INSTANT

Short | Fast | Focused

PostgreSQL Starter

Discover how to get started using PostgreSQL with minimum hassle!

Daniel K. Lyons

[PACKT] PUBLISHING

Instant PostgreSQL Starter

Discover how to get started using PostgreSQL with minimal hassle!

Daniel K. Lyons

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant PostgreSQL Starter

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the Daniel K. Lyons, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2013

Production Reference: 1180413

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-756-3

www.packtpub.com

Credits

Author

Daniel K. Lyons

Reviewer

Reid Givens

Acquisition Editor

Erol Staveley

Commissioning Editor

Ameya Sawant

Technical Editor

Saijul Shah

Project Coordinator

Suraj Bist

Proofreader

Mario Cecere

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

Cover Image

Nitesh Thakur

About the Author

Daniel K. Lyons is a programmer and database administrator at the NRAO in Socorro, NM. He's been using PostgreSQL professionally and privately for a decade and it is one of the few technologies that he likes despite much use. In his spare time he likes to program in Haskell and Prolog.

I'd like to thank my beautiful wife Liz, whose firm but gentle patience made this book happen.

About the Reviewer

Reid Givens is the owner of Viaduct Digital, LLC, a digital media, and web development agency out of New Mexico. Reid received a degree in Interactive Media from the International Academy of Design and Technology in 2001. Reid started his career as a creative, working as a print, and web designer for small agencies. Since then, he has moved from graphic designer to creative director, advertising manager, and a few years as an organizational strategist until starting his own agency in 2007. He also speaks about marketing and the web, and has spent a few years running his own digital agency. Reid has also been known to pick up drumsticks from time to time.

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit `www.packtpub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.packtpub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.packtpub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

packtlib.packtpub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.packtpub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Instant PostgreSQL Starter

Welcome to the *Instant PostgreSQL Starter*. This book has been especially created to provide you with all the information that you need to get set up with PostgreSQL. You will learn the basics of PostgreSQL, get started with installing your first database, and discover some tips and tricks for using and running PostgreSQL.

This document contains the following sections:

So, what is PostgreSQL? finds out what PostgreSQL actually is, what you can do with it, and why it's the most advanced open-source relational database management system.

Installation will show you how to download and install PostgreSQL with the minimum fuss and then set it up so that you can use it as soon as possible. We will take a brief detour afterwards to discuss a few important settings and how they affect performance.

Quick start – Creating your first table section will show you how to perform two of the core tasks of PostgreSQL; creating databases and users. Follow the steps to create your own database and administrator account, which will be the basis of most of your work in PostgreSQL.

Top 9 features you need to know about here you will learn how to perform six tasks with the most important features of PostgreSQL. By the end of this section you will be able to insert data and perform queries, examine database structure and size, backup and restore databases, import and export data to CSV, install extensions, and store unstructured data and passwords securely.

People and places you should get to know section provides you with many useful links to the project page and forums, as well as a number of helpful articles, tutorials, blogs, and the Twitter feeds of PostgreSQL super-contributors.

So, what is PostgreSQL?

PostgreSQL is the premier open-source relational database. Long the underdog of the database world, its recent surge in popularity has come from users of other databases searching for a system with better reliability guarantees, better querying capabilities, more features, more predictable operation, or simply wanting something easier to learn, understand, and use. You will find PostgreSQL to be all of these things and more.

- **Easy-to-use**: PostgreSQL's SQL commands are consistent with each other and standard. The command-line tools all take the same arguments. The data types do not have silent truncation or other strange edge-case behavior. Surprises are rare and everything generalizes to other aspects of the system.
- **Safe**: PostgreSQL is fully transactional, including destructive structural changes. This means you can safely try anything within a transaction, even deleting data or changing table structures, secure in the knowledge that if you roll back the transaction every change you've made will be reverted. Easy backup and restore makes it trivial to clone a database across the network and experiment on a private database copy.
- **Powerful**: PostgreSQL supports many sophisticated data types out of the box, including JSON, XML, geometric objects, hierarchies, tags, and arrays. New data types and functions can be written in SQL, C, or many embedded procedural languages including Python, Perl, TCL, and others. Extensions add diverse capabilities quickly and easily, including full-text search, slow query monitoring, password encryption, and more.
- **Reliable**: PostgreSQL is very friendly to both system and database administration. All connections are simple processes and can be managed by OS utilities. It also provides Windows into what the system is doing and which connections are doing what. The standard folder layout makes it easy to control where the data are stored so you can make the most use of your partitioning. It uses the standard system startup facilities on every platform.
- **Fast**: PostgreSQL makes strategic use of indexing and query optimization to do as little work as possible. It has one of the most advanced query planners of any relational database, yet exposes its internal reasoning through the `EXPLAIN` statement, so you can find and fix the performance issues if they arise.

PostgreSQL can handle most data storage and management needs with grace and is an excellent tool to learn from as well.

How can one use PostgreSQL?

If you've never used a relational database before, take heart: it's actually based on some fairly easy concepts, applied rigorously. You can think of PostgreSQL as managing a collection of spreadsheets, each with some quantity of tables. In a relational database, lots of people can be editing these tables simultaneously and the tables have a strict structure enforced by the database management system: each column is named, belongs to a particular type, and may have additional constraints defined on the table such as must-be-unique or belonging to a particular text format. You will interact with these tables using a language called the **Structured Query Language** (**SQL**), which was designed to be easy to learn and read without sacrificing power.

If you're already using a relational database, getting started with PostgreSQL is easy, you just install PostgreSQL, learn how to create users and databases and how to connect. From there it's just a matter of finding out what the differences are between your old relational database and PostgreSQL and starting to make use of new and interesting capabilities that only PostgreSQL has.

If you're already using a non-relational system such as a NoSQL database, your road will be similar but you may also have to learn something about how to structure a relational database. You will find that with replication and by storing XML, JSON, raw strings, and using the `ltree` and `hstore` PostgreSQL extensions, you can get a lot of the benefits of your NoSQL system right from PostgreSQL.

Now that you have a little background, let's get PostgreSQL running so we can start playing with it.

Installation

In five easy steps, you can install PostgreSQL and get it set up on your system.

Step 1 – what do I need?

Unlike a lot of software, PostgreSQL's requirements are pretty slim. To get started you'll need at least 200 MB of storage and at least 512 MB of RAM. PostgreSQL uses only as much memory as it is configured to, but it will use as much disk space as it needs. Typically this will be about five times the space of a flat file with the same data.

Step 2 – downloading PostgreSQL

There are many ways to get PostgreSQL installed on your computer. We'll be using the PostgreSQL packages from EnterpriseDB, which you can download from:

`http://www.enterprisedb.com/products-services-training/pgdownload`

This is the simplest and most platform-agnostic way to get started. Be sure to get the most recent stable version, which should be the top of the list. At the time of this writing, that version is **9.2.3**.

Step 3 – installing the package

Launch the installer. The installer may offer to change system settings and prompt you for a reboot. This is normal, accept it and restart the installer after the reboot is finished.

The first page of the installation wizard

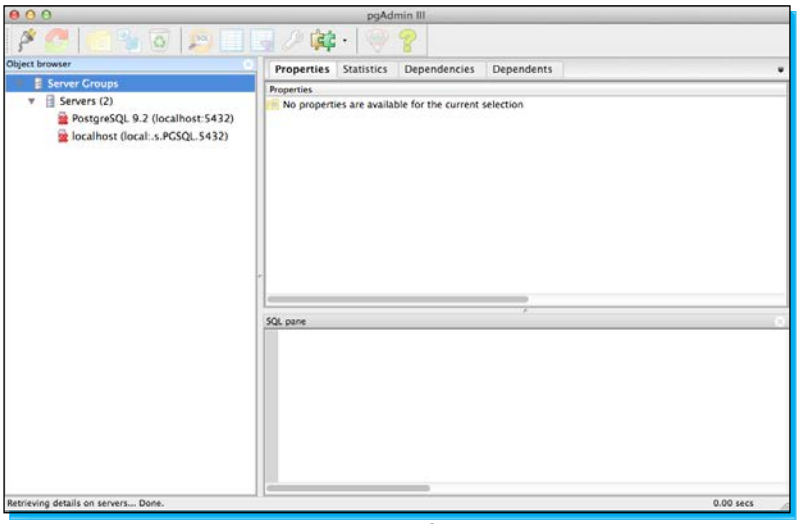
The PostgreSQL installer is intended to work for the gamut of PostgreSQL users, so most of the questions it asks are opportunities for advanced users to change a default rather than problems it needs you to solve. Since you're just getting started, there is no harm in taking the defaults.

