Reference — Deeper into SQL

This is a reference for the material covered in the "Deeper into SQL" lesson.

The **create table** statement

The full syntax of the **create table** statement is quite complex. See the PostgreSQL **create table** documentation for the whole thing. Here's the syntax for the form we're seeing in this lesson:

create table *table* **(** *column type [restriction]* **,** ... **)** *[rowrestriction]* **;**

There are a lot of restrictions that can be put on a column or a row. **primary key** and **references** are just two of them. See the "Examples" section of the **create table** documentation for many, many more.

Rules for normalized tables

In a normalized database, the relationships among the tables match the relationships that are really there among the data. Examples here refer to tables in Lessons 2 and 4.

1. Every row has the same number of columns.
In practice, the database system won't let us *literally* have different numbers of columns in different rows. But if we have columns that are sometimes empty (null) and sometimes not, or if we stuff multiple values into a single field, we're bending this rule.

The example to keep in mind here is the **diet** table from the zoo database. Instead of trying to stuff multiple foods for a species into a single row about that species, we separate them out. This makes it much easier to do aggregations and comparisons.

2. There is a unique *key* and everything in a row says something about the key.
The key may be one column or more than one. It may even be the whole row, as in the **diet** table. But we don't have duplicate rows in a table.

More importantly, if we are storing non-unique facts — such as people's names — we distinguish them using a unique identifier such as a serial number. This makes sure that we don't combine two people's grades or parking tickets just because they have the same name.

3. Facts that don't relate to the key belong in different tables.
The example here was the **items** table, which had items, their locations, and the location's street addresses in it. The address isn't a fact about the item; it's a fact about the location. Moving it to a separate table saves space and reduces ambiguity, and we can always reconstitute the original table using a **join**.

4. Tables shouldn't imply relationships that don't exist.
The example here was the **job_skills** table, where a single row listed one of a person's technology skills (like 'Linux') and one of their language skills (like 'French'). This made it look like their Linux knowledge was specific to French, or vice versa ... when that isn't the case in the real world. Normalizing this involved splitting the tech skills and job skills into separate tables.

The **serial** type

For more detail on the **serial** type, take a look at the last section of this page in the PostgreSQL manual.

Other subqueries

Here are some sections in the PostgreSQL documentation that discuss other forms of subqueries besides the ones discussed in this lesson:

Scalar Subqueries
Subquery Expressions
The FROM clause

Mooseball is not a real sport (yet), but you can get a roughly ball-shaped moose from Squishables.