.



A many-to-many relationship between two tables will be better represented as two one-to-many relationships those tables have with an intermediary table.

Relationships and keys work together in that a key in one table will normally relate to a key in another, as mentioned earlier.

Database modeling uses certain conventions to represent the structure of the database, which I'll follow through a series of images throughout this week. The symbols for the three types of relationships are shown below. These symbols, or variations on them, are commonly used to represent relationships in database modeling schemes.



Important Tips:

- The process of database design results in an ERD (entity-relationship diagram) or ERM (entity-relationship model). This graphical representation of a database uses shapes for tables and columns and the symbols from above figure to represent the relationships.
- The term "relational" in RDBMS actually stems from the tables, which are technically called relations.

First Normal Form (1NF)

As already stated, normalizing a database is the process of changing the database's structure according to several rules, called forms. Your database should adhere to each rule exactly, and the forms must be followed in order.

Every table in a database must have the following two qualities to be in First Normal Form (1NF):

- Each column must contain only one value (this is sometimes described as being atomic or indivisible).
- No table can have repeating groups of related data.

A table containing one field for a person's entire address (street, city, state, zip code, country) would not be 1NF compliant, because it has multiple values in one column, violating the first property. As for the second, a movies table that had columns such as actor1, actor2, actor3, and so on would fail to be 1NF compliant because of the repeating columns all listing the exact same kind of information.

To begin the normalization process, check the existing structure (Table-2) for 1NF compliance. Any columns that are not atomic should be broken into multiple columns. If a table has repeating similar columns, then those should be turned into their own, separate table.

To make a database 1NF compliant:

- Identify any field that contains multiple pieces of information.
  - Looking at Table-2, one field is not 1NF compliant: actual name. The example record contained both the first name and the last name in this one column.
  - The message date field contains a day, a month, and a year, plus a time, but subdividing past that level of specificity isn't warranted. And, MySQL can handle dates and times quite nicely using the DATETIME type.
  - Other examples of problems would be if a table used just one column for multiple phone numbers (mobile, home, work) or stored a person's multiple interests (cooking, dancing, skiing, etc.) in a single column.
- Break up any fields found in Step 1 into distinct fields (Table-3, see below).
  - To fix this problem for the current example, create separate first name and last name fields, each containing only one value.

Table-3: Forum Database, Atomic

Item	Example
message ID	325
username	troutster
password	mypass
first name	Michael
last name	Clarke
user email	email@example.com

forum	MySQL
message subject	Question about normalization
message body	I have a question about…
message date	November 2, 2017 12:20 AM

- Turn any repeating column groups into their own table.
    - The forum database doesn't have this problem currently, so to demonstrate what would be a violation, consider Table-4 below. The repeating columns—the multiple actor fields—introduce two problems. First, there's no getting around the fact that each movie will be limited to a certain number of actors when stored this way. Even if you add columns actor 1 through actor 100, there will still be that limit (of a hundred). Second, any record that doesn't have the maximum number of actors will have NULL values in those extra columns. You should generally avoid columns with NULL values in your database schema. As another concern, the actor and director columns are not atomic.

Table-4: Movies Table

Column	Value
movie ID	976
movie title	Casablanca
year released	1943
director	Michael Curtiz
actor 1	Humphrey Bogart
actor 2	Ingrid Bergman
actor 3	Peter Lorre

-
    - To fix the problems in the movies table, a second table would be created (Table-5). This table uses one row for each actor in a movie, which solves the problems mentioned in the last paragraph. The actor names are also broken up to be atomic. Notice as well that a primary key column should be added to the new table. The notion that each table has a primary key is implicit in the First Normal Form.

Table-5: Movies-Actors Table

ID	Movie	Actor First Name	Actor Last Name
1	Casablanca	Humphrey	Bogart
2	Casablanca	Ingrid	Bergman
3	Casablanca	Peter	Lorre
4	The Maltese Falcon	Humphrey	Bogart
5	The Maltese Falcon	Peter	Lorre

- Double-check that all new columns and tables created in Steps 2 and 3 pass the 1NF test.

