


How to produce meaningful datasets using only SQL

 medium.freecodecamp.org/how-to-produce-meaningful-datasets-using-only-sql-394c4781a5e0

Have you ever needed to seed a database with random, yet realistic, data for the purposes of testing, demonstration, or training? It's a very common requirement, and one I have faced many times.

It doesn't take long in the development timeline before someone needs to test with "real" data. Since my role in past projects has almost exclusively been oriented around the data tier of whatever solution was being created, this responsibility rested primarily on my shoulders.

Where did I turn? Naturally, to the programming language I knew best: SQL.

It's pretty amazing — no, fascinating — what you can accomplish out of the box with SQL. Sure, there are some great libraries out there now that can assist with this task. One of my favorites, in fact, is faker.js. I have actually converted many of the examples described below to JavaScript using faker.js.

The goal of this article, however, is to demonstrate what you can accomplish with SQL alone. Before utilities like this existed, SQL was all I had to work with, and working through challenges like these helped me develop a deeper appreciation for the power of SQL. I'm hoping it does the same for you!

Conventions and Tools

My local testing environment is PostgreSQL on OSX, and all the following tips are primarily written in PG dialect. Many of these tips can be adapted to other relational database platforms or SQL dialects. If you find one in particular that you need help converting, I am happy to help, just drop me a comment.

You will see that I make fairly regular use of Common Table Expressions (CTE), also known as *WITH* clauses. I highly recommend you learn about them and use them regularly as well. CTEs are very powerful and serve

as building blocks for complex logic that ultimately make such queries very readable and maintainable.

Other than that, I have a very strong preference for SQL styling that will become pretty clear. Right-aligned and uppercase keywords, uppercase names for built-in functions, etc.

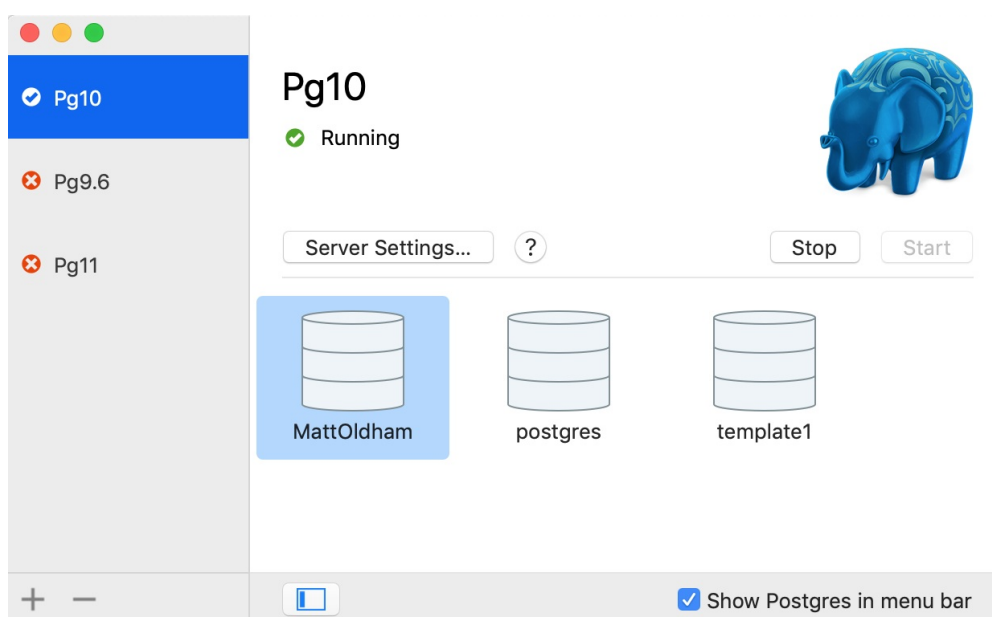
While all the code below is demonstrated using the PostgreSQL psql command line, you may find it more convenient to use a GUI client that suits your preferences. Also, you'll need to install PostgreSQL!

Note: Though you can certainly run SQL queries using tools like db-fiddle or sqlfiddle, many of my queries will actually require creating functions. Unfortunately, I was unsuccessful in attempts to create my functions using those tools. I welcome any comments with suggestions on how to make that work correctly.

