



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Troubleshooting PostgreSQL

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

Hans-Jürgen Schöning

[PACKT] open source*
PUBLISHING community experience distilled

Troubleshooting PostgreSQL

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

Hans-Jürgen Schönig



BIRMINGHAM - MUMBAI

Troubleshooting PostgreSQL

Copyright © 2015 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 author, 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: March 2015

Production reference: 1250315

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

ISBN 978-1-78355-531-4

www.packtpub.com

Credits

Author

Hans-Jürgen Schöning

Project Coordinator

Sanchita Mandal

Reviewers

Ludovic Gasc

Baji Shaik

Sheldon E. Strauch

Proofreaders

Simran Bhogal

Bernadette Watkins

Commissioning Editor

Julian Ursell

Indexer

Monica Ajmera Mehta

Acquisition Editor

Harsha Bharwani

Production Coordinator

Arvindkumar Gupta

Content Development Editor

Adrian Raposo

Cover Work

Arvindkumar Gupta

Technical Editor

Siddhesh Ghadi

Copy Editor

Vikrant Phadke

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.

About the Reviewers

Ludovic Gasc is a senior software developer and engineer at Eyepea/ALLOcloud, a highly renowned open source VoIP and unified communications company in Europe.

Over 5 years, Ludovic has developed redundant distributed systems for Telecom, based on Python, AsyncIO, and PostgreSQL.

He is also the creator of API-Hour. He writes efficient network daemons (HTTP, SSH, and so on) with ease. For more information, you can visit <http://www.gmludo.eu/>.

Baji Shaik has completed his engineering in telecommunications and started his career as a developer for C# and Java. Later, he worked as a consultant in production and development environments. He then explored different database technologies such as Oracle, PostgreSQL, and Greenplum. Baji is certified in PostgreSQL, and at present, he is leading a team of 15 members in a multinational organization and is directly involved in big data analytics.

Baji is from a small village named Vutukuru, Andhra Pradesh, India. Currently, he lives in Bangalore. His hobbies are watching movies and playing with Rubik's Cube. He likes to read and writes technical blogs.

Sheldon E. Strauch has been a veteran for 20 years of software consulting at companies such as IBM, Sears, Ernst & Young, and Kraft Foods. He has a bachelor's degree in business administration and leverages his technical skills to improve self-awareness of business. His interests include data gathering, management, and mining; maps and mapping; business intelligence; and application of data analysis for continuous improvement. He is currently focused on the development of end-to-end data management and mining at Enova International, a financial services company located in Chicago. In his spare time, Sheldon enjoys performing arts, particularly music, and traveling with his wife, Marilyn.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

This book is dedicated to all who have supported me over the years. Special thanks goes to my entire team here at Cybertec Schönig & Schönig GmbH (www.postgresql-support.de), which has done a great job over the years.

Thank you for your loyalty, your professionalism, and your wonderful characters. I cannot envision any better people to work with. Thank you all ...

Table of Contents

Preface	v
Chapter 1: Installing PostgreSQL	1
Deciding on a version number	1
Methods of installing PostgreSQL	2
Installing RPM packages	3
Installing Debian packages	4
Memory and kernel issues	5
Fixing memory issues	5
Adjusting kernel parameters for Linux	6
Adjusting kernel parameters for Mac OS X	7
Fixing other kernel-related limitations	7
Adding checksums to a database instance	8
Preventing encoding-related issues	8
Avoiding template pollution	9
Killing the postmaster	10
Summary	10
Chapter 2: Creating Data Structures	11
Grouping columns the right way	11
Deciding on data types and structure	13
Finding the right type	13
varchar versus text	13
numeric versus floating point	14
boolean fields versus bit fields	15
text versus cidr/inet/circle/point	16
Deciding on normalization	17
The 7th normal form	18
Arrays versus normalizations	19
Summary	20

Chapter 3: Handling Indexes	21
Understanding indexes in PostgreSQL	21
Using a simple index	22
How an index works	24
Avoiding trouble with indexes	25
Detecting missing indexes	26
Removing useless indexes	27
Solving common problems	29
Managing foreign keys	29
Indexing geometric data using GiST	29
Handling LIKE queries	30
Simple LIKE queries	31
More advanced LIKE queries	32
Finding good matches	34
Fixing full-text search	35
Not using full-text search at all	35
Full-text search and sorting	38
Summary	39
Chapter 4: Reading Data Efficiently and Correctly	41
Understanding the power of NULL	41
Seeing NULL in action	42
NULL and storage	43
Fixing disastrous joins	43
Create demo data for joins	44
Understanding outer joins	44
Reading large amounts of data	47
Making use of cursors	47
Synchronized scanning	50
Understanding prepared queries	51
Summary	53
Chapter 5: Getting Transactions and Locking Right	55
The PostgreSQL transaction model	55
Understanding savepoints	57
Understanding basic locking and deadlocks	58
Locking in FOR UPDATE mode	60
Avoiding performance bottlenecks	61
Avoiding table locks	62

Transaction isolation	63
Demonstrating read committed mode	63
Using repeatable read	65
Beyond repeatable read	65
Inspecting locks	66
Summary	69
Chapter 6: Writing Proper Procedures	71
Choosing the right language	71
Trusted versus untrusted	72
Managing procedures and transactions	72
Using transactions to avoid trouble	73
Understanding transactions and procedures	74
Procedures and indexing	75
LEAKPROOF versus NOT LEAKPROOF	77
Understanding memory	80
Procedures and cursors	80
Handling set-returning functions	81
Assigning memory parameters to functions	81
Summary	82
Chapter 7: PostgreSQL Monitoring	83
Checking the overall database behavior	83
Checking pg_stat_activity	84
Checking database-level information	86
Detecting I/O bottlenecks	88
Checking for conflicts	89
Chasing down slow queries	90
Notes about overhead	92
Resetting data	92
Inspecting internal information	92
Looking inside a table	92
Inspecting the I/O cache	94
Integrating with external tools	95
Using Nagios plugins	95
Alternative tools	96
Zabbix plugins	96
pganalyze-collector	96
pg_view – a simple alternative	97
Summary	97

Chapter 8: Fixing Backups and Replication	99
Using pg_dump	99
Creating textual dumps	99
Taking care of blobs	100
Handling passwords	100
Creating custom format dumps	101
Making use of many CPUs	102
Managing point-in-time recovery	102
How PITR works	103
Preparing PostgreSQL for PITR	103
Taking base backups	104
Replaying xlog	105
Making use of asynchronous replication	107
Working with pg_basebackup	108
Firing up replication	110
Promoting slaves to masters	112
Making replication safer	112
Switching to synchronous replication	113
Handling timelines	114
Summary	115
Chapter 9: Handling Hardware and Software Disasters	117
Checksums – preventing silent corruption	118
Zeroing out damaged pages	118
Dealing with index corruption	120
Dumping individual pages	120
Extracting the page header	121
Resetting the transaction log	122
Power-out-related issues	124
Summary	125
Chapter 10: A Standard Approach to Troubleshooting	127
Getting an overview of the problem	127
Attacking low performance	128
Reviewing indexes	128
Fixing UPDATE commands	130
Detecting slow queries	131
Fixing common replication issues	132
Fixing stopped replication	132
Fixing failed queries	134
Summary	134
Index	135

Preface

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.