Important Tips:

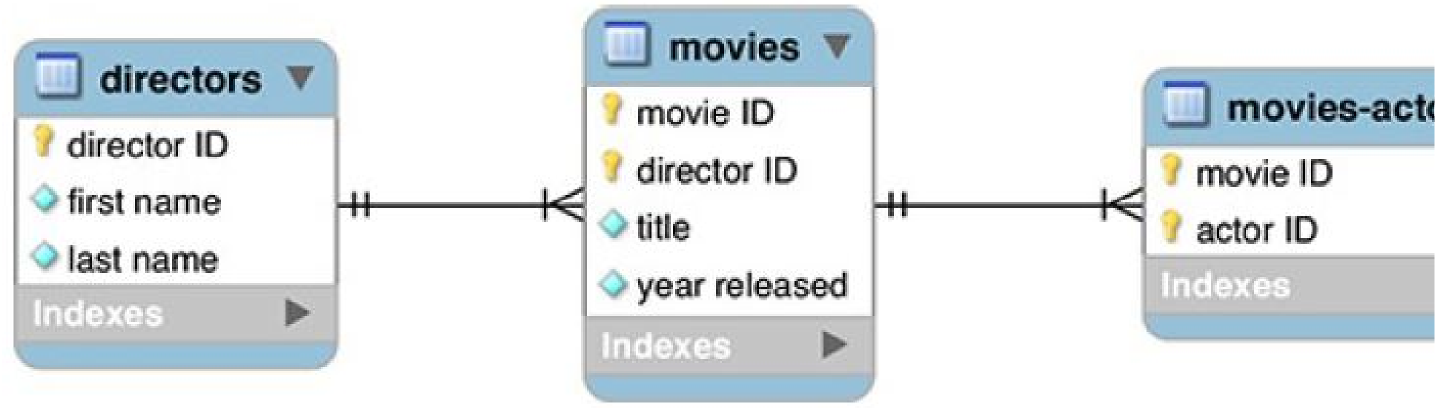
- The simplest way to think about 1NF is that this rule analyzes a table horizontally: inspect all of the columns within a single row to guarantee specificity and avoid repetition of similar data.
- Various resources will describe the normal forms in somewhat different ways, likely with much more technical jargon. What is most important is the spirit—and end result—of the normalization process, not the technical wording of the rules.

Second Normal Form (2NF)

For a database to be in Second Normal Form (2NF), the database must first already be in 1NF. You must normalize in order. Then, every column in the table that is not a foreign key must be dependent on the primary key. You can normally identify a column that violates this rule when it has non-key values that are the same in multiple rows. Such values should be stored in their own table and related back to the original table through a key.

Going back to the *cinema* example, a movies table (Table-4) would have the director Martin Scorsese listed 20+ times. This violates the 2NF rule, as the column(s) that store the directors' names would not be keys and would not be dependent on the primary key (the movie ID). The fix is to create a separate *directors* table that stores the directors' information and assigns each director a primary key. To tie the director back to the movies, the director's primary key would also be a foreign key in the *movies* table.

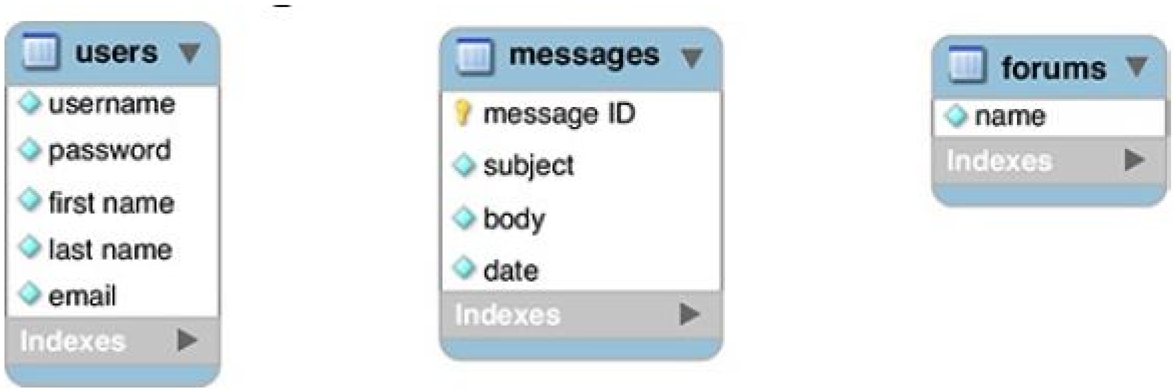
Looking at Table-5 (for actors in movies), both the movie name and the actor names are also in violation of the 2NF rule: they aren't keys and they aren't dependent on the table's primary key. In the end, the *cinema* database in this minimal form requires four tables. Each director's name, movie name, and actor's name will be stored only once, and any non-key column in a table is dependent on that table's primary key. In fact, normalization could be summarized as the process of creating more and more tables until potential redundancies have been eliminated.