- **Page 1**: The welcome screen. There's nothing to do here so just click **Next**.
- **Page 2**: Installation directory. Take the default location unless you want the software installed somewhere else.
- **Page 3**: Data directory. This is where PostgreSQL is going to store your actual data. The default is fine unless you want to control which disks PostgreSQL is using, which is an advanced topic for another day.
- **Page 4**: Set up password. This will be the password for the user `postgres`. Make sure to keep track of it, we'll be needing it to connect in the next section.
- **Page 5**: Server port. Take the default unless you have a very good reason not to.
- **Page 6**: Locale. Unless you have a good reason not to, choose a **UTF-8** locale. For example, if you're an American, **en_US.UTF-8** is a great choice.

Clicking **Next** from here will take you to the **Ready to install** wizard page, and one more **Next** will start the installation. Once this is complete, you'll have PostgreSQL installed on your computer!

Step 4 – connecting to PostgreSQL

Now that we have PostgreSQL installed, let's connect to it so we can get a brief taste of what life with PostgreSQL is like. The installer will have put a program called **pgAdmin III** on your computer, find it and run it and you should see this:

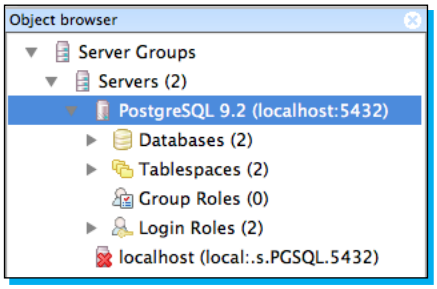


The main window of pgAdmin III

Double-click on the server named **PostgreSQL 9.2 (localhost: 5432)**. You will be prompted for your password. Use the same one you used on page 4 of the installer.

Step 5 – creating your first user and database

Once you authenticate, the **Object browser** tab will show several new options underneath **PostgreSQL 9.2 (localhost: 5432)**:

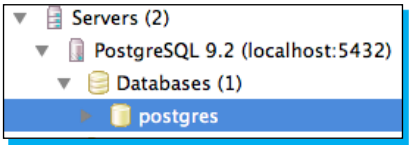


Items in the navigation tree underneath PostgreSQL 9.2 in pgAdmin III

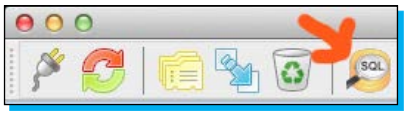
Now's a good time to familiarize yourself with this interface. We're going to be spending a lot of time in **pgAdmin III**, but most of what we'll be doing will be running SQL commands. **pgAdmin III** exposes graphical means of accomplishing many tasks, but the SQL is quick and direct.

Let's create a superuser account for you and your own database.

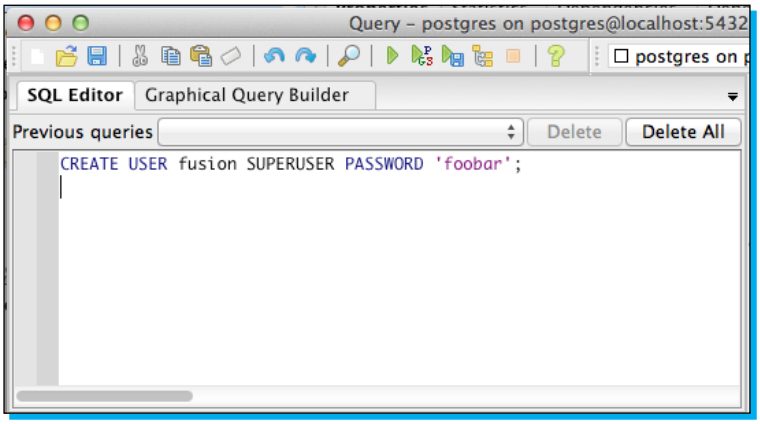
1. First, disclose the databases and click on `postgres`:



2. Next, click on the SQL button in the toolbar

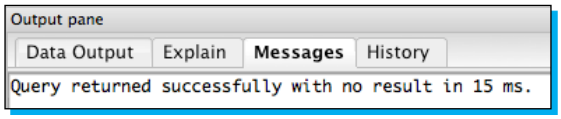


3. A new window will emerge for entering a query:



4. As you can see I've already entered a query here. This query will create a new superuser with login name `fusion`. You should choose your own name and a better password and do the same thing. When you're finished, click the green play icon to run the query:

5. If succeeded, nothing will have happened and you'll get a message from PostgreSQL back saying nothing happened:



   Most queries that cause a change in the database do not return values so there's never any reason to be alarmed because you receive no results.

6. Run the following SQL to generate a new database for this user:

```
CREATE DATABASE fusion OWNER fusion;
```

7. This will take somewhat longer but should also return with no results. Congratulations! You've successfully installed PostgreSQL, created a user account and your first database.

Quick start – creating your first table

Now that we have PostgreSQL installed, let's get started. The techniques you learn in this chapter are the bread and butter of using a database: inserting data, selecting it, updating it, and deleting it. Then to give you a glimpse of what makes the relational database relational, we'll go over a few basic joins.

For our example, let's build the backend for a basic blogging engine. All blogs have one master entity at the center of all the action: the blog post.

Step 1 – creating a table

Let's start by thinking about what sorts of things a blog post has. In fact, we just want to worry about the information a blog post has exactly such as:

- A title
- An author
- A category
- The date it was posted
- The text of the blog post

To create the table, open the query window in **pgAdmin III** just like we did at the end of the previous section, and then enter and run this SQL:

```
CREATE TABLE posts (
  title     VARCHAR PRIMARY KEY,
  author    VARCHAR NOT NULL,
  category  VARCHAR,
  posted_on DATE DEFAULT current_date,
  body      TEXT
);
```

This is called a `CREATE TABLE` statement and is part of the **Data Definition Language** (**DDL**), a subset of SQL. Don't worry about the spacing and line endings, they aren't significant, but they do make it more readable to other humans, which is important. Fundamentally every `CREATE TABLE` statement has the same structure: the `CREATE TABLE` words, the table name, and then parentheses around a bunch of column definitions. Those column definitions consist of the name, the type of the column, and whatever modifiers you want. In this table we've defined a primary key of `title`, which means the title is acting as the identifier for the whole row. We've also marked the `author` value as `NOT NULL`, which means PostgreSQL will prevent rows from being created without an author. It will also prevent rows from being modified in a way that removes the author. And we're using the `current_date` magic value for the created on column, so that if we omit that column it will default to today, whenever a row is added.

Most of these modifiers are supplying constraints, which are rules the database will enforce, and are your main tool for ensuring data validity and integrity. Nearly every table needs a primary key, which is just the column or set of columns that uniquely identify a row. In practice it's common to use an auto-incrementing integer for this purpose. Any kind of extra data added to the table to identify rows is called an artificial key, as opposed to a natural key, which is something intrinsic about the data that uniquely identifies it. While I prefer to use natural keys as much as possible, in practice there are reasons not to use them: most ORM software expects surrogate keys, it can be hard to correctly identify candidate keys, and there are legal restrictions on what you can use for the primary key that prevent you from choosing things such as social security numbers. In principle, any column you mark `UNIQUE NOT NULL` is at least a candidate key.