What you need for this book

This book has been written with Linux and Mac OS X in mind. However, 90 percent of all examples will also work with all changes on Windows platforms.

No special tools are needed to work through this book.

Who this book is for

This book has been written for and is dedicated to people who actively work with PostgreSQL and who want to avoid common pitfalls. It has been written with more than 15 years of professional PostgreSQL consulting, support, and training in mind. The goal is to really capture those hot topics and provide easy-to-understand solutions.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.


Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:
"The count (bid) function returns 1, while the count (*) function will return 3."


A block of code is set as follows:

```
test=# INSERT INTO t_test SELECT 10, 20, 30,  
      'abcd', 'abcd', 'abcd'  
      FROM generate_series(1, 10000000);  
INSERT 0 10000000
```

Any command-line input or output is written as follows:

```
SELECT * FROM website WHERE domain = 'www.cybertec.at';
```

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

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

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>.

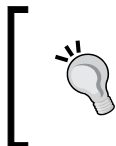
Once you have found your distribution, the first thing is to install this repository information for Fedora 20 as it is shown in the next listing:

```
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=1122263040, 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 1122263040 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).

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:

```
$ 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 $\text{ceil}((\text{max_connections} + \text{autovacuum_max_workers} + 4) / 16)$.
- `SEMMNS`: This is the maximum number of system-wide semaphores. It should be at least $\text{ceil}((\text{max_connections} + \text{autovacuum_max_workers} + 4) / 16) * 17$, and it should have room for other applications in addition to this.
- `SEMMSL`: This is the maximum number of semaphores per set. It should be at least 17.
- `SEMAPP`: This is the number of entries in the semaphore map.
- `SEVMX`: 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.

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).

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.

2

Creating Data Structures

This chapter is dedicated to data structures and their creation. The following topics will be covered:

- Grouping columns the right way
- Deciding on the right data type

Grouping columns the right way

Some people might wonder what is meant by the title of this section. Does it make a difference which order columns are aligned in? This might be a bit surprising, but it does. Even if a table contains the same data, its size on the disk might vary depending on the order of columns. Here is an example:

```
test=# CREATE TABLE t_test (  i1 int,
                                i2 int,
                                i3 int,
                                v1 varchar(100),
                                v2 varchar(100),
                                v3 varchar(100)
                                );
CREATE TABLE
test=# INSERT INTO t_test SELECT 10, 20, 30,
                                'abcd', 'abcd', 'abcd'
                                FROM generate_series(1, 10000000);
INSERT 0 10000000
```

A table with three columns has been created. First of all, there are three integer columns. Then some varchar columns are added. In the second statement, 10 million rows are added. The `generate_series` command is a nice way to generate a list of numbers. In this example, the output of `generate_series` is not used. I am just utilizing the function to repeat the static data in the `SELECT` clause.

Now the size of the table can be checked:

```
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
pg_size_pretty
-----
574 MB
(1 row)
```

The `pg_relation_size` returns the size of a table in bytes. In the case of a large relation, this is somewhat unhandy because users may easily end up with very large, hard-to-read numbers. Therefore, the return value of `pg_relation_size` should be wrapped into `pg_size_pretty`, which makes the size a lot more useful, as data is returned in a more human-readable format.

Let's try the same example again. This time, however, the order of the columns is changed. The rest of the example stays exactly the same:

```
test=# CREATE TABLE t_test (  v1 varchar(100),
                                i1 int,
                                v2 varchar(100),
                                i2 int,
                                v3 varchar(100),
                                i3 int
                                );
CREATE TABLE
test=# INSERT INTO t_test SELECT 'abcd', 10, 'abcd',
                                20, 'abcd', 30
                                FROM generate_series(1, 10000000);
INSERT 0 10000000
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
pg_size_pretty
-----
651 MB
(1 row)
```

The table has grown considerably, even though the data inside the table is exactly the same. The reason for this problem is called *alignment* and can be found deep inside the code. The theory is as follows: a CPU has a hard time if a field does not start at a multiplier of CPU word-size. Therefore, PostgreSQL will accordingly align data physically.

The most important point here is that it can make sense to group columns with similar data types next to each other. Of course, the outcome and the potential difference in size greatly depend on the content. If "abc" was used instead of "abcd" in this example, the results would not have shown any difference; both versions of the table would have had a fixed size of 498 MB.



Note that different types of CPUs (x86_32, x86_64, Sparc, ARM, and so on.) may not lead to the same results here.

Deciding on data types and structure

After this brief introduction to organizing columns, it makes sense to discuss various other issues, such as choosing the right data type and normalization. **Normalization** is a standard technique to organize data and it has proven to be essential over the years. The idea is to avoid redundancy to ensure that changes only have to be made in one place. To make your database project a success, deciding on proper types is essential and will help to prevent a great deal of trouble later on in the project. The goal of this chapter is to make you aware of the potentials of using the right types.

Finding the right type

Finding the right data type sounds easy. However, there is a lot to consider, and there are some issues that might actually cause problems.

varchar versus text

The first two types to be compared are `varchar` and `text`. Have you ever seen a data structure like this?

```
test=# CREATE TABLE t_product (
      prodnr      int,
      name        varchar(20),
      description  varchar(4096)
    );
CREATE TABLE
```

A product number defined as an integer can make a lot of sense (some people use alphanumeric product names, but never mind). Keeping the name limited to 20 characters is also reasonable. There might be real limitations here. Maybe, a field on some printout is limited to 20 characters, or some label somewhere is limited to a 20-character title.

But what about the description? Does 4096 really impose a useful restriction here? I guess not. It is just an arbitrary number. What the person really meant is rather something like this:

```
description      text
```

The beauty of `text` is that it is not limited to a certain number of characters but to 1 GB per entry. Hence, `text` can make life for software developers a little easier.

numeric versus floating point

While `varchar` versus `text` was more a question of convenience, `numeric` versus `floating point` is a matter of life and death because it is all about correctness.

What is the difference anyway? Here is an example using `float`:

```
test=# SELECT '1000.1'::float - '1000' AS float;
         float
-----
0.10000000000000023
(1 row)
```

What you see here is that the example is close, but not necessarily absolutely correct. The reason is the way floating-point numbers work.