For the quickest path to a local PostgreSQL environment on MacOS, here are a couple of tools I recommend:

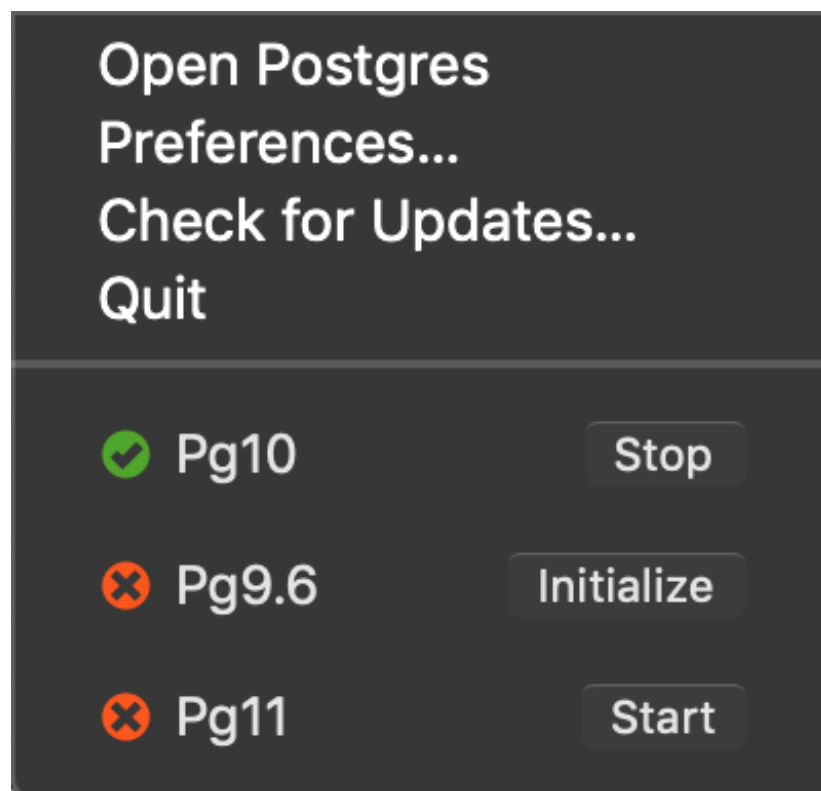
Postgres.app

Postgres.app will install PostgreSQL as a native application, and is the easiest way to get up and running quickly while also supporting multiple parallel versions.



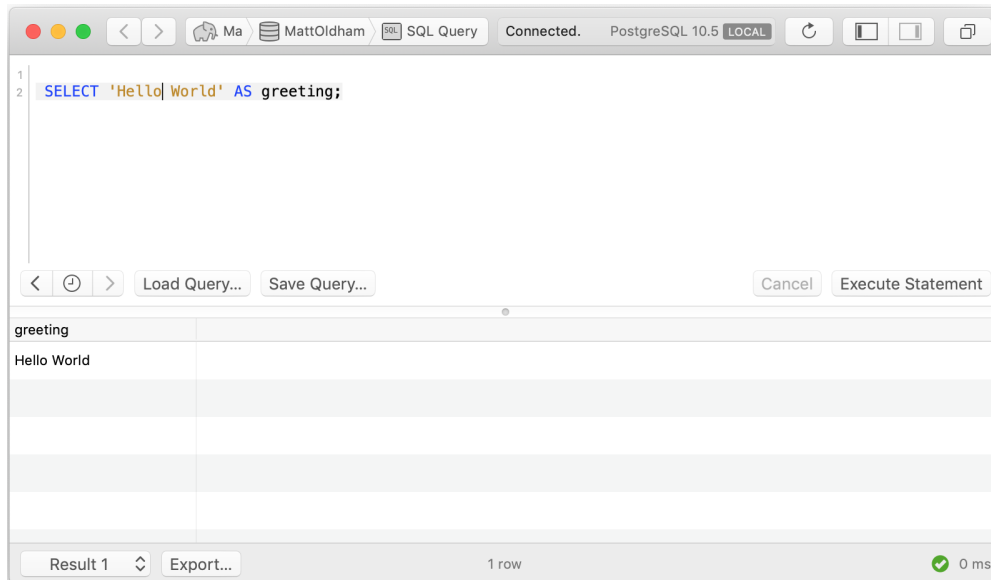
Postgres.app with multiple versions

Postgres.app is also accessible from the menu bar, allowing you to quickly start/stop PostgreSQL.



Postico

Developed by the same folks as Postgres.app, [Postico](#) is a very simple, yet elegant, UI for PostgreSQL. There are a number of other good graphical clients out there, and I personally use [DBVisualizer](#) for most of my development work. Postico, however, is a free client that makes getting up and running very easy. While there are some limitations in the trial version, there is no time limit.



Postico

With those formalities out of the way, let's dive into the main content.

Basic random value

The simplest building block for randomization in PostgreSQL is the `random` function.

Basic Random Value

According to the [PostgreSQL documentation](#) the `random` function returns a "random value in the range $0.0 \leq x < 1.0$ ".