After you run the query, PostgreSQL creates the table, but there's nothing in it yet, so let's write our first blog post.

Step 2 – inserting some data

The first thing you want to do with a fresh table is insert some data. This is done by the running one or more aptly-named `INSERT` statements. Delete the `CREATE TABLE` text from the **query** window, then type this and run it:

```
INSERT INTO posts (title, author, category, body)
VALUES ('Hello, world!',
        'Your name here',
        'first-posts',
        'This is probably the shortest blog post ever. You''ll be glad
because INSERT isn''t a very convenient user interface.');
```

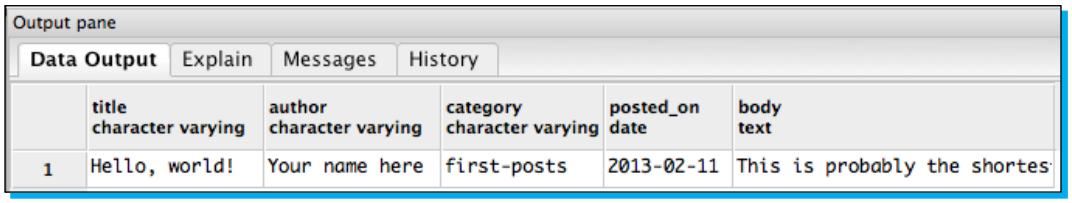
Much like the `CREATE TABLE` statement, there is a pattern at work here, which is: `INSERT INTO` table name (column names) and `VALUES` (data for each column). You don't have to list the column names if you're going to insert into all the columns, but I recommend you get in the habit anyway, because it affords you more control, and if your table grows more columns someday, you don't want to have to rewrite all your insertion code. The other important detail here is that the strings are wrapped in single quotes, and you can insert a literal single quote by putting it in the string twice.

After you run this, you should get the message: One row affected. We're making progress!

Step 3 – querying the data

Let's do the simplest query possible:

```
SELECT * FROM posts;
```



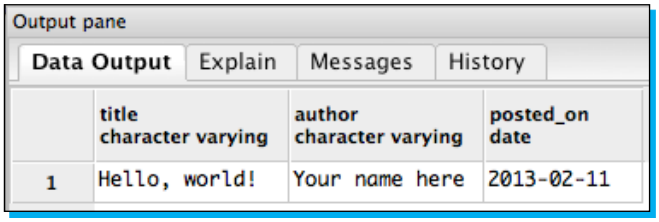
Query output

Now the resemblance to spreadsheets should be more clear: this really is just one row in a table.

`SELECT` is the only statement type in the **Data Query Language** (**DQL**) subset of SQL. It's also the most complex statement type in SQL, by a huge margin, so we're just going to try and get the flavor of it in this book.

The first and simplest way to improve on our query is to be a little choosier about which columns we want. After all, if the body of the blog post gets large you don't want PostgreSQL transmitting it back to you in every query. Let's just get the `title`, `author`, and `posted_on date` value:

```
SELECT title, author, posted_on FROM posts;
```



Query output with just three columns

This feature will become more useful as you start using joins. The Joined-up tables can easily wind up dozens to hundreds of columns, so it's helpful to just ask for the ones you need.

Step 4 – updating data

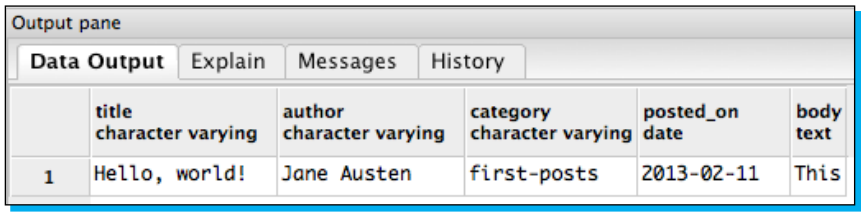
The last two operations on data are updating and deleting. The `UPDATE` statement starts out very straightforward but can get quite complex, with about as much power as the `SELECT` statement. Let's do a simple change and fix the author on the `posts` table:

```
UPDATE posts
SET author = 'Jane Austen'
WHERE title = 'Hello, world!';
```

Every `UPDATE` will have a table name and a `SET` clause. A basic update like this one will also need a `WHERE` clause with the primary key, which in our case is the title. A unique feature of PostgreSQL is the `RETURNING` clause. We can use it from an insert, update, or delete statement to return the rows that were created, modified, or deleted. So for instance, we can use it here to see the row after the modification:

```
UPDATE posts
SET author = 'Jane Austen'
WHERE title = 'Hello, world!'
RETURNING *;
```

Let's run the query and see what happens:



Update with `RETURNING` clause.

The update was successful and we see the change we made to the author column.

Step 5 – deleting data

The `DELETE` statement is pretty simple to use. Let's delete our one row:

```
DELETE FROM posts WHERE title = 'Hello, world!';
```

Congratulations! You now know how to create a table and query, insert, modify, and delete data.

Top 9 features you need to know about

To get the most out of PostgreSQL you should know about some of the many powerful features that come with it. Right out of the box there are many great capabilities available to help you solve your problems by pushing work down into the database. Any problem you let PostgreSQL solve for you is a problem you don't have to solve yourself in the application layer.

In this chapter, you'll learn how to store passwords, unstructured data, and XML, some advanced querying techniques including recursive queries, and you'll learn how to take database backups and restore them.

Let's start with one of the hardest problems to get right: storing passwords.

Storing Passwords

One of the first problems you're going to solve in most web applications is authentication. Users will need accounts and they'll need a way to log into those accounts. The way this is done conventionally on the web is by verifying a password.

The worst thing you could do is store the password in a `varchar` column like any other text. To understand why, you have to think about security and doing that means putting yourself in the shoes of a malicious attacker. You're going to do everything you can to prevent an attacker from gaining access to your database, but good security means ensuring that even if the attacker gets ahold of your database your users' credentials will remain safe. To an attacker, having access to the plaintext passwords would be an incredible win. So you mustn't allow that to happen.

The first step towards better password security is storing a hash value instead of the password. A hash value is just a number generated from text. A cryptographic hash value has two important properties: first, a small change to the input will result in a dramatically different hash value, and second, the input cannot be generated from the hash value. The hash value is one way. So instead of storing the password, we'll store the hash value of the password and when the user tries to authenticate, we'll just check that the password they entered hashes to the same value we have stored from when they set the password. If they match, the user entered the same password. This alone is not sufficient. Imagine if two users use the same password. If you just store the hash value, these two users are going to have the same hash value in the database. Worse, the attacker can generate the hash values for common passwords and go looking through the database for matching. So the next improvement over storing hashes is storing salted hashes. A salt value is just some randomly generated text. If you generate different random text for each user and append it to the password, thanks to the first property of a cryptographic hash algorithm, they'll wind up with dramatically different hashes. The attacker won't be able to simply look for the hashes of common passwords.

One unfortunate property of cryptographic hashes still remains: they're designed for speed. This is great when you're hashing a large document, but when you're dealing with passwords it's not very helpful. You don't want the attacker to be able to check millions of passwords with a particular salt value very rapidly. So ideally you want an algorithm that is computationally expensive.

Thankfully, PostgreSQL comes with an extension that handles all these minutia for you: the `pgcrypto` module. To install it, run the following SQL in **pgAdmin III** in the **query** window as before:

```
CREATE EXTENSION pgcrypto;
```

Now let's create a basic users table. For now we'll just have the username and the columns needed to store the password using `pgcrypto`. In your own applications you'll add other columns to this table, and maybe use other fields instead of a username, such as an account number or an e-mail address, but the same basic ideas will apply.

```
CREATE TABLE users (
    username varchar,
    password_hash varchar
);
```

To insert a new user, we must use `gen_salt` and `crypt` to generate the requisite values for these columns. Let's see an example, a user `bob` with password `bad password`.

```
INSERT INTO users (username, password_hash)
VALUES ('bob', crypt('bad password', gen_salt('bf')));
```

You can then authenticate a user by selecting from this table where the usernames match and the supplied password sent through crypt equals the stored hash, like this:

```
SELECT * FROM users WHERE username = 'bob' and password_hash =
crypt('bad password', password_hash);
```

You'll notice the preceding statement returns `bob`, but if you use any text other than `bad password`, you don't:

```
SELECT * FROM users WHERE username = 'bob' and password_hash =
crypt('password', password_hash);
```

This powerful module contains a number of other convenient functions. It's worth taking a few minutes to browse the documentation and see what else it can do for you. It is always better to use off-the-shelf cryptography code rather than write it yourself. For more information, see `http://www.postgresql.org/docs/9.2/static/pgcrypto.html`.