The inner working of the floating point standard is defined in IEEE 754, which is described at:

http://en.wikipedia.org/wiki/IEEE_floating_point.

The main issue is that depending on the magnitude of the number, precision will go down. This might be fine if you are dealing with some sort of sensor data. However, it is totally impossible if you happen to work with money. The price of a stamp is the same, no matter whether you are living in a slum or you happen to be a billionaire. The number is always a static number. With floating point, it would be merely impossible to achieve that.

To get around the chip's **FPU** (short for **floating-point unit**), developers and designers alike can turn to `numeric`. The beauty of `numeric` is that it relies entirely on the working of the CPU's integer unit. It offers fixed, predefined, and standard behavior to handle numbers of arbitrary precision. It is perfect for numbers, as the following example shows:

```
test=# SELECT '1000.1'::numeric - '1000';
?column?
-----
0.1
(1 row)
```

The `numeric` option should be used for financial data and all other applications requiring precision.



The numeric is a lot faster than floating-point-based data types. However, it's not about speed; it's about the correctness of your results. A correct result is usually a lot more important than a fast result.

boolean fields versus bit fields

In the past, I have seen countless situations in which people have used a massive number of boolean columns in a table. This is not especially harmful, but it might not be the best thing either. A table in PostgreSQL can hold up to 1600 columns. However, keeping 100 boolean columns for a certain scenario is not too handy.

In PostgreSQL, you can consider the following way out: two data types called `bit` and `varbit` are offered to reduce the number of columns and simplify things. Here is an example:

```
test=# CREATE TABLE t_test (x bit(10));
CREATE TABLE
test=# INSERT INTO t_test VALUES ('0110000101');
INSERT 0 1
test=# SELECT * FROM t_test;
      x
-----
0110000101
(1 row)
```

In this example, a `bit(10)` column has been created. The meaning of `bit(10)` is that a field with 10 bits is created.

If you want to retrieve the data, it is possible to simply select the entire column or to retrieve a single bit inside the field (`get_bit`). Of course, bits can also be modified (`set_bit`). Here is an example:

```
test=# SELECT get_bit(x, 5), set_bit(x, 5, 1), x::int FROM t_test;
 get_bit | set_bit |  x
-----+-----+-----
       0 | 0110010101 | 389
(1 row)
```

If you are not sure how many bits you will need, there is a data type called `varbit`. Basically, it works just like `varchar`. Here is an example:

```
test=# ALTER TABLE t_test ADD COLUMN y varbit(20);
ALTER TABLE
```

A column that can hold up to 20 bits, is added to `t_test`.

As mentioned before, `bit` and `varbit` are just options. It is not necessary to use this kind of stuff, but they can help to improve your data structures in a way to prevent trouble.

text versus cidr/inet/circle/point

One common mistake made by many people is to choose a generic data type instead of a specific, optimized data type. Of course, everything can be stored as text somehow, but in most cases, it makes a lot more sense to use the proper data type.

Some of those specialized, optimized types are dealt with in this section.

Let's focus on IP addresses first. To represent an IP address, you have two choices: `inet` and `cidr`. Here is an example:

```
test=# SELECT '192.168.0.34/32'::cidr, '192.168.0.34/32'::inet;
      cidr      |      inet
-----+-----
 192.168.0.34/32 | 192.168.0.34
(1 row)
```

The `cidr` option represents an IP address that includes a netmask. The `inet` option will only store an IP address without a netmask.

By the way, both data types will also work for IPv6:

```
test=# SELECT '::1'::cidr, '::1'::inet;
      cidr      |      inet
-----+-----
 ::1/128 | ::1
(1 row)
```

The beauty here is that you have a special set of operators for those types, and you can even do the math, as the following example shows:

```
test=# SELECT '::1'::cidr + 43242342;
      ?column?
-----
 ::2.147.211.103
(1 row)
```

Besides data types related to IPs, there are a couple of types related to geometric objects. A common mistake is that people store geographical positions in two columns:

```
test=# CREATE TABLE t_location (
      loc    text,
      x      numeric,
```

```

        y    numeric
    );
CREATE TABLE

```

Of course, you can always use two fields, but querying this kind of data structure is a really nasty task. Just imagine searching an object within 50 kilometers. With two columns, this is clearly a complicated, nasty-to-do task.

The standard distribution of PostgreSQL (even without PostGIS, which is a popular GIS extension installed) offers some geometric data types such as points, circles, rectangles, paths, and so on. To make the previous table more efficient, the following can be done:

```

test=# CREATE TABLE t_test (loc text, coord point);
CREATE TABLE

```

Points can be used easily. Here is an example:

```

test=# SELECT '(10, 30)::point;
       point
-----
(10,30)
(1 row)

```

The really nice thing here is that there is quite a lot of useful functionality around. If you want to determine the distance between two points, you can make use of the \leftrightarrow (distance) operator:

```

test=# SELECT '(10, 30)::point <-> '(20, 20)::point;
       ?column?
-----
14.142135623731
(1 row)

```

You could've done that with a simple formula, but why? PostgreSQL already has all the functionality for you. Therefore, using an optimized, predefined data type is highly beneficial.

Deciding on normalization

When starting with databases, one of the first things people learn is data normalization. Normalization makes a lot of sense because it helps to reduce redundancy and it ensures that data is structured in a professional and nice way.

However, in some cases, normalization is taken way too far.

The 7th normal form

As mentioned in many cases, normalization can be easily exaggerated. In such a case, performance will suffer and things will be really complex. Here is an example of normalization taken too far: a few years ago, a professor at a local university confronted me with an interesting example when I was talking to students. He told me that he uses the following example to explain to students that normalization can easily be taken too far. He invented the 7th normal form (which, of course, does not exist in professional database literature).

Here is an example:

```
test=# CREATE TABLE t_test (name text);
CREATE TABLE
test=# INSERT INTO t_test VALUES ('sealevel');
INSERT 0 1
```

There is still some redundancy in sealevel. Some letters show up twice, so we can normalize that:

```
test=# CREATE TABLE t_char (id int, char char(1));
CREATE TABLE
test=# INSERT INTO t_char VALUES
      (1, 's'), (2, 'e'), (3, 'a'), (4, 'l'), (5, 'v');
INSERT 0 5
```

In this table, one entry is made for each letter. In addition to this, a second table can be created to finally form the word:

```
test=# CREATE TABLE t_word (order_id int, char_id int);
CREATE TABLE
```

The sealevel can, therefore, be represented as the following table:

```
test=# INSERT INTO t_word VALUES
      (1, 1), (2, 2), (3, 3), (4, 4), (5, 2),
      (6, 5), (7, 2), (8, 4);
INSERT 0 8
```

Joining entries together will reveal the word:

```
test=# SELECT order_id, char
      FROM   t_word AS a, t_char AS b
      WHERE  a.char_id = b.id
      ORDER BY order_id;
 order_id | char
-----+-----
```

1		s
2		e
3		a
4		l
5		e
6		v
7		e
8		l

(8 rows)


What is the point of all this? The point is that you can normalize your data beyond recognition. But does it really make sense? In the example just outlined, everybody will agree that this is clearly useless and way beyond what is justified. Therefore, it is highly recommended if it is really worth introducing new relations, even if there is no chance of ever making changes to them.

