LEDE PROGRAM: DATA AND DATABASES DAY 1

Creating a simple relational database

Please note: the style and layout of this page (and in rare cases some of the language) is taken from the postgreSQL documentation. don't be afraid to read or search through it when you need to. Also a helpful supplemental tutorial can be found at: http://www.postgresqltutorial.com/

Today we are going to create a very simple database in PostgreSQL. (First you will need to download the PostgresApp.) A relational database can be imagined as a set of related tables with rows and columns. Rows are the individual entries (or records). Columns are the categories (key values) for each table.

Step 1: Create That Database

Come up with a name, using only letters, numbers or the _ sign. And type in the following command and hit return.

```
$ create database dbname;
```

Step 2: Connect to the Database

The database exists! Make sure you are connected to the database. If you are not seeing the prompt "dbanme=#" you need to connect.

```
$ \c dbname
```

Step 3: Make Tables & Columns

Now let's see if there is anything in it. Use the following command:

```
$ \d
```

"Did not find any relations." Right? That's because we haven't made any tables. Before we run any command to make a table, we need to know what columns the table should have. To do this, we need to understand data types. When we make a table, each column must have a specific data type. Here is an overwhelming list of the options.

All Data Types

Name	Aliases	Description
`bigint`	`int8`	signed eight-byte integer
`bigserial`	`serial8`	autoincrementing eight-byte integer
`bit [ (n) ]`		fixed-length bit string
`bit varying [ (n) ]`	`varbit`	variable-length bit string
`boolean`	`bool`	logical Boolean (true/false)
`box`		rectangular box on a plane
`bytea`		binary data ("byte array")
`character [ (n) ]`	`char [ (n) ]`	fixed-length character string
`character varying [ (n) ]`	`varchar [ (n) ]`	variable-length character string
`cidr`		IPv4 or IPv6 network address
`circle`		circle on a plane
`date`		calendar date (year, month, day)
`double precision`	`float8`	double precision floating-point number (8 bytes)
`inet`		IPv4 or IPv6 host address
`integer`	`int, int4`	signed four-byte integer
`interval [ fields ] [ (p) ]`		time span
`json`		textual JSON data
`jsonb`		binary JSON data, decomposed

Name	Aliases	Description
`line`		infinite line on a plane
`lseg`		line segment on a plane
`macaddr`		MAC (Media Access Control) address
`money`		currency amount
`numeric [ (p, s) ]`	`decimal [ (p, s) ]`	exact numeric of selectable precision
`path`		geometric path on a plane
`pg_lsn`		PostgreSQL Log Sequence Number
`point`		geometric point on a plane
`polygon`		closed geometric path on a plane
`real`	`float4`	single precision floating-point number (4 bytes)
`smallint`	`int2`	signed two-byte integer
`smallserial`	`serial2`	autoincrementing two-byte integer
`serial`	`serial4`	autoincrementing four-byte integer
`text`		variable-length character string
`time [ (p) ] [ without time zone ]`		time of day (no time zone)
`time [ (p) ] with time zone`	`timetz`	time of day, including time zone
`timestamp [ (p) ] [ without time zone ]`		date and time (no time zone)
`timestamp [ (p) ] with time zone`	`timestamptz`	date and time, including time zone
`tsquery`		text search query
`tsvector`		text search document
`txid_snapshot`		user-level transaction ID snapshot
`uuid`		universally unique identifier
`xml`		XML data

Here is a more reasonable list of pretty much all the datatypes you might need to use at this point.

Common Data Types

Name	Aliases	Description
`boolean`	`bool`	logical Boolean (true/false)
`character [ (n) ]`	`char [ (n) ]`	fixed-length character string
`character varying [ (n) ]`	`varchar [ (n) ]`	variable-length character string
`date`		calendar date (year, month, day)
`double precision`	`float8`	double precision floating-point number (8 bytes)
`integer`	`int, int4`	signed four-byte integer
`interval [ fields ] [ (p) ]`		time span
`money`		currency amount
`numeric [ (p, s) ]`	`decimal [ (p, s) ]`	exact numeric of selectable precision
`real`	`float4`	single precision floating-point number (4 bytes)
`serial`	`serial4`	autoincrementing four-byte integer
`text`		variable-length character string
`time [ (p) ] [ without time zone ]`		time of day (no time zone)
`time [ (p) ] with time zone`	`timetz`	time of day, including time zone
`timestamp [ (p) ] [ without time zone ]`		date and time (no time zone)
`timestamp [ (p) ] with time zone`	`timestamptz`	date and time, including time zone

Now in order to create a table we need to know its columns and the datatype for each column. So figure those out first. I am making a database of "Friends" -- think about what kind of information you would want to keep and what datatypes that might entail.

```
CREATE TABLE friends (
    age integer,
    name varchar(80),
    birthday date
);
```

Step 4: Populate the Database

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is conceptually inserted one row at a time. Of course you can also insert more than one row, but there is no way to insert less than one row. Even if you know only some column values, a complete row must be created.

To create a new row, use the INSERT command. For example:

```
INSERT INTO friends VALUES (54, 'Keanu', '1964-09-02');
```

To see what's in the table just use the SELECT command (* shows everything). In the beginning, use this command over and over as you add things so you can see the changes you made.

```
SELECT * FROM friends;
```

The data values are listed in the order in which the columns appear in the table, separated by commas. The above syntax has the drawback that you need to know the order of the columns in the table. To avoid this you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

```
INSERT INTO friends (age, name, birthday) VALUES (54, 'Keanu', '1964-09-02');
INSERT INTO friends (name, age, birthday) VALUES ('Keanu', 54, '1964-09-02');
```

Many users consider it good practice to always list the column names.

You can insert multiple rows in a single command:

```
INSERT INTO friends (age, name, birthday) VALUES
    (57, 'Larry', '1961-07-30'),
    (59, 'Carrie-Anne', '1967-08-21'),
    (57, 'Hugo', '1960-04-04');
```

But wait, what if we end up getting two Hugos in there, you want to keep track of our individual people somehow. Let's add a column that assigns and numbers to each row. Use ALTER TABLE.

```
ALTER TABLE friends ADD COLUMN friend_id serial;
```

But wait again, how do I know that they are really my friends? Let's put in a true/false column.

```
ALTER TABLE friends ADD COLUMN still_friend boolean DEFAULT true;
```

And it just so happens that Larry doesn't like being called Larry. So maybe he's not a friend anymore. Using UPDATE and SET and WHERE.

```
UPDATE friends SET still_friend = FALSE WHERE name = 'Larry';
```

What if I changed his name to Lawrence?

```
UPDATE friends SET name = 'Lawrence' WHERE name = 'Larry';
```

Nope he is still not on board. And since this is a friends list, let's remove him from the whole thing.

```
DELETE FROM friends WHERE name = 'Lawrence';
```

Now I want to keep track of messages from them --and hopefully there'll be multiple messages. So instead of cluttering this table with more rows and columns let's make a new table.

```
CREATE TABLE messages (
    friend_id integer,
    time timestamp,
    message text
);
```

I insert my most recent message data.

```
INSERT INTO messages (friend_id, time, message) VALUES
    (1, now(), 'YO'),
    (1, now(), 'HEY'),
    (1, now(), 'how are you doing today?');
```

And now I can use a JOIN to bring those two tables together.

```
SELECT friends.name, messages.time, messages.message
    FROM messages
    JOIN friends ON friends.friend_id = messages.friend_id;
```