Note: To make the *cinema* database 2NF compliant (given the information being represented), four tables are necessary. The directors are represented in the *movies* table through the *director ID* key; the movies are represented in the *movies-actors* table through the *movie ID* key; and the actors are represented in the *movies-actors* table through the *actor ID* key.

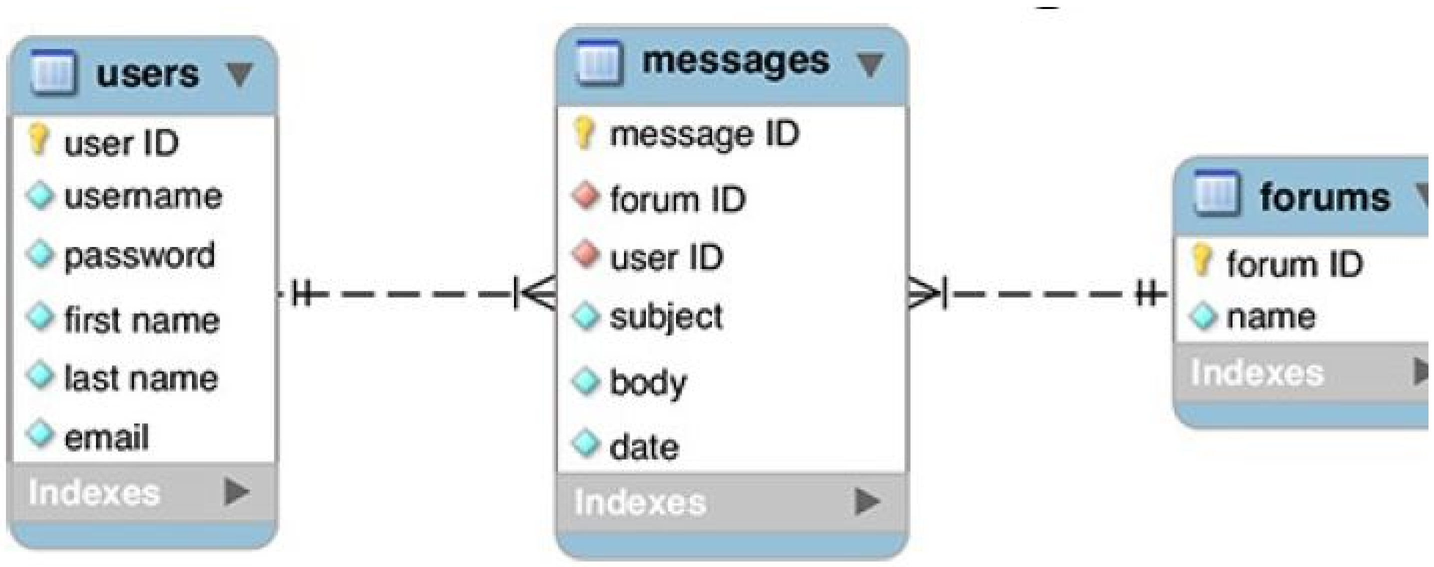
To make a database 2NF compliant:

- Identify any non-key columns that aren't dependent on the table's primary key.
  - Looking at Table-3, the username, first name, last name, email, and forum values are all non-keys (message ID is the only key column currently), and none are dependent on the message ID. Conversely, the message subject, body, and date are also non-keys, but these do depend on the message ID.
- Create new tables accordingly, to make the forum database 2NF compliant, three tables are necessary. The most logical modification for the forum database is to make three tables: users, forums, and messages. In a visual representation of the database, create a box for each table, with the table name as a header and all its columns (also called its attributes) underneath.