Arrays versus normalizations

To avoid excessive normalization, you can turn your attention to arrays. Arrays are a simple, and in many cases, a good way to reduce the number of relations, especially when you don't need hard integrity.

What is the main benefit of an array and of avoiding joins? Keep in mind that when using a join, you have to read data from two tables. This often means additional I/O, and most importantly, you have to join the data, which leads to sorting, hashing, merging, and so on.

If data is stored inline in an array, there is no need to read from two tables and join, merge, and so on. You may be able to enjoy the simplicity of a sequential scan or the speed of a single-index lookup to retrieve the data needed.

 Note that arrays are no Holy Grail! They are just a tool used to speed things up and a good option to improve data structures to prevent I/O problems.

To show how arrays can be used to reduce normalization, the following example may be beneficial to you. The goal is to come up with a data structure for storing which people have been to which countries:

```
test=# CREATE TABLE t_country (
      id          int,
      country_code char(2)
);
CREATE TABLE
```


Then some data can be added to the table:

```
test=# INSERT INTO t_country VALUES
      (1, 'DE'), (2, 'AT'), (3, 'BE'), (4, 'NL');
INSERT 0 4
```

In the next step, a table linking people and countries together is created:

```
test=# CREATE TABLE t_travel (
      person      text,
      country_id   int);
CREATE TABLE
```

Basically, this is the standard way of describing a relation like that. Take a look at this example:

```
test=# CREATE TABLE t_travel (
      person      text,
      country_list char(2)[]
);
CREATE TABLE
```

It is very unlikely that the name of a country will change, and even if it does, you may not care too much. Imagine that somebody has been to the Soviet Union. Even after its breakup, you would still not change the name of the country you have visited in retrospect. Therefore, integrity is really not an issue here.

The beauty is that you can save on all the expenses that might be caused by the join. In fact, in large-scale systems, random I/O can be a major bottleneck, which should be avoided. By keeping data close together, you can avoid a ton of problems.

In fact, too many joins on too much data can be a real source of trouble in a productive system because as soon as your server hits a random I/O limitation imposed on you by your disk, performance will drop dramatically.

Summary

In this chapter, you learned how to optimize data structures by avoiding some basic pitfalls. Of course, a discussion of how to write data structures properly can never be exhaustive, and around 10 pages are never enough. However, the idea of this chapter was to introduce some of the lesser-known aspects of these topics that you might be able to use to fix your applications.

The next chapter is all about troubleshooting in the sphere of indexing.

3

Handling Indexes

In this chapter, one of the most important topics in the area of database work will be covered – indexing. In many cases, missing or wrong indexes are the main source of trouble, leading to bad performance, unexpected behavior, and a great deal of frustration. To avoid these issues, this chapter will present all the information that you need to make your life as easy as possible.

The following issues will be covered:

- Understanding PostgreSQL indexing
- Types of indexes
- Managing operator classes
- Detecting missing indexes
- Reading execution plans
- Fixing `LIKE`
- Fixing regular expressions
- Improving indexes and functions
- Indexes and insertions
- Indexes and foreign keys

Understanding indexes in PostgreSQL

In this section, you will learn the fundamental concept of indexing. As already mentioned, indexes are highly important and their importance can hardly be overestimated. Therefore, it is important for you to understand what indexes are good for and how they can be used efficiently without harming performance. To illustrate indexes, the first thing to do is to introduce the concept of so-called execution plans.

Execution plans are a component vital to understanding PostgreSQL's performance as a whole. They are your window to the inner workings of the database system. It is important to understand execution plans and indexes.

Using a simple index

The basic idea of SQL is that you can write a query and the database will decide on the best strategy to execute the request. Deciding on the best strategy is the task of the so-called optimizer. The idea behind the optimizer is to figure out which query is expected to impose the lowest costs on the system, and go for that strategy. Consider the following example:

```
test=# CREATE TABLE t_test (id int4);
CREATE TABLE
```

A table with only one column has been created. Now 10 million rows are added:

```
test=# INSERT INTO t_test
      SELECT * FROM generate_series(1, 10000000);
INSERT 0 10000000
```

Once the data has been added, a simple query looking for a single row can be issued. The point here is that the query is prefixed with the `explain` keyword:

```
test=# explain SELECT * FROM t_test WHERE id = 423425;
          QUERY PLAN
-----
Seq Scan on t_test  (cost=0.00..169247.71
 rows=1 width=4)
  Filter: (id = 423425)
 Planning time: 0.143 ms
(3 rows)
```

As a result, PostgreSQL returns the so-called execution plan. It tells you how the system plans to execute the query. What is an execution plan? Basically, PostgreSQL will execute SQL in four stages:

- **Parser:** At this stage, the syntax of the query will be checked. The parser is what complains about syntax errors.
- **Rewrite system:** The rewrite system will rewrite the query, handle rules, and so on.

- **Optimizer:** At this point, PostgreSQL will decide on the strategy and come up with a so-called execution plan, which represents the fastest way through the query to get the result set.
- **Executor:** The plan generated by the optimizer will be executed by the executor, and the result will be returned to the end user.

The execution plan is exactly what has been created here. It tells us how PostgreSQL plans to execute the query. What we see is a so-called sequential scan. It means that the entire table is read and PostgreSQL checks every row to see whether it matches the condition or not. As our table is really long here, it takes quite a while to execute the query.

To receive some more detailed information, `explain analyze` can be used:

```
test=# explain analyze SELECT *
      FROM   t_test
     WHERE  id = 423425;
          QUERY PLAN
-----
Seq Scan on t_test  (cost=0.00..169247.71
 rows=1 width=4)
(actual time=60.842..1168.681 rows=1 loops=1)
  Filter: (id = 423425)
  Rows Removed by Filter: 9999999
Planning time: 0.050 ms
Execution time: 1168.706 ms
(5 rows)
```

The total runtime of `explain analyze` is 1.1 seconds—quite a lot for a small query. Of course, if the query was executed without `explain analyze`, it would have been a little faster. However, it is still close to a second. Close to a second is certainly not what end users should expect from a modern database.

