Troubleshooting PostgreSQL

Intercept problems and challenges typically faced by PostgreSQL database administrators with the best troubleshooting techniques

Hans-Jürgen Schönig
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Installing PostgreSQL'
- A synopsis of the book’s content
- More information on Troubleshooting PostgreSQL

About the Author

Hans-Jürgen Schönig has been in the PostgreSQL business since 2000. His company, Cybertec Schönig & Schönig GmbH (http://www.postgresql-support.de), serves clients around the globe, providing 24/7 support, replication, development, consulting, and training. He has written numerous books on PostgreSQL.
Troubleshooting PostgreSQL

First of all, I have to say "thank you" to you, for deciding to give this little book a chance as you are about to read this preface. I hope this book will give you valuable insights, and I sincerely hope that it will contribute to successful work on PostgreSQL. I did my best to include all of the information I think is needed to make you greatly successful and efficient when it comes to your daily business.

In my 15 years as a professional PostgreSQL consultant and commercial supporter at my company, Cybertec Schönig & Schönig GmbH (www.postgresql-support.de), I have seen countless setups—big, small, medium, complex, and simple. Over the past couple of years, I have assembled the 'best of problems,' which I would like to present in this book.

The book you are looking at is unlike all other books. It has not been written in some tutorial-style way. It really focuses on common problems and their solutions. Each problem described will include some background information, why it happens, how it can be solved, and what you can do to avoid it. In this way, I hope to solve and fix your problems as quickly and efficiently as possible.

I truly hope that you find this little book useful, and that it contains all you need to make your setup work.

What This Book Covers

*Chapter 1, Installing PostgreSQL*, explains how to install PostgreSQL on various types of systems. Source as well as binary installation are discussed and common pitfalls are presented.

*Chapter 2, Creating Data Structures*, focuses entirely on optimizing data structures. A lot can be achieved by following some basic rules, such as ordering columns the right way and a lot more.

*Chapter 3, Handling Indexes*, is probably the most important chapter when it comes to troubleshooting. If indexes are not used wisely, performance problems are certain to happen, and customer satisfaction will be low. In this chapter, indexes and their impact are discussed in great detail.

*Chapter 4, Reading Data Efficiently and Correctly*, provides insights into how to read data efficiently. Especially when reading large amounts of data, the way data is read does make a real difference, and a lot of potential is lost if data is retrieved in a dumb way.
Chapter 5, Getting Transactions and Locking Right, focuses on common problems related to transactions. How can problematic locks be found, and how can bad locking be avoided in general? How can consistency be ensured, and how can speed be optimized? All of these questions are discussed in this chapter.

Chapter 6, Writing Proper Procedures, attacks the most common pitfalls related to stored procedures. In particular, performance-related topics are covered in detail.

Chapter 7, PostgreSQL Monitoring, is dedicated to all administrators out there who want to ensure that their systems stay up and running 24 x 7. Monitoring can be the key to success here, and therefore, an entire chapter is dedicated to this highly important topic.

Chapter 8, Fixing Backups and Replication, focuses on backups as well as advanced replication topics. This chapter discusses point-in-time recovery, asynchronous replication, as well as synchronous replication in a practical way. On top of that, normal, textual backups are explained.

Chapter 9, Handling Hardware and Software Disasters, shows you all the tools required to handle typical hardware and filesystem failures. It shows how to rescue a PostgreSQL database and restore as much data as possible.

Chapter 10, A Standard Approach to Troubleshooting, outlines a basic and standard approach that works for most cases of ad hoc diagnosis in the field of PostgreSQL. A step-by-step guide is presented, and it can easily be replicated by users to attack most common performance issues.
Installing PostgreSQL

In this chapter, we will cover what can go wrong during the installation process and what can be done to avoid those things from happening. At the end of the chapter, you should be able to avoid all of the pitfalls, traps, and dangers you might face during the setup process.

For this chapter, I have compiled some of the core problems that I have seen over the years, as follows:

- Deciding on a version during installation
- Memory and kernel issues
- Preventing problems by adding checksums to your database instance
- Wrong encodings and subsequent import errors
- Polluted template databases
- Killing the postmaster badly

At the end of the chapter, you should be able to install PostgreSQL and protect yourself against the most common issues popping up immediately after installation.

Deciding on a version number

The first thing to work on when installing PostgreSQL is to decide on the version number. In general, a PostgreSQL version number consists of three digits. Here are some examples:

- 9.4.0, 9.4.1, or 9.4.2
- 9.3.4, 9.3.5, or 9.3.6
Installing PostgreSQL

The last digit is the so-called minor release. When a new minor release is issued, it generally means that some bugs have been fixed (for example, some time zone changes, crashes, and so on). There will never be new features, missing functions, or changes of that sort in a minor release. The same applies to something truly important—the storage format. It won’t change with a new minor release.