- Assign or create new primary keys.
  - Each table needs its own primary key. Using the techniques described earlier, ensure that each new table has a primary key. Here I've added a user ID field to the users table and a forum ID field to forums. These are both surrogate primary keys. Because the username field in the users table and the name field in the forums table must be unique for each record and must always have a value, you could have them act as the primary keys for their tables. However, this would mean that these values could never change (per the rules of primary keys) and the database would be a little slower, using text-based keys instead of numeric ones.
- Create the requisite foreign keys and indicate the relationships. To relate the three tables, add two foreign keys to the messages table, each key representing one of the other two tables.
  - The final step in achieving 2NF compliance is to incorporate foreign keys to link associated tables. Remember that a primary key in one table will often be a foreign key in another.
  - With this example, the user ID from the users table links to the user ID column in the messages table. Therefore, users has a one-to-many relationship with messages: each user can post multiple messages, but each message can be posted by only one user.

- Also, the two forum ID columns are linked, creating a one-to-many relationship between messages and forums: each message can only be in one forum, but each forum can have multiple messages.
- There is no direct relationship between the users and forums tables.



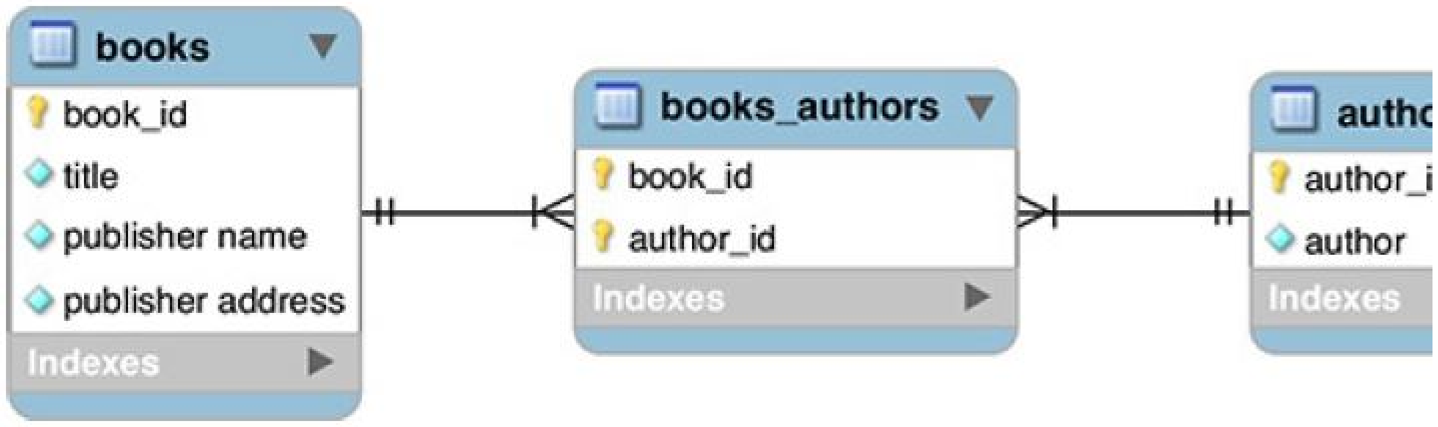
Important Tips:

- Another way to test for 2NF is to look at the relationships between tables. The ideal is to create one-to-one or one-to-many situations. Tables that have a many-to-many relationship may need to be restructured.
- Looking back at an earlier step (the movies-actors table is an intermediary table), which turns the many-to-many relationship between movies and actors into two one-to-many relationships. You can often tell a table is acting as an intermediary when all its columns are keys. In fact, in that table, the primary key could be the combination of the movie ID and the actor ID.
- A properly normalized database should never have duplicate rows in the same table: two or more rows in which the values in every non–primary key column match.
- To simplify how you conceive of the normalization process, remember that 1NF is a matter of inspecting a table horizontally, and 2NF is a vertical analysis: hunting for repeating values over multiple rows.