This is where an index can help. Here is how we create an index:

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
```

Creating the index will solve all problems in a miraculous way. Now the query executes in a fraction of a millisecond, thousands of times faster than before:

```
test=# explain analyze SELECT *
      FROM   t_test
     WHERE  id = 423425;
          QUERY PLAN
-----
```

```
-----  
Index Only Scan using idx_id on t_test  
  (cost=0.43..8.45 rows=1 width=4)  
  (actual time=0.013..0.014 rows=1 loops=1)  
    Index Cond: (id = 423425)  
    Heap Fetches: 1  
  Planning time: 0.059 ms  
  Execution time: 0.034 ms  
(5 rows)
```

The important point here is that the optimizer has found out that the index is a lot more beneficial. How did it happen? Internally, PostgreSQL performs a so-called cost-based optimization. This means that an index is not simply taken when it is around. PostgreSQL carefully calculates costs (just some number) and takes the cheapest plan promising the best execution times. In the preceding example, the total cost of the plan using the index is just 8.45. The sequential scan comes up with a stunning estimate of 169247.71 penalty points. Logically, the index is taken because it promises (and provides) far better return times. It is also important to see that there are two times returned by PostgreSQL: the time needed to plan the query and the time to do the actual execution. In the case of short queries, planning time starts becoming relevant.

How an index works

So far, only a simple B-tree has been created. How does a B-tree work internally? In PostgreSQL, Lehman-Yao trees are used. These are binary trees, which have some very nice features. First of all, a B-tree has logarithmic runtime behavior. This means that if the tree grows, the time needed to query the tree won't increase proportionally. One million times more data might result in probably 20 times slower index lookup times (this depends on factors such as distribution of values, amount of RAM available, clock speed, and so on).

To find out more about binary trees, check out http://en.wikipedia.org/wiki/Binary_tree.

There is more – B-trees in PostgreSQL have some more positive features such as Lehman-Yao trees (the trees used in a PostgreSQL B-tree implementation) can be modified by many people at the same time. Therefore, they are a wonderful data structure for scaling up operations. The power of doing things concurrently is important in modern systems and not just for databases.

Avoiding trouble with indexes

Indexes are not always a solution to the problem; they can also be the problem by themselves. The following example outlines a common pitfall. It should be avoided at all costs:

```
test=# CREATE TABLE t_test (id int, x text);
CREATE TABLE
test=# INSERT INTO t_test SELECT x, 'house'
      FROM generate_series(1, 10000000) AS x;
INSERT 0 10000000
test=# CREATE INDEX idx_x ON t_test (x);
CREATE INDEX
```

Before taking a look at the way the index is used, it makes sense to inspect the size of the table as well as the size of the indexes:

```
test=# SELECT
      pg_size_pretty(pg_relation_size('t_test')),
      pg_size_pretty(pg_relation_size('idx_x'));
 pg_size_pretty | pg_size_pretty
-----+-----
422 MB          | 214 MB
(1 row)
```

The table created in this example is 422 MB large (only the table). On top of that, there is 214 MB taken up by the index. So, overall the size of the table with the index is larger than 600 MB.

The problem is that the index is of no use in this example:

```
test=# explain SELECT * FROM t_test WHERE x = 'house';
          QUERY PLAN
-----
Seq Scan on t_test  (cost=0.00..179054.03 rows=10000000
    width=10)
  Filter: (x = 'house'::text)
(2 rows)
```

Despite the index, PostgreSQL will still use a sequential scan. The reason for that is simple: all values in the table are the same, the table is not selective enough, and therefore, the index is more or less useless. Why doesn't PostgreSQL use the index at all? Let's think about it for a moment. The goal of the index is to reduce I/O. If we expect to utilize all rows in the table, we'll have to read the entire table anyway, but if we use the index, it would be necessary to read all index entries on top of the table. Reading the index and the entire table is a lot more expensive than just reading the table. Therefore, PostgreSQL decides on the cheaper strategy.



Keep in mind that PostgreSQL only has to read those parts of the index that are relevant to the query. This example is really a special case designed to require the entire table and the entire index.

However, the index is not always useless:

```
test=# explain SELECT * FROM t_test WHERE x = 'y';
               QUERY PLAN
-----
Index Scan using idx_x on t_test  (cost=0.43..4.45
 rows=1 width=10)
   Index Cond: (x = 'y'::text)
   Planning time: 0.096 ms
(3 rows)
```

In this example, we choose to select a value that is not present in the table (or is expected to be pretty rare). In this case, the index is chosen. In other words, selectivity is the key to the usefulness of an index. If a field is not selective, an index will be pointless.

Detecting missing indexes


In this chapter, the importance of indexing has already been demonstrated. However, what is the best way to figure out whether indexes are missing or not? The `pg_stat_user_tables` view is a system view containing all the relevant information. Here is a query that will definitely be beneficial to you:

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
       idx_scan, seq_tup_read / seq_scan
FROM   pg_stat_user_tables
WHERE  seq_scan > 0
ORDER BY seq_tup_read DESC;
```

The query provides us with `schemaname` and `relname` to identify the table inside the database. The `seq_scan` field will return the number of times the table has been read sequentially. During those sequential scans, the database had to read `seq_tup_read` rows. The `idx_scan` field informs us about the number of index scans, and finally, the average number of rows needed by `seq_scan` is displayed.

The beauty of this query is that those tables that may need an index will show up on top. How do you know? A table that is read sequentially over and over again and that contains a large number of rows is definitely a candidate.

It obviously makes sense to go through the top candidates and check each table. Keep in mind that PostgreSQL can tell you which tables may be a problem, but it won't tell you which columns have to be indexed. Some basic knowledge about the application is needed. Otherwise, you will end up doing guesswork. It is really necessary to figure out which columns your application filters on. At this point, there is no automatic algorithm to check that.

 Make sure that you are not falling into the trap of simply indexing everything. Creating too many indexes won't do any good, as shown in the next section.

Removing useless indexes

Some of you might wonder why too many indexes are bad. While reading data, indexes are no problem at all in 99 percent of all cases (unless the planner makes a bad decision or some caches are too small). However, when it comes to insertion, indexes are a major performance bottleneck.

To compare things, a simple test can be created. One million rows should be added to an empty table as well as to an indexed table. To compare runtimes, `\timing` is used:

```
test=# \timing
Timing is on.
```

The beauty of `\timing` is that the time needed to execute a query is displayed right at the end of the command. First, 1 million rows are added:

```
test=# INSERT INTO t_test SELECT *
      FROM generate_series(1, 1000000);
INSERT 0 1000000
Time: 6346.756 ms
```

It takes around 6 seconds to do this. What if data is added to an empty table?