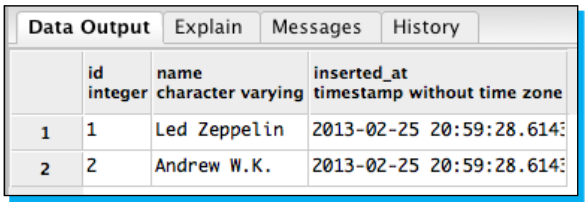
The RETURNING clause

One special feature of PostgreSQL that comes in very handy is the `RETURNING` clause. We've already seen a very basic use of it in creating your first table, so you know you can use it to receive back the row you just inserted or any part of it. This is a handy way to get back the ID value of newly-inserted rows or other information that may be computed by PostgreSQL during insertion. Let's try it with an automatically incrementing integer:

```
CREATE TABLE bands (
    id SERIAL PRIMARY KEY,
    name VARCHAR,
    inserted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO bands (name)
VALUES ('Led Zeppelin'), ('Andrew W.K.')
RETURNING *;
```

PostgreSQL will return the result of the insert as follows:



Data Output | Explain | Messages | History

	id integer	name character varying	inserted_at timestamp without time zone
1	1	Led Zeppelin	2013-02-25 20:59:28.614
2	2	Andrew W.K.	2013-02-25 20:59:28.614

Result of `INSERT` with `RETURNING`

It also works with `UPDATE` and `DELETE`, as follows:

```
UPDATE bands
SET name = 'Blue Öyster Cult'
WHERE id = 1
RETURNING *;

DELETE FROM bands
WHERE id = 1
RETURNING *;
```

These will both return the row with ID `1`. The `UPDATE` will return the row after applying the changes. You aren't limited to `RETURNING *`, you can use any SQL that goes in the column list of a `SELECT` statement there, no matter how absurd:

```
INSERT INTO bands (id, name)
VALUES (1, 'Led Zeppelin')
RETURNING id*5, length(name);
```

In practice, this is mostly used to return the serial ID value, much like MySQL's `SELECT LAST_INSERT_ID()`.

Storing unstructured data with hstore

Another contrib module gaining wider use these days is `hstore`. This module creates a new column type with the same name for storing hash tables or dictionaries of key/value strings. While it is best to design your tables according to relational database theory, there are frequently cases where you don't know ahead of time what columns you'll need to store.

The standard solution to this problem is to create a separate table for the key/value pairs. Suppose we wanted to store arbitrary metadata on posts. The tables might look like this:

```
CREATE TABLE posts (
    title VARCHAR PRIMARY KEY,
    content TEXT
);

CREATE TABLE post_metadata (
    title VARCHAR REFERENCES posts,
    key VARCHAR,
    value TEXT,
    PRIMARY KEY (title, key)
);
```

Now if we wanted to record, say, the author and publication date of the post, we could do that like this:

```
INSERT INTO posts VALUES ('Hello, World', '');
INSERT INTO post_metadata (title, key, value) VALUES
('Hello, World', 'reviewer', 'Daniel Lyons'),
('Hello, World', 'proofread on', '2012-02-21');
```

In this case, such a facility might be helpful because we may not want to write a full-fledged review system or we might want to allow a subsystem to store its data in the database without changing the structure of the database.

We can do the same thing with just one table using `hstore`. First load the `hstore` extension:

```
CREATE EXTENSION hstore;
```

Now let's `DROP` the old tables and recreate the `posts` table with an `hstore` column:

```
DROP TABLE post_metadata;
DROP TABLE posts;

CREATE TABLE posts (
```

```
  title VARCHAR PRIMARY KEY,
  content TEXT,
  metadata hstore
);
```

Now we can proceed to insert the post with the review and proofread metadata directly:

```
INSERT INTO posts (title, content, metadata) VALUES
('Hello, world', '',
   '"reviewer"     => "Daniel Lyons",
    "proofread on" => "2013-02-21"');
```

You can use the `->` operator to retrieve a value from the `hstore` column in a query. For instance, if we wanted to get all the reviewers with the posts they reviewed, we could do this:

```
SELECT title, metadata->'reviewer' as reviewer
FROM posts;
```

If we wanted to recover a table like the `post_metadata` table from before, we can do that with the function each:

```
SELECT title, (each(metadata)).key,(each(metadata)).value
FROM posts;
```

We can update the metadata using the `||` operator:

```
UPDATE posts
SET metadata = metadata ||
                 '"proofread on" => "2013-02-23"'
WHERE title = 'Hello, world';
```

Refer to the online documentation (section *F.16*) for more details about this see `http://www.postgresql.org/docs/9.2/static/hstore.html`.

Storing XML

It's also perfectly easy to store XML with PostgreSQL. There's a built-in XML type, as well as several built-in functions that permit querying and transformation of XML data. For demonstration, let's create a simple table for storing HTML documents and put a couple of sample documents into it:

```
CREATE TABLE documents (
    doc xml,
    CONSTRAINT doc_is_valid CHECK(doc IS DOCUMENT)
);
```

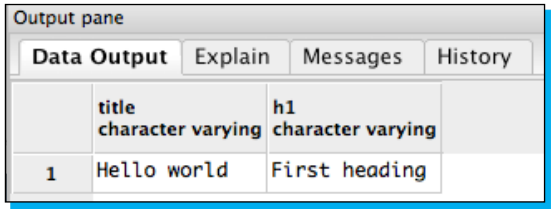
The `xml` type itself permits invalid data, so we've added a constraint to ensure that the XML in the `doc` column will always be well-formed. Let's add a simple HTML document.

```
INSERT INTO documents VALUES (
'<html>
<head><title>Hello world</title></head>
<body><h1>First heading</h1></body>
</html>');
```

We can use the `xpath` function to retrieve parts of XML documents. XPath is a small language for accessing parts of XML documents. The basic syntax is tag names separated by slashes much like a `filesystem` path. Let's see an example query:

```
SELECT
    (xpath('//title/text()', doc))[1]::varchar AS title,
    (xpath('//h1/text()', doc))[1]::varchar AS h1
FROM documents;
```

The result of this query will look like this:



The syntax of the `xpath` function is `xpath(path, document)`. Our path queries look very similar to each other: `//title/text()`, which means approximately find the text within the `<title>` element wherever it occurs. The XPath queries can always return multiple values, so we're using the array subscript syntax to obtain just the first result, which is why both columns look like `(xpath(path, doc))[1]`. Finally, we cast the result to a `varchar` value with the `column::type` syntax and name the column with the `AS` syntax. There's a lot going on in there, but we're getting exactly what we want out.

We can make this query a little simpler to run by defining a function as follows:

```
CREATE FUNCTION document_title(xml) RETURNS varchar AS $$ SELECT
(xpath('//title/text()', $1))[1]::varchar
$$ LANGUAGE 'sql';
```

Now we can do the query a little more simply:

```
SELECT document_title(doc) from documents;
```

We can even do fancier tricks with this, such as indexing on the title:

```
CREATE INDEX document_title_index
ON documents (document_title(doc) varchar_pattern_ops);
```

The `varchar_pattern_ops` code indicates to PostgreSQL that you want to perform `LIKE` queries against this index. This will make it possible for queries like this to be executed against this index:

```
SELECT * FROM documents
WHERE document_title(doc) LIKE 'Hello%';
```

Bear in mind that until the table becomes large PostgreSQL will prefer sequential scans. This is one reason it's essential to test with data similar in size and shape to production data.