Third Normal Form (3NF)

A database is in Third Normal Form (3NF) if it is in 2NF and every non-key column is mutually independent. If you followed the normalization process properly to this point, you may not have 3NF issues. You would know that you have a 3NF violation if changing the value in one column would require changing the value in another. In the *forum* example thus far, there aren't any 3NF problems, but I'll explain a hypothetical situation where this rule would come into play.

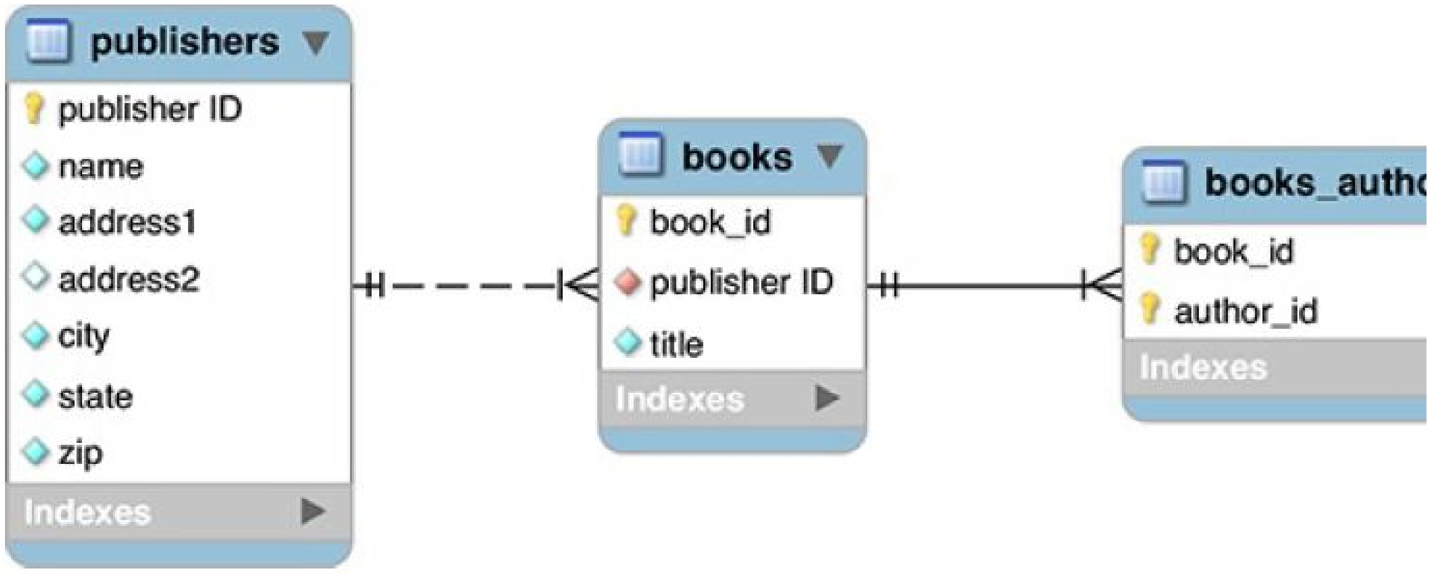
Take, as an example, a database about books. After applying the first two normal forms, you might end up with one table listing the books, another listing the authors, and a third acting as an intermediary table between books and authors, since there's a many-to-many relationship there. If the books table listed the publisher's name and address, that table would be in violation of 3NF (this database as currently designed fails the 3NF test.). The publisher's address isn't related to the book, but rather to the publisher itself. In other words, that version of the *books* table has a column that's dependent on a non-key column: the publisher's name.



As I said, the *forum* example is fine as is, but I'll outline the 3NF steps just the same, showing how to fix the books example just mentioned.