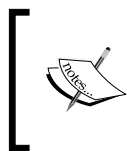
```
test=# CREATE TABLE t_fast (id int, x text);
CREATE TABLE
Time: 92.034 ms
test=# INSERT INTO t_fast SELECT *
      FROM generate_series(1, 1000000);
INSERT 0 1000000
Time: 2078.413 ms
```


Doing the same thing without an index is around three times faster. Keep in mind that the main bottleneck in this example is actually the creation of the data itself. If `generate_series` was free, the difference would have been a lot larger. You have to keep in mind that every index that does not yield any benefit will be a destructive index because it needs space on the disk, and more importantly, it will slow down writes dramatically. If your application is write-bound, additional indexes can be a killer. However, in my professional career, I have seen that write-bound applications are really a minority. Therefore, overindexing can be as dangerous as underindexing. However, underindexing is the more obvious problem because you will instantly see that certain queries are slow. If there are too many indexes, performance problems are usually a bit more subtle.

To prevent problems related to too many indexes, it takes time to check for useless indexes periodically. PostgreSQL has all the onboard means to do that. The best place to start is a system view called `pg_stat_user_indexes`. It tells you which indexes are used and how often:

```
test=# \d pg_stat_user_indexes
View "pg_catalog.pg_stat_user_indexes"
  Column      |  Type  | Modifiers
-----+-----+-----
 relid        | oid    |
 indexrelid   | oid    |
 schemaname   | name   |
 relname      | name   |
 indexrelname | name   |
 idx_scan     | bigint |
 idx_tup_read | bigint |
 idx_tup_fetch| bigint |
```

The relevant field here is called `idx_scan`. It tells us how often a certain index has been utilized. If this index is rarely used, it might make sense to drop it.



Keep in mind that dropping an index on a very small table (maybe on a lookup table) or on a table that is supposed to grow anyway might not be a good idea. It really makes sense to focus on big tables and their indexes.

Dropping indexes can free up a lot of space, reduce I/O, and boost writes.

Solving common problems

After this brief introduction to indexes, the focus of this chapter will shift to troubleshooting. The most common challenges related to indexing and troubleshooting will be described in this section.

Managing foreign keys

One of the most important issues causing trouble and confusion when it comes to indexing has to do with foreign keys. Consider the following example:

```
test=# CREATE TABLE t_person (id    int    PRIMARY KEY,
                                name text);
CREATE TABLE
test=# CREATE TABLE t_car (car_id  int,
                             person_id int REFERENCES t_person (id),
                             info    text);
CREATE TABLE
```

It is necessary to have a unique index irrespective of a primary key on the `t_person` table. However, we often forget to index the other side of the foreign key (in our case, the `t_car` table). Just imagine looking up all cars owned by a certain person. You would definitely want an index scan on the `t_car` table to speed up this query.



Keep in mind that PostgreSQL does not create an automatic index here. It is your task to make this work.

It is highly recommended to ensure that both sides of the equation are indexed.

Indexing geometric data using GiST

Indexing geometric data leads many people into a simple trap – they try to store geometric objects in a traditional way. I have already pointed out this trap in one of the previous chapters.

But how can things be done properly now? The PostGIS project (<http://postgis.net/>) has all that it takes to properly index geometric data. PostGIS is built on the so-called GiST index, which is a part of PostgreSQL. What is a GiST? The idea of GiST is to offer an indexing structure that provides alternative algorithms that a normal B-tree is not capable of providing (for example, operations such as `contains`).

The technical details outlining how GiST works internally would definitely be way beyond the scope of this book. Therefore I advise you to check out http://postgis.net/docs/manual-2.1/using_postgis_dbmanagement.html#idp7246368 for further information on GiST and indexing geometric data.

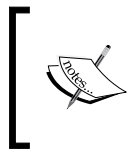
Handling LIKE queries

LIKE is a widely used component of the SQL language that allows the user to perform fuzzy searches. Sometimes, you have an idea about what you are looking for but you are not exactly sure what it is. In these situations, a wildcard search can come in handy and improve user experience dramatically. Just think of the following example: you are reading a fax that has just dropped in. On a poor-quality fax, can you really distinguish a B from an 8 all the time? I guess not. LIKE will fix that by allowing you to put placeholders in a query.

To demonstrate the power of LIKE and the problems arising along with it, I have compiled a small example. The goal is to look for names of cities and villages. Here is the required data:

```
test=# CREATE TABLE t_location (name text);
CREATE TABLE
test=# COPY t_location FROM PROGRAM
      'curl www.cybertec.at/secret/orte.txt';
COPY 2354
```

The CREATE TABLE part is pretty straightforward and easy to understand. To feed data to the system, you can rely on COPY. In this example, sample data can be found on the author's website. The data is read efficiently using a pipe (COPY PROGRAM) fed by a command-line program called curl.



The curl is a command-line tool used to transfer data from or to a server using one of the supported protocols (HTTP, HTTPS, FTP, FTPS, SCP, SFTP, TFTP, DICT, TELNET, LDAP or FILE). The command is designed to work without user interaction.

The file on the server has only one column, so this works out pretty neatly. If COPY is successful, 2354 rows should be in your table.

Simple LIKE queries

Let's focus our attention on a simple query now. All rows starting with Wiener should be retrieved:

```
test=# SELECT * FROM t_location
      WHERE name LIKE 'Wiener%';
      name
-----
Wiener Neustadt
Wiener Neudorf
Wienerwald
(3 rows)
```

The query works like a charm. However, the execution plan of the query uncovers a real performance problem:

```
test=# explain SELECT * FROM t_location
      WHERE name LIKE 'Wiener%';
      QUERY PLAN
-----
Seq Scan on t_location  (cost=0.00..43.42
 rows=1 width=13)
  Filter: (name ~~ 'Wiener% '::text)
 Planning time: 0.078 ms
(3 rows)
```

The table will be read sequentially. To fix this problem, a special index is needed:

```
test=# CREATE INDEX idx_name
      ON t_location (name text_pattern_ops);
CREATE INDEX
```

You cannot just deploy a normal index on the columns. To make LIKE work, a special operator class is needed in most cases. For text, this operator class is called `text_pattern_ops` (for varchar, it would be `varchar_pattern_ops`).

What is an operator class? It is basically a strategy used by the index. Just imagine a simple B-tree. Small values will go to the left edge, and large values to the right edge of the tree. In simple terms, an operator class will know what is considered to be small and what is considered to be a large value. Operator classes are a way to teach your index how to behave.

