Week 1: Discussion Forum Post 2

Instructions

Respond to the Discussion Question posted by your instructor below and then respond to at least two peer's posts (by Sunday before Midnight).

Read the beginning section of Chapter 2 called "Designing Tables" (pp. 32-34).

Define, in your own words, what it means for a database to be "normalized." Briefly discuss one possible consequence when a database is not normalized.

**Designing Tables**

So far, we have only two tables in our database, and we have not really talked about how we decide what goes in each table, except in the very informal way of doing what looked reasonable. This design, which includes tables, columns, and relationships, is more correctly called a schema.

Designing a database schema with more than a couple of dozen tables can be quite challenging if the data is complex. Database designers earn their money by being good at this difficult task. Fortunately, for relatively simple databases, with up to perhaps ten tables, it’s possible to come up with a fairly good design just by applying some basic rules of thumb, rather than needing to apply rules in a more formal way.

In this section, we are going to look at the simple sample database we are starting to build, and figure out a way to decide what tables we need.

**Understanding Some Basic Rules of Thumb**

When a database is designed, it is often normalized; that is, a set of rules is applied to ensure that data is broken down in an appropriate fashion. In Chapter 12, we will look at database design in a formal way. To get started, all we require are some simple ground rules. These rules are just to help you understand the initial database, named “*bpsimple”*, we will be using to explore SQL and PostgreSQL in this and the following chapters. We strongly suggest that you don’t just read these rules, and then dash off to design a database with 20 tables. Work your way through the book—at least until Chapter 12.

**Rule One: Break Down the Data into Columns**

The first rule is to put only one piece of information, or data attribute, in each column. This comes naturally to most people, provided they consciously think about it. In our original spreadsheet, we have already quite naturally broken down the information for each customer into different columns, so the name was separate from the ZIP code, for example.

In a spreadsheet, this rule just makes it simpler to work on the data; for example, to sort by the ZIP code. In a database, however, it is essential that the data is correctly broken down into attributes.

Why is this so important in databases? From a practical point of view, it is difficult to specify that you want the data between the twenty-ninth and thirty-fifth characters from an address column, because that happens to be where the ZIP code lives. There is bound to be some place where the rule does not hold, and you get the wrong piece of data. Another reason for the data to be correctly broken down is that all columns in a database must have the same type, unlike a spreadsheet, which is quite forgiving about the types of data in a column.

**Rule Two: Have a Unique Way of Identifying Each Row**

You will remember that when we tried to decide how to identify each row in the spreadsheet example at the beginning of this chapter, we had a problem of not being sure what would be unique. As was mentioned, this was because there was no primary key. In general, it doesn’t need to be a single column that is unique; it could be a pair of columns taken together, or occasionally even the combination of three columns that uniquely identifies a row. It is rare, and probably a mistake if you find yourself requiring more than three columns to uniquely identify a row.

In any case, there must be a way of saying, with absolute certainty, if I look at the contents of a particular column, or group of columns in this row, I know it will have a value different from all other rows in this table. If you cannot find a column, or at most a combination of three columns, that uniquely identifies each row, it’s time to add an extra column to fulfill that purpose. In our customer table, we added an extra column, “customer\_id”, to identify each row.

**Rule Three: Remove Repeating Information**

Recall that when we tried to store order information in the customer table, it looked rather untidy because of the repeating groups. For each customer, we repeated order information as many times as was required. This meant that we could never know how many columns were needed for orders. In a database, the number of columns in a table is effectively fixed by the design. So we must decide in advance how many columns we need, what type they are, and name each column before we can store any data. Never try to store repeating groups of data in a single row.

The way around this restriction is to do exactly what we did with our orders and customers data: split the data into separate tables. Then you can join the tables together when you need data from both tables. In our example, we used the column “customer\_id” to join the two tables.

More formally, what we had was a many-to-one relationship; that is, there could be many orders received from a single customer.

**Rule Four: Get the Naming Right**

This is occasionally the hardest rule to implement well. What do we call a table or column? Tables and columns should have short, meaningful names. If you cannot decide what to call something, it’s often a clue that all is not well in your table and column design.

In addition to coming up with appropriate names, most database designers have their own personal rules of thumb, or naming conventions, that they use to ensure the naming of tables and columns in a database is consistent. Don’t have some table names singular and some plural. For example, rather than naming one table office and the other departments, use office and department. If you decide on a naming rule for an id column—perhaps the table name with an appended \_id—stick to that rule. If you use abbreviations, always use them consistently. If a column in one table is a key to another table (a foreign key, as explained in Chapter 12), try to give them the same base name. In a complex database, it can get very annoying when names are not quite consistent, such as “customer\_id”, “customer\_ident”, “cust\_id”, and “cust\_no”.

Achieving this apparently simple goal of getting the names right is often surprisingly challenging, but the rewards in simplified maintenance are considerable.

Normalizing a database means designing a structure that stores the data in a logical and related way. The most common is to normalize all databases and this process has its advantages and disadvantages. Databases can hold a significant amount of information, perhaps millions or billions of data snippets.

Normalizing a database reduces its size and avoids data duplication, ensuring that each data is saved only once, but normalizing a database is a complex and difficult task. Large databases with considerable amounts of information, such as those seen in bank systems, need careful analysis and design before they can be normalized. Knowing the use of a database, such as whether it should be optimized for reading data, writing it, or both also affects the normalization process. A poorly normalized database can perform poorly and store data inefficiently.

Hi Ibrahim,

We can also say that normalizing a database ensures referential integrity in the data, referencing the relationships between data in joined tables. Without it, data in one table can lose connections to other tables where there is related data. This leads to orphaned and inconsistent data. A normalized database, with unions between tables, can prevent this from happening.

Hi Anderson,

We can also say that normalizing a database logically groups the data, application developers, who make programs that "talk" to a database, find it easier to deal with a normalized database. The data accessed is more logically organized, as are the real-world objects represented by it. This makes it easier to design, write, and change applications.