To make a database 3NF compliant:

- Identify any fields in any tables that are interdependent.
  - As just stated, what you need to look for are columns that depend more on each other than they do on the record as a whole. In the *forum* database, this isn't an issue. Just looking at the *messages* table, each *subject* will be specific to a *message ID*, each body will be specific to that *message ID*, and so forth. With a books example, the problematic fields are those in the *books* table that pertain to the publisher.
- Create new tables accordingly.
  - If you found any problematic columns in Step 1, like *address1, address2, city, state, and zip* in a *books* example, you would create a separate publishers table. (Addresses would be more complex once you factor international publishers in.)
- Assign or create new primary keys.
  - Every table must have a primary key, so add publisher ID to the new tables.
- Create the requisite foreign keys that link any of the relationships. Going with a minimal version of a hypothetical *books* database, one new table is created for storing the publisher's information. Finally, add a *publisher ID* to the books table. This effectively links each book to its publisher.

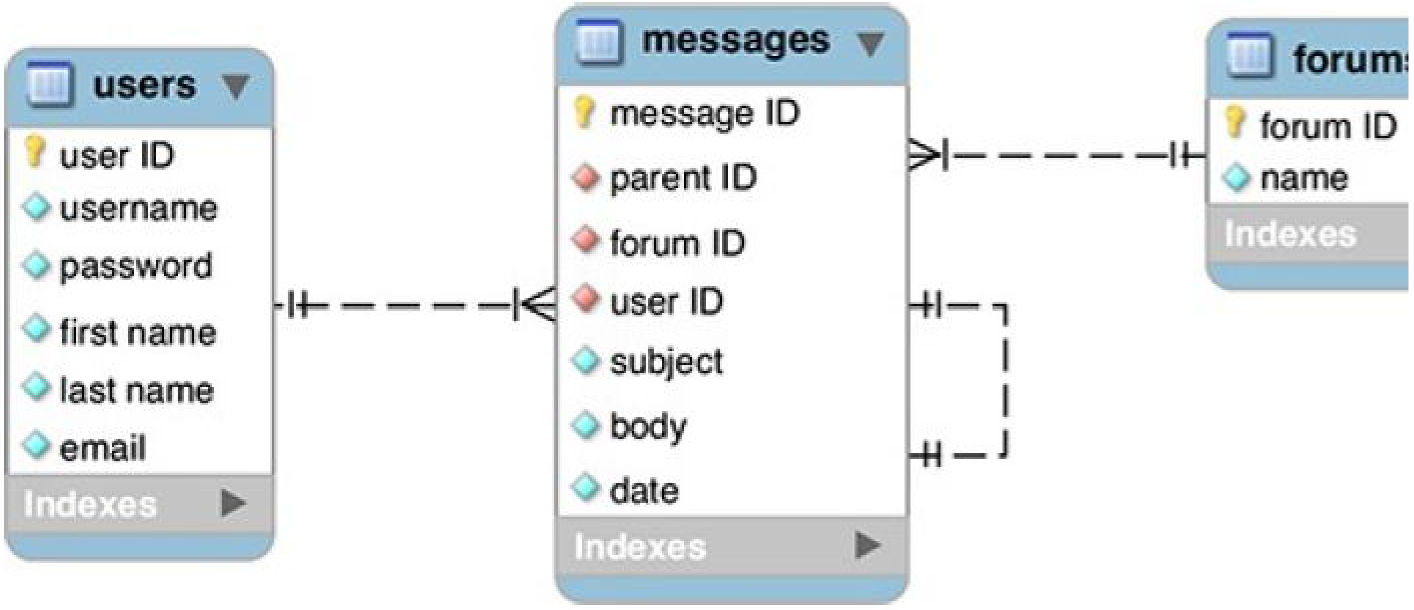


Important Tips:
Despite the existence of set rules for how to normalize a database, two different people could normalize the same example in slightly different ways. Database design does allow for personal preference and interpretations. The important thing is that a database has no clear and obvious NF violations. Any NF violation will likely lead to problems down the road.