In this example, the special operator class for the index is necessary in order to enable an index scan in the first place. Here is how it works:

```
test=# explain SELECT * FROM t_location
      WHERE name LIKE 'Wiener%';
      QUERY PLAN
-----
Index Only Scan using idx_name on t_location
(cost=0.28..8.30 rows=1 width=13)
Index Cond: ((name ~>= ~ 'Wiener'::text)
AND (name ~< ~ 'Wienes'::text))
Filter: (name ~~ 'Wiener% '::text)
Planning time: 0.197 ms
(4 rows)
```

Just relax for a second and take a look at the way PostgreSQL uses the index. Basically, it says that `wiener%` means larger than `wiener` and smaller than `wienes`. In other words, PostgreSQL uses a clever rewrite to solve the problem.

Note that the percent sign has to be at the end of the query string. Given the indexing strategy, this sign is not allowed somewhere in the middle or at the beginning.

Keep in mind that those small numbers can vary. In the case of small tables, there is quite some noise in your performance data. So if one of your queries takes a millisecond more or less, don't panic! This is normal.

More advanced LIKE queries

In many cases, it is really necessary to have a percent sign on both sides of the search string:

```
test=# explain SELECT * FROM t_location
      WHERE name LIKE '%Wiener%';
      QUERY PLAN
-----
Seq Scan on t_location (cost=0.00..43.42 rows=1 width=13)
Filter: (name ~~ '%Wiener% '::text)
Planning time: 0.120 ms
(3 rows)
```

As you can see, we end up with a sequential scan again. Clearly, a sequential scan is not an option on large tables.

To fix this problem, you can turn to PostgreSQL extensions. Extensions are a nice feature of PostgreSQL that allow users to easily integrate additional functionality with the server. To solve the problem of indexing `LIKE`, an extension called `pg_trgm` is needed. This special extension provides access to so-called **Trigrams**, a concept especially useful when it comes to fuzzy searching:

```
test=# CREATE EXTENSION pg_trgm;
CREATE EXTENSION
```

`CREATE EXTENSION` enables the extension so that an index capable of handling trigrams can be created:

```
test=# CREATE INDEX idx_special ON t_location
      USING gist(name gist_trgm_ops);
CREATE INDEX
```

This time, a GiST index has to be created. The `gist_trgm_ops` can serve as the operator class to handle our little problem.

Don't worry about the precise inner working of this index type. It will nicely solve the problem of fuzzy searching, as shown in this example:

```
test=# explain SELECT * FROM t_location
      WHERE name LIKE '%Wiener%';
          QUERY PLAN
-----
Index Scan using idx_special on t_location
 (cost=0.14..8.16 rows=1 width=13)
  Index Cond: (name ~~ '%Wiener%'::text)
Planning time: 0.232 ms
(3 rows)
```

The index will be able serve our request.



Keep in mind that trigram-based indexes are a highly important feature, as many applications tend to suffer from indexing problems related to fuzzy searches. GiST and trigrams can really help to stop these problems from causing major performance bottlenecks.

Finding good matches

The queries shown so far are not the only queries that can cause issues. Just imagine searching for a name. Maybe, you are not aware of how to spell it precisely. You issue a query and the result is just an empty list. Empty lists can be pretty frustrating for end users. Frustration has to be avoided at any cost, so a solution is needed.

The solution comes in the form of the so-called distance operator (<->). It works like this:

```
test=# SELECT 'PostgreSQL' <-> 'PostgreSQL';
      ?column?
-----
    0.230769
(1 row)
```

The distance between PostgreSQL and PostgressSQL is 0.23, so it is quite close. The distance can be between 0 (meaning identical) and 1 (totally different).

However, the question is not so much about the actual distance between two words but more about what is closest. Let's imagine we are looking for a village called Gramatneusiedl. Now an application designer cannot expect that somebody is able to spell this tricky name. So maybe, an end user might go and ask for Kramatneusiedl. A precise match would not return even a single row, which is somewhat frustrating to the end users.

Nearest neighbor search (KNN) can come to the rescue and return those four villages that are closest to our search string:

```
test=# SELECT *, name <-> 'Kramatneusiedl'
      FROM   t_location
      ORDER BY name <-> 'Kramatneusiedl'
      LIMIT 4;
      name      | ?column?
-----+-----
Gramatneusiedl | 0.52381
Klein-Neusiedl | 0.76
Pötzneusiedl   | 0.791667
Kramsach       | 0.809524
(4 rows)
```

The results are pretty good. Most importantly, the village the end user has really looked for is also among those results.

Just imagine an application trying to solve this without the ability to use KNN. Performance would go down the drain instantly.

KNN works also for numbers and the like. To use this functionality with numeric values, it is necessary to install the `btree_gist` extension.

Fixing full-text search

After focusing on some highly important and common problems related to indexing, it is time to discuss the most important pitfalls in the area of full-text indexing.

Not using full-text search at all

One of the most common mistakes made is not to use full-text search at all. Many people tend to use `LIKE` instead of proper, PostgreSQL-style full-text searching. Doing that can open a Pandora's box and result in various performance issues.

To demonstrate the point, some sample text can be loaded. In this case, Shakespeare's *Hamlet* has been selected, which can be downloaded freely from www.gutenberg.org (an archive of free books). To load the data, end users can turn to `curl`, just as was shown before in the previous example. PostgreSQL will read the data from the pipe:

```
test=# CREATE TABLE t_text (payload text);
CREATE TABLE
test=# COPY t_text FROM PROGRAM 'curl http://www.gutenberg.org/cache/epub/2265/pg2265.txt';
COPY 5302
```

In this example, 5302 lines of text have been loaded. The goal is to look for all lines that contain `company` and `find`. Here is what you should not do:

```
test=# SELECT * FROM t_text
      WHERE payload ILIKE '%company%'
      AND payload ILIKE '%find%';
      payload
-----
What company, at what expense: and finding
(1 row)
```


One line has been returned. This does not look like a problem but it is. Let's assume you are not sure whether you are looking for `company` or `companies`. If you google for `car`, you would be happy with `cars` as well, wouldn't you? The same applies here. So, instead of asking PostgreSQL for a fixed string, it makes sense to process the data in the table before indexing it. The technical term here is *stemming*. Here is how it can be done:

```
test=# SELECT to_tsvector('english',
    'What company, at what expense: and finding');
           to_tsvector
-----
'compani':2 'expens':5 'find':7
(1 row)
```

English language rules are used to stem the text just returned. What comes out is a set of three tokens. Note that some words are missing, as they are simple stop words. There is no semantic payload in `what`, `at`, or `and`, so those words can safely be skipped.

The next important observation is that words are stemmed. Stemming is a process that treats words such that they are reduced to some sort of root (not really the root of the word but usually pretty similar to it). The following listing shows an example of stemming:

```
test=# SELECT to_tsvector('english',
    'company, companies, find, finding');
           to_tsvector
-----
'compani':1,2 'find':3,4
(1 row)
```

As you can see, `company` and `companies` are both reduced to `compani`, which is not a real word but a good representation of the actual meaning. The same applies to `find` and `finding`. The important part now is that this vector of stemmed words is indexed; the word showing up in the text isn't. The beauty here is that if you are searching for `car`, you will also end up finding `cars` – a nice, little improvement!