About JSON. PostgreSQL 9.2 has a new built-in type for JSON documents. In theory this works like the XML type, you can create columns with the `json` type today. However, there is very little function support in this release. Your best option if you need JSON processing in PostgreSQL today is to install PLV8 to embed the v8 Javascript engine into PostgreSQL and then use Javascript stored procedures of your own devising. PLV8 can be downloaded from `http://code.google.com/p/plv8js/wiki/PLV8`.

Recursive queries

XML is one solution to dealing with hierarchical data, but it isn't the most natural for the relational database. Instead, you often wind up with nested categories, or filesystem-like folder hierarchies, or links back to older records. A popular way to structure a relational database for data with this shape is using a self reference, a link back to a `parent` object in the same table. For instance, you might want to model categories that can have subcategories with arbitrary nesting. A simple table might look like this:

```
CREATE TABLE categories (
    id SERIAL PRIMARY KEY,
    name VARCHAR,
    parent_id INTEGER REFERENCES categories
);
```

What makes this structure recursive is that self-referencing `parent_id` column, which refers to the table we're defining from within its own definition. We would treat categories with a `NULL` value for the `parent_id` column as top-level categories, categories that do not belong within any other category.

To get a feel for the kind of data, let's put a few categories in there for an online retailer. Say we sell shirts and books, and books we further divide into fiction and non-fiction, and then we'll put programming books inside non-fiction. It might look like the following:

```
INSERT INTO categories (id, name, parent_id) VALUES
(1, 'Shirts', NULL),
(2, 'Books', NULL),
(3, 'Fiction', 2),
(4, 'Non-fiction', 2),
(5, 'Programming', 4);
```

Usually you won't put these in manually like this, but you can see that `Fiction` and `Non-fiction` are children of the `Books` category because their `parent_id` values are `2`, the value for `Books` is `NULL` and `Programming` has the `parent_id` value as `4`, which is the `id` value for `Non-fiction`.

Suppose you want to make navigation breadcrumbs given a category. You need to look up that category and then you need to look up the `parent` category. You need to do this until the `parent` category is `NULL`. In a procedural pseudocode, the process might look like the following:

```
def write_breadcrumbs(category):
    row = getOne("SELECT * FROM categories \
                  WHERE id = ?", category)

    while row['parent_id'] != NULL:
        write(row['name'])
        write(' > ')

        row = getOne("SELECT * FROM categories \
                      WHERE id = ?", row['parent_id'])
```

This kind of solution leads to the *N+1 query* problem. There's an action you want to take for a particular value, but to take that action you have to run an arbitrary number of separate queries. Recursive queries in PostgreSQL provide you with a way to have the database do the heavy lifting instead. Because we'll be using recursion in the SQL, let's first see what a recursion formulation would look like in our pseudocode:

```
def write_breadcrumbs(category):
    row = getOne("SELECT * FROM categories \
                  WHERE id = ?", category)
    write(row['name'])
    if row['parent_id'] != NULL:
        write(' > ')
        write_breadcrumbs(row['parent_id'])
```

It's debatable whether this is better code; it's shorter, and it has fewer bugs, but it also might expose the developer to the possibility of stack overflows. Recursive functions always have some similar structure though, some number of base cases that do not call the function recursively and some number of inductive cases that work by calling the same function again on slightly different data. The final destination will be one of the base cases. The inductive cases will peel off a small piece of the problem and solve it, then delegate the rest of the work to an invocation of the same function.

PostgreSQL's recursive query support works with something called the **common table expressions** (**CTEs**). The idea is to make a named alias for a query. We won't delve into the details too much here, but all recursive queries will have the same basic structure:

```
WITH RECURSIVE recursive-query-name AS (
    SELECT <base-case> FROM table
    UNION ALL
    SELECT <inductive-case> FROM table
    JOIN <recursive-query-name> ON )
SELECT * FROM <recursive-query-name>;
```

For an example, let's get all the categories above `Programming`. The base case will be the `Programming` category itself. The inductive case will be to find the `parent` of a category we've already seen:

```
WITH RECURSIVE programming_parents AS (
    SELECT * FROM categories WHERE id = 5
    UNION ALL
    SELECT categories.* FROM categories
    JOIN programming_parents ON
         programming_parents.parent_id = categories.id)
SELECT * FROM programming_parents;
```

This works as we'd hope:

Data Output | Explain | Messages | History

	id integer	name character varying	parent_id integer
1	5	Programming	4
2	4	Non-fiction	2
3	2	Books	

Output of a simple recursive query

Without using this trick we'd have to do three separate queries to get this information, but with the trick it will always take one query no matter how deeply nested the categories are.

We can also go in the other direction and build up something like a tree underneath a category by searching for categories that have a category we've already seen as a `parent` category. We can make the hierarchy more explicit by building up a path as we go:

```
WITH RECURSIVE all_categories AS (
    SELECT *, name as path FROM categories WHERE parent_id IS NULL
    UNION ALL
    SELECT c.*, p.path || '/' || c.name
    FROM categories AS c
    JOIN all_categories p ON p.id = c.parent_id)
SELECT * FROM all_categories;
```

Data Output | Explain | Messages | History

	id integer	name character varying	parent_id integer	path character varying
1	1	Shirts		Shirts
2	2	Books		Books
3	3	Fiction	2	Books/Fiction
4	4	Non-fiction	2	Books/Non-fiction
5	5	Programming	4	Books/Non-fiction/Programming

Finding all the categories with their path

We can be more discriminating with the search by looking for a particular category as the starting value in the base case:

```
WITH RECURSIVE all_categories AS (
    SELECT *, name as path FROM categories WHERE id = 2
    UNION ALL
    SELECT c.*, p.path || '/' || c.name
    FROM categories AS c
    JOIN all_categories p ON p.id = c.parent_id)
SELECT * FROM all_categories;
```

Data Output | Explain | Messages | History

	id integer	name character varying	parent_id integer	path character varying
1	2	Books		Books
2	3	Fiction	2	Books/Fiction
3	4	Non-fiction	2	Books/Non-fiction
4	5	Programming	4	Books/Non-fiction/Programming

The category tree rooted at Books

This is hugely useful with hierarchical data.

Full-text search

Often you will have large text documents in your database and want to search for documents by their content. If your documents are small and your table has few rows, the ordinary `LIKE` queries are sufficient, but once your documents get bigger or you want relevance ranking, you will start to want a better solution. For this, PostgreSQL has built-in support for full-text searching. Let's make a simple table and test it out:

```
CREATE TABLE posts (
    title varchar primary key,
    content text
);

INSERT INTO posts (title, content) VALUES
('Hello, world', 'This is a test of full-text searching!'),
('Greetings', 'Posts could be blog articles, or ads
or anything else.');
```

Now a basic non-full-text search would be to use the `LIKE` operator, which could look like this:

```
SELECT title FROM posts WHERE content LIKE '%article%';
```

This will return the second row's title, `Greetings`. This approach is nice, but very limited, after all, if we had used `%post%` nothing would have been returned because of a difference in case.

The text search facility is a bit more complex. Before the search can be performed we have to convert the document into `tsvector` and the query into `tsquery`. Then we must use the `@@` operator to perform a `tsquery` against a `tsvector`.

We can convert text into `tsvector` using the `to_tsvector` function and we can convert a query into `tsquery` using `plainto_tsquery` like so:

```
SELECT title FROM posts
WHERE to_tsvector(content) @@ plainto_tsquery('post');
```

This query will return the second row, because under the covers `to_tsvector` does more than change the types, it also normalizes the text by removing suffixes such as '-s' and '-ing'. Along similar lines, `plainto_tsquery` does more than converting a string into a query, it also insinuates some special operators that the full-text search system can understand, translating a string like 'blog posts' into 'blog & post', which will only match documents that possess both words. If you want to use the extended search language directly, you can do so, just use `to_tsquery` instead, and refer to section 8.11.2 of the PostgreSQL manual, which describes the text search operators `&`, `|`, and `!`, at `http://www.postgresql.org/docs/9.2/static/datatype-textsearch.html#DATATYPE-TSQUERY`.