These little facts have a wide range of consequences. As the binary format and the functionality are unchanged, you can simply upgrade your binaries, restart PostgreSQL, and enjoy your improved minor release.

When the digit in the middle changes, things get a bit more complex. A changing middle digit is called a major release. It usually happens around once a year and provides you with significant new functionality. If this happens, we cannot just stop or start the database anymore to replace the binaries. In this case, we face a real migration process, which will be discussed later on in this book.

If the first digit changes, something really important has happened. Examples of such important events were introductions of SQL (6.0), the Windows port (8.0), streaming replication (9.0), and so on. Technically, there is no difference between the first and the second digit—they mean the same thing to the end user. However, a migration process is needed.

The question that now arises is this: if you have a choice, which version of PostgreSQL should you use? Well, in general, it is a good idea to take the latest stable release. In PostgreSQL, every version number following the design patterns I just outlined is a stable release.

As of PostgreSQL 9.4, the PostgreSQL community provides fixes for versions as old as PostgreSQL 9.0. So, if you are running an older version of PostgreSQL, you can still enjoy bug fixes and so on.

Methods of installing PostgreSQL

Before digging into troubleshooting itself, the installation process will be outlined. The following choices are available:

- Installing binary packages
- Installing from source

Installing from source is not too hard to do. However, this chapter will focus on installing binary packages only. Nowadays, most people (not including me) like to install PostgreSQL from binary packages because it is easier and faster.
Basically, two types of binary packages are common these days: RPM (Red Hat-based) and DEB (Debian-based). In this chapter, installing these two types of packages will be discussed.

Installing RPM packages
Most Linux distributions include PostgreSQL. However, the shipped PostgreSQL version is somewhat ancient in many cases. Recently, I saw a Linux distribution that still featured PostgreSQL 8.4, a version already abandoned by the PostgreSQL community. Distributors tend to ship older versions to ensure that new bugs are not introduced into their systems. For high-performance production servers, outdated versions might not be the best idea, however.

Clearly, for many people, it is not feasible to run long-outdated versions of PostgreSQL. Therefore, it makes sense to make use of repositories provided by the community. The Yum repository shows which distributions we can use RPMs for, at http://yum.postgresql.org/repopackages.php.

Once you have found your distribution, the first thing is to install this repository information for Fedora 20 as it is shown in the next listing:

```
yum install http://yum.postgresql.org/9.4/fedora/fedora-20-x86_64/pgdg-fedora94-9.4-1.noarch.rpm
```

Once the repository has been added, we can install PostgreSQL:

```
yum install postgresql94-server postgresql94-contrib
/usr/pgsql-9.4/bin/postgresql94-setup initdb
systemctl enable postgresql-9.4.service
systemctl start postgresql-9.4.service
```

First of all, PostgreSQL 9.4 is installed. Then a so-called database instance is created (initdb). Next, the service is enabled to make sure that it is always there after a reboot, and finally, the postgresql-9.4 service is started.

The term database instance is an important concept. It basically describes an entire PostgreSQL environment (setup). A database instance is fired up when PostgreSQL is started. Databases are part of a database instance.
Installing Debian packages

Installing Debian packages is also not too hard. By the way, the process on Ubuntu as well as on some other similar distributions is the same as that on Debian, so you can directly use the knowledge gained from this section for other distributions.

A simple file called `/etc/apt/sources.list.d/pgdg.list` can be created, and a line for the PostgreSQL repository (all the following steps can be done as root user or using `sudo`) can be added:

```
deb http://apt.postgresql.org/pub/repos/apt/ YOUR_DEBIAN_VERSION_HERE-pgdg main
```

So, in the case of Debian Wheezy, the following line would be useful:

```
deb http://apt.postgresql.org/pub/repos/apt/ wheezy-pgdg main
```

Once we have added the repository, we can import the signing key:

```
$# wget --quiet -O - \
    https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key add -
OK
```

Voilà! Things are mostly done. In the next step, the repository information can be updated:

```
apt-get update
```

Once this has been done successfully, it is time to install PostgreSQL:

```
apt-get install "postgresql-9.4"
```

If no error is issued by the operating system, it means you have successfully installed PostgreSQL. The beauty here is that PostgreSQL will fire up automatically after a restart. A simple database instance has also been created for you.

If everything has worked as expected, you can give it a try and log in to the database:

```
root@chantal:~# su - postgres
$ psql postgres
psql (9.4.1)
Type "help" for help.
postgres=#
```
Memory and kernel issues

After this brief introduction to installing PostgreSQL, it is time to focus on some of the most common problems.