Building on this simple function, can we generate a random value in a different specified range, like 1 and 100?

Random value in a given range

Random Value in a Given Range

Nice. To make this reusable for other scenarios that will build upon it, let's convert it to a function:

Random Number Function

Now, let's test it out:

Select Random Number

Very cool. Can we get random integers instead of decimals?

Select Random Integer

Great! Now we have something we can use in multiple scenarios. Let the real fun begin!

Random boolean values

With our new `randomNumber` function, Booleans now become very easy as well since PostgreSQL recognizes `0` as `false` and `1` as `true`.

Select Random Boolean

Let's turn this one into a function as well, which will come in handy very shortly:

Random Boolean Function

Now let's give this a trial run to make sure it's working like we expect:

Select Random Booleans

Half of the random boolean values are TRUE and half are FALSE—perfect!

Weighted random booleans

Many times you need to simulate authentic variation in your seed data, and weighting is one way to accomplish this. For example, I may want to generate boolean values for a population of data where the majority of values should be false. We're going to accomplish this with yet another function so that we can easily reference it in SQL:

Random Weighted Boolean Function

Now, let's return to my previous hypothetical scenario. If I want the false values to be the majority, I can simply pass in a lower `trueWeight`. Since our base `randomNumber` function returns values between `0` and `1`, I'll need to represent `trueWeight` as a decimal "percentage". The other part of testing this out is to generate enough values to determine if my weighting logic is working sufficiently over a larger population. For this we'll use the extremely handy PostgreSQL function `generate_series`.

Select Random Weighted Boolean

So, across 100 iterations, our weighted boolean function generated approximately 75% `false` values!

Random row(s) from a table

Another interesting facet of the `random` function is that it can be used in the `ORDER BY` clause. Let's take a look at an example by generating a sample data set (again, using `generate_series`) with and without a randomized ordering.

Randomized Ordering

Pretty cool! Now we can easily grab a random, single row from any given data set using the PostgreSQL LIMIT clause:

Random Row Selection

Random value from an enumerated list

This is one of the most common scenarios I have run across. We need randomized data, but the values should be restricted to an enumerated list that we know in advance.

For example, say we wanted to return a random value from the list `[Cyan, Magenta, Yellow, Black]`. The first challenge will be to turn our list into a data set. This is easily done by leveraging PostgreSQL ARRAYS. Let's give it a try:

Select List to Array

Our list of values has been converted to an ARRAY (notice the curly braces `{}` around the list). But how do we return the values as individual rows? PostgreSQL includes a nifty array function named `unnest` that does this for us:

Unnest Array

Building on this concept as well as several of our previous examples, we can create another function that enables us to use this approach for any enumerated list. We'll make special use of the `[]` notation for defining a

`TEXT` array, since treating all input values as `TEXT` will give us the greatest flexibility.

Random Value From List Function

Now let's test out our new function:

Select Random Values From Lists

Wow! I hope you're having as much fun as I am! 😊

Random text

This is an interesting one. We can easily generate a *random* string. Notice I did not say "meaningful". Here's one way:

Random String

We can also shorten it to the desired length (note: using the `md5` function will only generate a 32-byte string):

Random String of Specified Length

We can also manipulate the text case:

Random String of Specified Length and Case

This will only provide limited value, especially if you need longer strings that are more meaningful (e.g. person names, addresses, etc).

To address the length concern, let's start with an example I originally found posted by user `Lyndon S` on [StackOverflow](#). Here is my adaptation, which will produce a random list of 10 characters from the English alphabet:

Random English Characters

Building on that, we can use our old friend `generate_series` along with a new friend `string_agg` to generate random "words" of a specified length. Note that I have artificially limited the maximum string length to `100` characters:

Select Random Words

So we have a working solution for random “words”, but this will not suffice for use cases that require simulated “real” data. To randomly generate truly meaningful strings requires having a list of meaningful values stored in a table. I have done this before by pre-populating a “dummy names” table from which I can, for example, randomly select a random first name and last name for a person. Again, this is also where third party libraries like faker.js come in handy.

Miscellaneous use cases

Using a combination of some of the things we have learned so far (and a few new concepts), here are several interesting use cases I have come across that required randomized seed data:

Body Temperature

Phone Number

Future date in the next 12 months

Random date within the past 30 days

Random date of birth for a person under the age of 18

I hope this gives you an idea just how easy it is to accomplish some pretty powerful things using basic SQL. All the SQL Tips in this article (plus more in the future) can be found [here](#). I’d love to hear your favorite tips and clever SQL solutions, so please share them in the comments below!