As nice as this is, it's pretty inefficient, because PostgreSQL isn't using any indexes on this data, so it must regenerate `tsvector` for each row, on every select. We can speed things up considerably by indexing `tsvector`:

```
CREATE INDEX posts_content_search_index ON posts
USING gin(to_tsvector('english', content));
```

The `english` argument there tells us to_`tsvector` what language we want it to use. `gin` is a type of index (**Generalized Inverted Index** (**GIN**)). The manual details when you would choose GIN and when you would choose **Generalized Search Table** (**GIST**)indexes, but the rule of thumb for text search is that documents, which are read more often than written should use GIN indexes while frequently-updated documents should use GIST.

Now that we have this index in place, performing full text searches becomes very efficient. Next, one might want to know something about the quality of the search results. You can find out by using the `ts_rank` or `ts_rank_cd` functions:

```
SELECT title, ts_rank_cd(to_tsvector(content), query)
FROM posts, plainto_tsquery('post') AS query
WHERE to_tsvector('english', content) @@ query
ORDER BY ts_rank_cd(to_tsvector(content), query)
```

The result here shows that we have a rank code of 0.1, which is low, but with such simple documents it's unlikely we could get it to be higher.

One more improvement would be to use the title in the vector, and give it a weight to reflect its greater importance. You can construct such vectors quite easily, though the syntax is somewhat odd: `setweight(tsvector, char)`. This function assigns a weight to the enclosed `tsvector`. To make good use of it, we have to combine several `tsvectors` with different weights. The regular string concatenation operator || lets us do exactly that. Let's see it in action by recreating the index, this time including the title and assigning the title and document different weights:

```
DROP INDEX posts_content_search_index;
CREATE INDEX posts_content_search_index ON posts
USING gin((setweight(to_tsvector('english', title), 'A')
      ||(setweight(to_tsvector('english', content), 'B'))
));
```

We've decided to give the `title` value a weight of `A`, the highest, and the content `B`, the second highest. In practice you might have other levels for tags or keywords, the author, the category, or whatever else you might imagine. If we were indexing a products table, the product name could be weight `A`, the manufacturer could be weight `B`, keywords weight `C`, and the description or reviews weight `D`. It's up to you to decide what the different levels are. The available levels are `A` through `D`.

These nested functions are getting unwieldy to write, so let's make a view to simplify things for us:

```
CREATE VIEW posts_fts AS
SELECT *,
  setweight(to_tsvector('english', title), 'A') ||
  setweight(to_tsvector('english', content), 'B')
    as vector
FROM posts
```

Now we can make the query look a lot tidier:

```
SELECT title, ts_rank_cd(vector, query)
FROM posts_fts, plainto_tsquery('hello') as query
WHERE vector @@ query
```

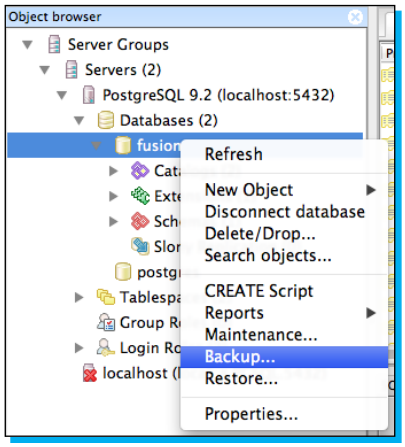
Also, notice the rank has gone from `0.1` to `1`, because `hello` is in the `title` value, which has rank `A`.

Backup and restore

It's important to be comfortable with database backup and restore procedures. The only way to conquer nerves about these procedures is to do them frequently enough that they become second nature. I'm going to show you how to do these tasks through **pgAdmin III** on your desktop and I'll include the command-line equivalents so that you can see how to perform these tasks in a headless Unix server environment.

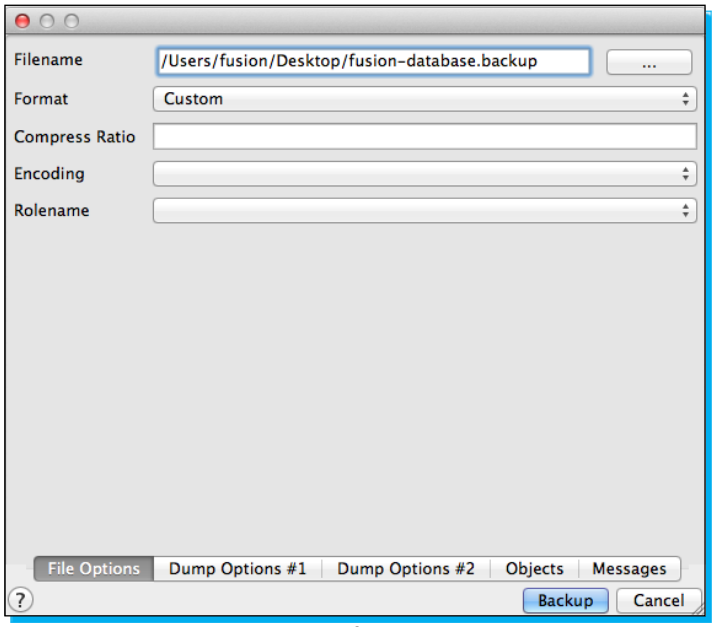
If you've been following along there should be some tables with a small amount of data in them on your server.

In **pgAdmin III**, right-click on the database and select **Backup...**:



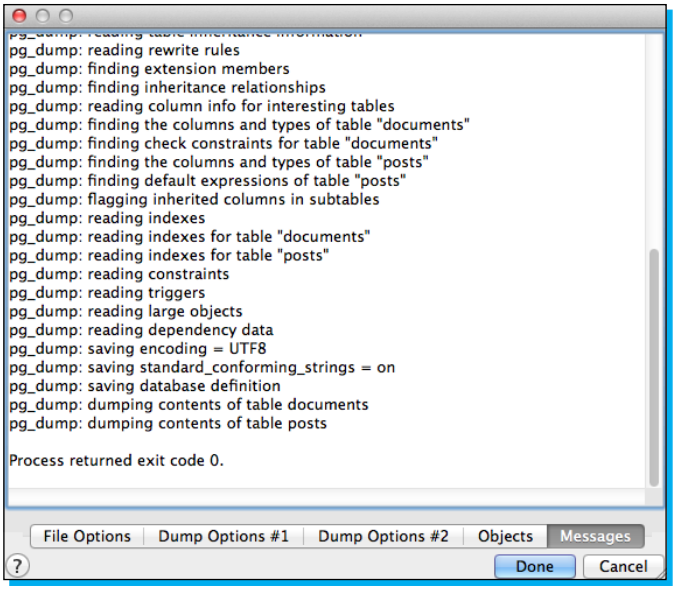
The backup menu item

You'll see a new window appear as follows:



The backup configuration dialog

As I've done, choose the **Custom** value in **Format** and ensure that your filename ends in `.backup` and then click **Backup**. The dialog will change into this:

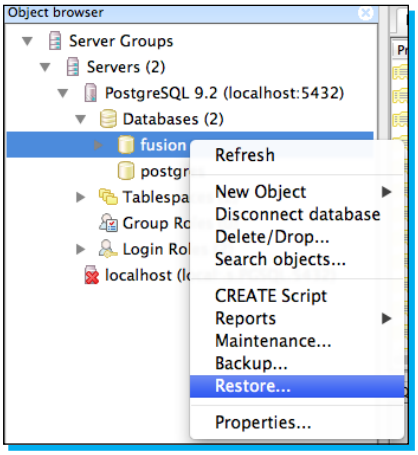


Output from pg_dump

The key here is the last line: **Process returned exit code 0**. If you get a different response, read through the dialog and see what happened, chances are, you ran out of disk space. Otherwise the backup worked fine and you have the entire contents of that database in a file that can be loaded back into this or any other PostgreSQL server (assuming it is the same version or newer).