Fixing memory issues

Some of the most important issues are related to the kernel and memory. Up to version 9.2, PostgreSQL was using the classical system V shared memory to cache data, store locks, and so on. Since PostgreSQL 9.3, things have changed, solving most issues people had been facing during installation.

However, in PostgreSQL 9.2 or before, you might have faced the following error message:

- **FATAL**: Could not create shared memory segment
- **DETAIL**: Failed system call was `shmget`: (key=5432001, size=112263040, 03600)
- **HINT**: This error usually means that PostgreSQL's request for a shared memory segment exceeded your kernel's `SHMMAX` parameter. You can either reduce the request size or reconfigure the kernel with larger `SHMMAX`. To reduce the request size (currently 112263040 bytes), reduce PostgreSQL's shared memory usage, perhaps by reducing shared_buffers or max_connections.

If the request size is already small, it's possible that it is less than your kernel's `SHMMIN` parameter, in which case raising the request size or reconfiguring `SHMMIN` is called for.

The PostgreSQL documentation contains more information about shared memory configuration.

If you are facing a message like this, it means that the kernel does not provide you with enough shared memory to satisfy your needs. Where does this need for shared memory come from? Back in the old days, PostgreSQL stored a lot of stuff, such as the I/O cache (shared_buffers, locks, autovacuum-related information and a lot more), in the shared memory. Traditionally, most Linux distributions have had a tight grip on the memory, and they don't issue large shared memory segments; for example, Red Hat has long limited the maximum amount of shared memory available to applications to 32 MB. For most applications, this is not enough to run PostgreSQL in a useful way—especially not if performance does matter (and it usually does).
Installing PostgreSQL

To fix this problem, you have to adjust kernel parameters. Managing Kernel Resources of the PostgreSQL Administrator's Guide will tell you exactly why we have to adjust kernel parameters.

For more information, check out the PostgreSQL documentation at http://www.postgresql.org/docs/9.4/static/kernel-resources.htm.

This chapter describes all the kernel parameters that are relevant to PostgreSQL. Note that every operating system needs slightly different values here (for open files, semaphores, and so on).

Since not all operating systems can be covered in this little book, only Linux and Mac OS X will be discussed here in detail.

Adjusting kernel parameters for Linux

In this section, parameters relevant to Linux will be covered. If shmget (previously mentioned) fails, two parameters must be changed:

```bash
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

In this example, shmmax and shmall have been adjusted to 16 GB. Note that shmmax is in bytes while shmall is in 4k blocks. The kernel will now provide you with a great deal of shared memory.

Also, there is more; to handle concurrency, PostgreSQL needs something called semaphores. These semaphores are also provided by the operating system. The following kernel variables are available:

- **SEMMNI**: This is the maximum number of semaphore identifiers. It should be at least ceil((max_connections + autovacuum_max_workers + 4) / 16).
- **SEMMNS**: This is the maximum number of system-wide semaphores. It should be at least ceil((max_connections + autovacuum_max_workers + 4) / 16) * 17, and it should have room for other applications in addition to this.
- **SEMMUL**: This is the maximum number of semaphores per set. It should be at least 17.
- **SEMMAP**: This is the number of entries in the semaphore map.
- **SEMMX**: This is the maximum value of the semaphore. It should be at least 1000.

Don't change these variables unless you really have to. Changes can be made with `sysctl`, as was shown for the shared memory.
Adjusting kernel parameters for Mac OS X

If you happen to run Mac OS X and plan to run a large system, there are also some kernel parameters that need changes. Again, /etc/sysctl.conf has to be changed.

Here is an example:

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

Mac OS X is somewhat nasty to configure. The reason is that you have to set all five parameters to make this work. Otherwise, your changes will be silently ignored, and this can be really painful.

In addition to that, it has to be assured that SHMMAX is an exact multiple of 4096. If it is not, trouble is near.

If you want to change these parameters on the fly, recent versions of OS X provide a `sysctl` command just like Linux. Here is how it works:

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

Fixing other kernel-related limitations

If you are planning to run a large-scale system, it can also be beneficial to raise the maximum number of open files allowed. To do that, /etc/security/limits.conf can be adapted, as shown in the next example:

```
postgres   hard    nofile    1024
postgres   soft    nofile    1024
```

This example says that the `postgres` user can have up to 1,024 open files per session.

Note that this is only important for large systems; open files won't hurt an average setup.
Installing PostgreSQL

Adding checksums to a database instance

When PostgreSQL is installed, a so-called database instance is created. This step is performed by a program called initdb, which is a part of every PostgreSQL setup. Most binary packages will do this for you and you don't have to do this by hand.

Why should you care then? If you happen to run a highly critical system, it could be worthwhile to add checksums to the database instance. What is the purpose of checksums? In many cases, it is assumed that crashes happen instantly — something blows up and a system fails. This is not always the case. In many scenarios, the problem starts silently. RAM may start to break, or the filesystem may start to develop slight corruption. When the problem surfaces, it may be too late. Checksums have been implemented to fight this very problem. Whenever a piece of data is written or read, the checksum is checked. If this is done, a problem can be detected as it develops.

How can those checksums be enabled? All you have to do is to add -k to initdb (just change your init scripts to enable this during instance creation). Don't worry! The performance penalty of this feature can hardly be measured, so it is safe and fast to enable its functionality.

Keep in mind that this feature can really help prevent problems at fairly low costs (especially when your I/O system is lousy).

Preventing encoding-related issues

Encoding-related problems are some of the most frequent problems that occur when people start with a fresh PostgreSQL setup. In PostgreSQL, every database in your instance has a specific encoding. One database might be en_US@UTF-8, while some other database might have been created as de_AT@UTF-8 (which denotes German as it is used in Austria).

To figure out which encodings your database system is using, try to run psql -l from your Unix shell. What you will get is a list of all databases in the instance that include those encodings.

So where can we actually expect trouble? Once a database has been created, many people would want to load data into the system. Let's assume that you are loading data into the aUTF-8 database. However, the data you are loading contains some ASCII characters such as ä, ö, and so on. The ASCII code for ö is 148. Binary 148 is not a valid Unicode character. In Unicode, U+00F6 is needed. Boom! Your import will fail and PostgreSQL will error out.
If you are planning to load data into a new database, ensure that the encoding or character set of the data is the same as that of the database. Otherwise, you may face ugly surprises.

To create a database using the correct locale, check out the syntax of `CREATE DATABASE`:

```
test=# \h CREATE DATABASE
Command: CREATE DATABASE
Description: create a new database
Syntax:
CREATE DATABASE name
  [ [ WITH ] [ OWNER [=] user_name ]
   [ TEMPLATE [=] template ]
   [ ENCODING [=] encoding ]
   [ LC_COLLATE [=] lc_collate ]
   [ LC_CTYPE [=] lc_ctype ]
   [ TABLESPACE [=] tablespace_name ]
   [ CONNECTION LIMIT [=] connlimit ] ]
```

`ENCODING` and the `LC*` settings are used here to define the proper encoding for your new database.

**Avoiding template pollution**

It is somewhat important to understand what happens during the creation of a new database in your system. The most important point is that `CREATE DATABASE` (unless told otherwise) clones the `template1` database, which is available in all PostgreSQL setups.

This cloning has some important implications. If you have loaded a very large amount of data into `template1`, all of that will be copied every time you create a new database. In many cases, this is not really desirable but happens by mistake. People new to PostgreSQL sometimes put data into `template1` because they don't know where else to place new tables and so on. The consequences can be disastrous.

However, you can also use this common pitfall to your advantage. You can place the functions you want in all your databases in `template1` (maybe for monitoring or whatever benefits).
Installing PostgreSQL

Killing the postmaster

After PostgreSQL has been installed and started, many people wonder how to stop it. The most simplistic way is, of course, to use your service postgresql stop or /etc/init.d/postgresql stop init scripts.

However, some administrators tend to be a bit crueler and use kill -9 to terminate PostgreSQL. In general, this is not really beneficial because it will cause some nasty side effects. Why is this so?

The PostgreSQL architecture works like this: when you start PostgreSQL you are starting a process called postmaster. Whenever a new connection comes in, this postmaster forks and creates a so-called backend process (BE). This process is in charge of handling exactly one connection. In a working system, you might see hundreds of processes serving hundreds of users. The important thing here is that all of those processes are synchronized through some common chunk of memory (traditionally, shared memory, and in the more recent versions, mapped memory), and all of them have access to this chunk. What might happen if a database connection or any other process in the PostgreSQL infrastructure is killed with kill -9? A process modifying this common chunk of memory might die while making a change. The process killed cannot defend itself against the onslaught, so who can guarantee that the shared memory is not corrupted due to the interruption?

This is exactly when the postmaster steps in. It ensures that one of these backend processes has died unexpectedly. To prevent the potential corruption from spreading, it kills every other database connection, goes into recovery mode, and fixes the database instance. Then new database connections are allowed again.

While this makes a lot of sense, it can be quite disturbing to those users who are connected to the database system. Therefore, it is highly recommended not to use kill -9. A normal kill will be fine.

Keep in mind that a kill -9 cannot corrupt your database instance, which will always start up again. However, it is pretty nasty to kick everybody out of the system just because of one process!

Summary

In this chapter, you learned how to install PostgreSQL using binary packages. Some of the most common problems and pitfalls, including encoding-related issues, checksums, and versioning were discussed.

In the next chapter, we'll deal with data structures and common issues related to them.
Where to buy this book

You can buy Troubleshooting PostgreSQL from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.