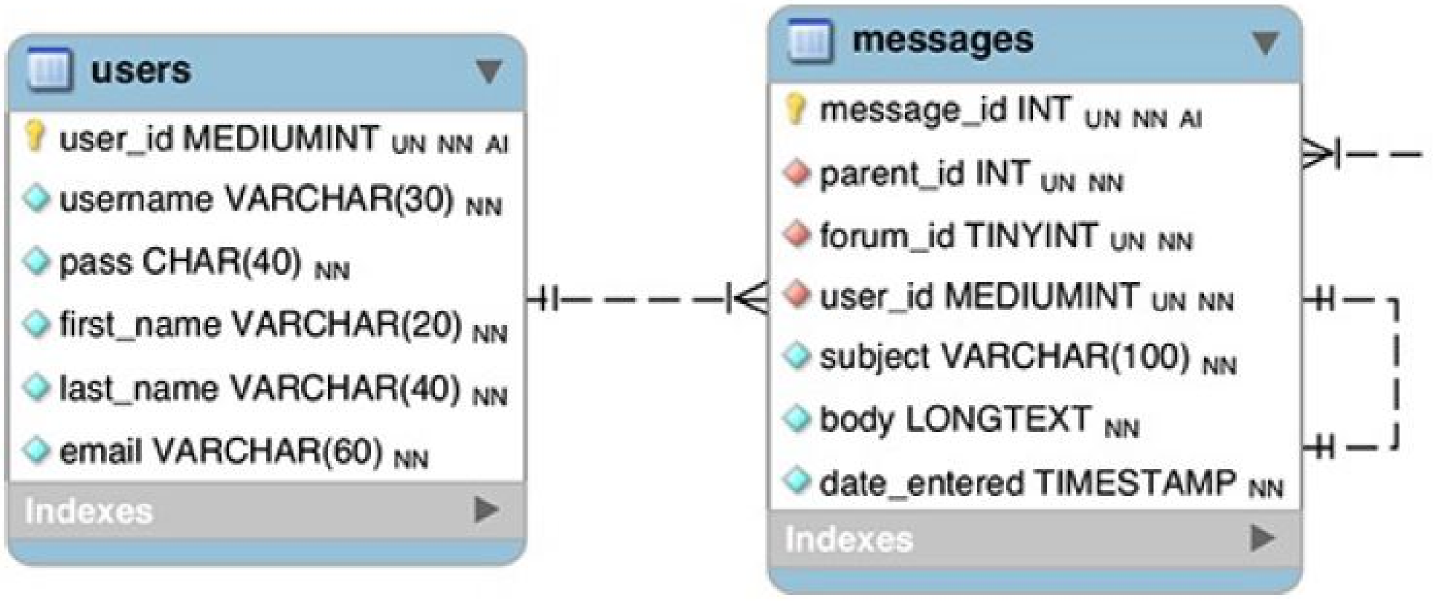
Reviewing the Design

After walking through the normalization process, it's best to review the design one more time. You want to make sure that the database stores all the data you may ever need. Often the creation of new tables, thanks to normalization, implies additional information to record. For example, although the original focus of the *cinema* database was on the movies, now that there are separate *actors* and *directors* tables, additional facts about those people could be reflected in those tables.

With that in mind, although there are many additional columns that could be added to the *forum* database, particularly regarding the user, one more field should be added to the *messages* table. Because one message might be a reply to another, some method of indicating that relationship is required. One solution is to add a *parent_id* column to *messages* (to reflect a message hierarchy, the *parent_id* column is added to *messages*). If a message is a reply, its *parent_id* value will be the *message_id* of the original message (so *message_id* is acting as a foreign key to this same table). If a message has a *parent_id* of 0, then it's a new thread, not a reply).



After making any changes to the tables, you must run through the normal forms one more time to ensure that the database is still normalized. Finally, choose the column types and names, per the concepts we learned from last week. Note that every integer column is UNSIGNED (noted as UN), the three primary key columns are also designated as AUTO_INCREMENT (noted as AI), and every column is set as NOT NULL (noted as NN). Here is the final ERD for the forums database:


users
user_id MEDIUMINT UN NN AI
username VARCHAR(30) NN
pass CHAR(40) NN
first_name VARCHAR(20) NN
last_name VARCHAR(40) NN
email VARCHAR(60) NN
Indexes
messages
message_id INT UN NN AI
parent_id INT UN NN
forum_id TINYINT UN NN
user_id MEDIUMINT UN NN
subject VARCHAR(100) NN
body LONGTEXT NN
date_entered TIMESTAMP NN
Indexes