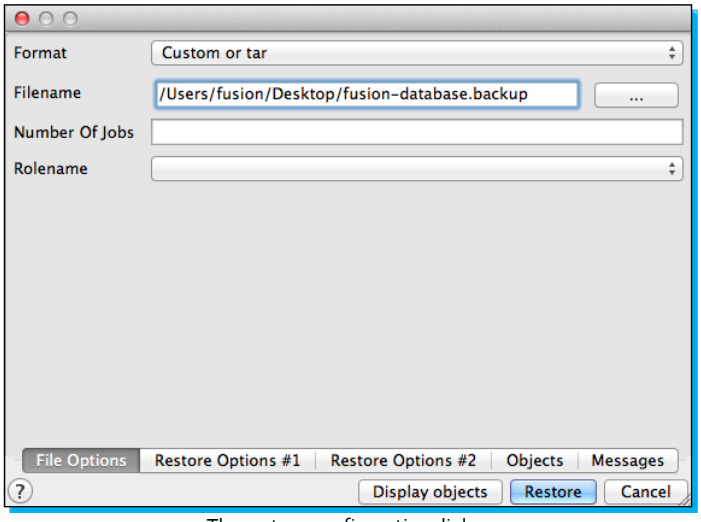
Now let's delete this database and re-create it. Right-click on the database name and choose **Delete/Drop...**. It will prompt you to proceed. Go ahead and delete the database, then create a new one by right-clicking on the databases icon in the tree. All you have to supply is the name.

Now right-click on the database in the tree and choose **Restore...**:



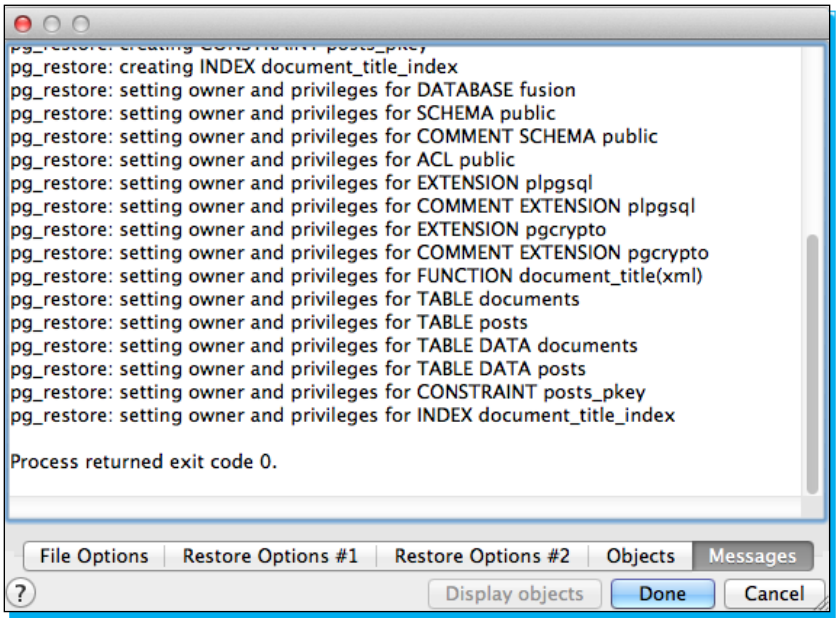
The Restore... command

All you should need to specify in the dialog is the path to the backup file, as follows:



The restore configuration dialog

Proceed with the restore and you should get similar output to the backup, again ending in **Process returned exit code 0** as follows:



Output from `pg_restore`

That's it, you've backed up and restored your database.

From Unix, what you've just done is equivalent to this sequence of commands:

```
$ pg_dump -Fc -f fusion-database.backup -U fusion fusion
$ dropdb fusion
$ createdb -O fusion fusion
$ pg_restore -d fusion fusion-backup.backup
```

Examining database size

One question you're bound to ask yourself sooner or later is, how big are these tables? PostgreSQL has some convenient functions you can use to find out just how big various things are in the backend. For instance, to see the total size of a database, you can run `pg_database_size`. I have wrapped all these examples with `pg_size_pretty`, which takes a number of bytes and returns a human-friendly string with appropriate units (kilobytes, megabytes, and so on):

```
SELECT pg_size_pretty(pg_database_size('fusion'));
```

This returned 6629 KB for me. To see just the size of a table, you can run:

```
SELECT pg_size_pretty(pg_relation_size('posts'));
```

This returned just 4 KB for me. Note that this is just the raw table size and doesn't include indexes. To get the whole picture, use `pg_total_relation_size`:

```
SELECT pg_size_pretty(pg_total_relation_size('posts'));
```

This returned 32 KB, a marked increase in size just from including indexes. Keeping your index sizes reasonable is a big part of database administration. The trick is choosing columns to index by knowing which queries need to be optimized. There's a lot of literature on the subject, but one of the best ways to learn more is to read Markus Winand's excellent tutorial on the subject, *Use the Index, Luke!* at `http://use-the-index-luke.com/`.

PostgreSQL provides quite a few helpful built-in functions like these. See section *9.26. System Administration Functions* of the documentation for more: `http://www.postgresql.org/docs/9.2/static/functions-admin.html`.

Examining query performance

One of the most amazing things about relational databases is what goes on under the hood when you perform a query. In broad strokes, the database first parses the query, checks to make sure it is sensible, and then dreams up many different ways to perform the query. It then ranks these query plans according to estimated cost and chooses the cheapest one to perform. This process is happening transparently with every query you issue. So what do you do when you realize a query is performing poorly?

The answer is to put `EXPLAIN` at the start of your `SELECT` statement. This tells PostgreSQL that rather than executing your query you want it to tell you what it would do in order to execute it. If you use `EXPLAIN ANALYZE` instead it will also run the query and include actual timing data. Let's run the following query on our `posts` table and see what happens:

```
EXPLAIN ANALYZE SELECT * FROM posts;
```

PostgreSQL responds with the following text:

```
 Seq Scan on posts  (cost=0.00..16.40 rows=640 width=96) (actual
time=21.817..21.819 rows=1 loops=1)
 Total runtime: 21.883 ms
```

This is not a very interesting query, but you can see PostgreSQL has told us that it's going to use a sequential scan on the table. Depending on the size of the table and whether or not it is cached in memory this may or may not be a problem, but without looking for specific values it would be hard to optimize. Let's get a more interesting query plan using one of our recursive queries:

```
EXPLAIN ANALYZE
WITH RECURSIVE all_categories AS (
```

```
    SELECT *, name as path FROM categories WHERE id = 2
    UNION ALL
    SELECT c.*, p.path || '/' || c.name
    FROM categories AS c
    JOIN all_categories p ON p.id = c.parent_id)
SELECT * FROM all_categories;
```

The query plan we get back looks like this:

```
 CTE Scan on all_categories  (cost=291.34..302.96 rows=581 width=72)
(actual time=24.356..24.600 rows=4 loops=1)
    CTE all_categories
        ->  Recursive Union  (cost=0.00..291.34 rows=581 width=72)
        (actual time=24.350..24.585 rows=4 loops=1)
            ->  Index Scan using categories_pkey on categories
            (cost=0.00..8.27 rows=1 width=40) (actual
            time=24.344..24.347 rows=1 loops=1)
                Index Cond: (id = 2)
                ->  Subquery Scan on "*SELECT* 2"
                (cost=0.33..27.72 rows=58 width=72) (actual
                time=0.062..0.066 rows=1 loops=3)
                ->  Hash Join  (cost=0.33..27.14 rows=58 width=72)
                (actual time=0.056..0.059 rows=1 loops=3)
                    Hash Cond: (c.parent_id = p.id)
                    ->  Seq Scan on categories c
                    (cost=0.00..21.60 rows=1160 width=40) (actual
                    time=0.005..0.007 rows=5 loops=3)
                        ->  Hash  (cost=0.20..0.20 rows=10
                        width=36) (actual time=0.008..0.008 rows=1
                        loops=3)
                        Buckets: 1024  Batches: 1  Memory
                        Usage: 1kB
                        ->  WorkTable Scan on all_categories p
                        (cost=0.00..0.20 rows=10 width=36) (actual
                        time=0.002..0.003 rows=1 loops=3)
                         Total runtime: 24.758 ms
```

Despite the complexity of the plan, it's still executing pretty quickly. Note that a complex plan isn't necessarily a bad thing. If you are doing searches for particular values and seeing both a performance problem and sequential scan, it probably means you're missing an index, but the query plan will probably be very simple. In this case, the performance probably has more to do with the table being small enough to fit in memory.

One the most important things to know about optimizing database performance is that the database will behave differently depending on its configuration and the size of the tables being queried. You may have tables with the same structure on your machine and on production, but if your production database has 4 gigabytes of data in this table and your laptop has 10 megabytes, you might see radically different query plans. It's essential when debugging database performance problems that your databases be configured as close to identical as possible and have as close to identical data. Different hardware, different configuration files, different versions of PostgreSQL, or different data can all cause large differences in performance.

If you want to know more about PostgreSQL performance optimization with the `EXPLAIN` statement, consult section *14.1. Using EXPLAIN* of the official PostgreSQL documentation, available here at `http://www.postgresql.org/docs/9.2/static/using-explain.html`.

People and places you should get to know

We sure have come a long way in a short time! There are a lot of things we haven't gone over yet, but fortunately PostgreSQL has some of the most complete documentation of any open source project, and also a large and well-informed community who are happy to help new and experienced users alike.

Official sites

By far the greatest resource in your arsenal is the official PostgreSQL documentation, available here:

```
http://www.postgresql.org/docs/
```

That page has links for several versions (*8.4* through *9.2* currently), both with and without commentary. The documentation itself is quite large. I recommend you glance at the table of contents and just get a feel for the scope of material covered. For deeper study it would be wise to examine sections *2* and *3* on SQL and administration. The SQL commands section is especially helpful. Naturally, Google searches tend to direct you to the documentation. When you do get to the documentation from a search result, make sure to follow one of the version links at the top of the page to get to the right version for you after a search, Google seems to prefer older versions of the documentation for some keywords.

There is also a wiki community available at:

```
http://wiki.postgresql.org/
```

The wiki has a lot of soft documentation such as tutorials, migration advice from, and comparisons with other database systems. It also hosts the official FAQ and a curated list of books and manuals.

Support

PostgreSQL has both free, community-provided support as well as several companies selling commercial support. Joining the pgsql-general mailing list is recommended for general questions and help. There are many other mailing lists for more specialized discussion, you can see a complete list here:

```
http://www.postgresql.org/community/lists/
```

You can also get help on *stackoverflow* and *Database Administrators*' Stack Exchange:

```
http://www.stackoverflow.com
```

```
http://dba.stackexchange.com
```

stackoverflow is usually more appropriate for SQL querying help and other programming-oriented questions while the other is better for query performance optimization and database configuration questions. If you ask your question just right, you'll get an answer in minutes.

Community

Planet PostgreSQL is a blog aggregator at `http://planet.postgresql.org/`. Subscribing to it will get you a variety of articles about PostgreSQL from various people invested in the community. It's a great way to stay abreast of what new features are coming or hearing interesting tips and advice.

Twitter

There is also an official Twitter account for PostgreSQL, `@postgresql`.

Thank you for buying

Instant PostgreSQL Starter

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

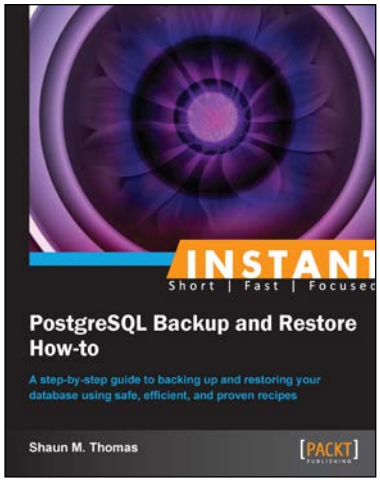
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] PUBLISHING

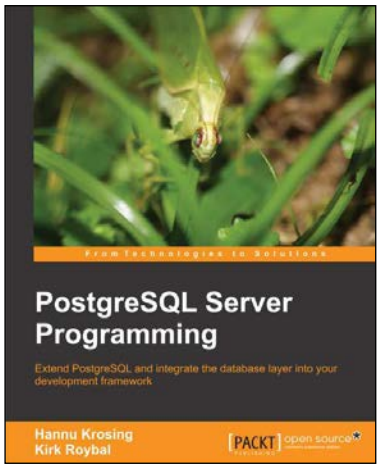


Instant PostgreSQL Backup and Restore How-to

ISBN: 978-1-78216-910-9 Paperback: 54 pages

A step-by-step guide to backing up and restoring your database using safe, efficeint, and proven recipes

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results.
2. Back up and restore PostgreSQL databases
3. Use built-in tools to create simple backups
4. Restore the easy way with internal commands
5. Cut backup and restore time with advanced techniques



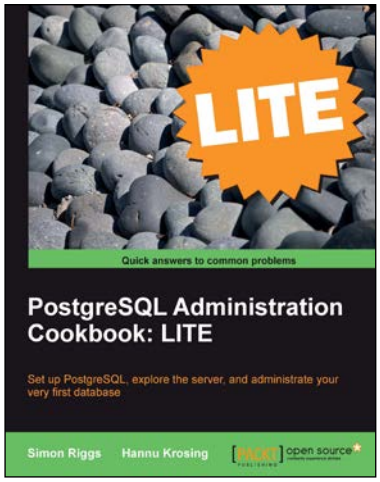
PostgreSQL Server Programming

ISBN: 978-1-84951-698-3 Paperback: 234 pages

Extend PostgreSQL and intergrate the database layer into your development framework

1. Understand the extension framework of PostgreSQL, and leverage it in ways that you haven't even invented yet
2. Write functions, create your own data types, all in your favourite programming language
3. Step-by-step tutorial with plenty of tips and tricks to kick-start server programming

[PACKT] PUBLISHING



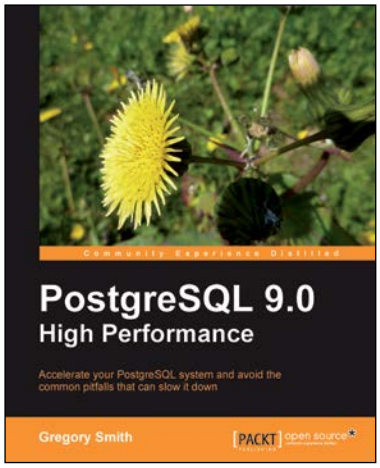
PostgreSQL 9 Administration Cookbook: LITE Edition

ISBN: 978-1-84951-620-4 Paperback: 104 pages

Set up PostgreSQL, explore the server, and administrate your very first database

1. Learn the basics of working with PostgreSQL; connect to a server, explore the database, and adjust the size of your database to the space available
2. Amend your database with the addition or removal of columns, schemas and tablespaces
3. Part of Packt's Cookbook series: Each recipe is a carefully organized sequence of instructions to complete the task as efficiently as possible

---



PostgreSQL 9.0 High Performance

ISBN: 978-1-84951-030-1 Paperback: 468 pages

Accelerate your PostgreSQL system and avoid the common pitfalls that can slow it down

1. Learn the right techniques to obtain optimal PostgreSQL database performance, from initial design to routine maintenance
2. Discover the techniques used to scale successful database installations
3. Avoid the common pitfalls that can slow your system down
4. Filled with advice about what you should be doing; how to build experimental databases to explore performance topics, and then move what you've learned into a production database environment