There is one more observation; mind the numbers showing up behind the stemmed word. Those numbers represent the position of the word in the stemmed string.



Keep in mind that stemming is a highly language-dependent process. Therefore, the language must be passed to the call.

To ensure good performance, it is necessary to create an index. For full-text searches, GIN indexes have proven to be the most beneficial index. Here is how it works for our copy of *Hamlet*:

```
test=# CREATE INDEX idx_gin ON t_text
      USING gin (to_tsvector('english', payload));
CREATE INDEX
test=# SELECT *
      FROM t_text
      WHERE to_tsvector('english', payload)
             @@ to_tsquery('english', 'find & company');
             payload
-----
What company, at what expense: and finding
(1 row)
```

Once the index has been created, it can be used. The important part here is that the `tsvector` function created by the stemming process is compared to our search string. The goal is to find all lines containing `find` and `company`. Exactly one line is returned here. Notice that `company` shows up before `finding` in the sentence. The search string is stemmed as well, so this result is possible.

To prove the preceding point, the following list contains the execution plan showing the index scan:

```
test=# explain SELECT *
      FROM t_text
      WHERE to_tsvector('english', payload)
             @@ to_tsquery('english', 'find & company');
      QUERY PLAN
-----
Bitmap Heap Scan on t_text
  (cost=20.00..24.02 rows=1 width=33)
    Recheck Cond: (to_tsvector('english'::regconfig,
      payload) @@ '''find'' & ''compani''':tsquery)
    -> Bitmap Index Scan on idx_gin
      (cost=0.00..20.00 rows=1 width=0)
        Index Cond: (to_tsvector('english'::regconfig,
      payload) @@ '''find'' &
      ''compani''':tsquery)
Planning time: 0.096 ms
(5 rows)
```

PostgreSQL's full-text indexing provides a lot more functionality than is actually described here. However, given the scope of this book, it is not possible to cover all of this wonderful functionality.

Full-text search and sorting

Full-text search is powerful. However, there are also some corner cases you should be aware of. Consider the following type of query:

```
SELECT * FROM product WHERE "fti_query" ORDER BY price;
```

The use case for such a query can be simple: find all books containing certain words in the title, and sort them by price; or find all restaurants containing *pizza* in their description and sort them by postal code (in most countries, a number).

The way a GIN index is organized is perfect for full-text search. The problem is that inside GIN, the item pointers pointing to the row in the table are sorted by position, not by some criteria such as price.

So let's assume the following query:

```
SELECT *  
FROM products  
WHERE field @@ to_tsquery('english', 'book')  
ORDER BY price;
```

During my career as a PostgreSQL consultant, I came across a database that contained 4 million product descriptions with the word *book*. What will happen here is that PostgreSQL will have to find all of these books and sort them by price. Fetching 4 million rows from an index and having to sort 4 million rows is actually a death sentence to the performance of the query.

People might argue now, "Why not narrow down the price window?" In this case, it won't help because a price range of 20 to 50 euros would still include, say, 3 million products.

As of PostgreSQL 9.4, it is very hard to fix this type of query. Therefore, developers have to be very careful and aware of the fact that a GIN index does not necessarily provide sorted output. Therefore, the amount of data returned from this type of query is directly proportional to the runtime of the SQL request.

Future versions of PostgreSQL may solve this problem once and for all. In the meantime, however, people have to live with the fact that you cannot expect an infinite number of rows to be sorted in no time.

Summary

This chapter covered the important aspects of PostgreSQL indexing. The focus was on common queries causing common indexing-related problems faced by many people. Some aspects of full-text searching were covered as well.

In the next chapter, you will learn how to read data efficiently.

4

Reading Data Efficiently and Correctly

Originally, this chapter was called "Reading Data Efficiently." Thinking about it for a while actually convinced me that "and Correctly" should be appended to the title. The reason is that this chapter is not just about speed; it is also about getting things right and ensuring that the result can be generated fast. What good is a fast result if it happens to be just plain wrong?

The following issues will be covered:

- NULL handling
- Fixing joins
- Using cursors
- Synchronized seq scans
- Prepared queries
- Indexes and foreign keys

Understanding the power of NULL

Some of you might wonder why a section about NULL is included in this book. Remember that it is all about troubleshooting, and trust me, NULL can be a real source of trouble if its pitfalls are not understood properly. NULL is actually a pretty old thing. E. F. Codd (the father of relational databases) mentioned the concept of NULL in a 1975 paper. The idea of NULL is therefore central to the world of relational systems.

However, NULL has not always been widely understood and accepted among SQL developers. Many people have experienced troubles in the face of NULL. In this section, we will shed some light on this mysterious thing and solve the riddle of NULL.

The first and most important thing to know is that NULL does not mean empty string; it means unknown. Not knowing is not the same as empty. For example, if you know that your wallet is empty, then it means you know that there is nothing in it. Not knowing what is in your wallet is a totally different situation. It is important to keep this in mind. The golden rule is that NULL means undefined.

Seeing NULL in action

After this bit of theoretical introduction, it is time to dive into some real action to see how NULL really behaves. The first important observation is that NULL is not the same as an empty string:

```
test=# SELECT NULL = '';
?column?
-----
(1 row)
```

Many people expect this query to return false. In fact, the return value is NULL. Why? Imagine you are holding nothing in your left hand. We don't know what is in your right hand. Do both hands contain the same thing? We don't know. The second hand might be empty as well but we don't know for certain. Therefore, the answer to this question is unknown.

To check whether a certain value is NULL or not, `IS NULL` has to be used. In this case, the answer will be definitely false:

```
test=# SELECT '' IS NULL;
?column?
-----
f
(1 row)
```

The same logic applies to the next example:

```
test=# SELECT NULL = NULL;
?column?
-----

(1 row)

test=# SELECT NULL IS NULL;
?column?
-----
t
(1 row)
```

Again, the first query returns `NULL`, as we don't know if both sides are the same. Remember that the unknown item in your left hand may not be the same as the unknown item in your right hand.

NULL and storage

The good news is that `NULL` does not need space in the disk. In PostgreSQL, every row contains a so-called `NULL` bitmap. It indicates whether a field happens to be `NULL` or not. Therefore, an individual `NULL` value does not need any additional space in the disk.

This is not quite true for an empty string. A `varchar` string will need a fair amount of space even if it is empty because it is a perfectly defined value.

Fixing disastrous joins

`NULL` is not the only thing crying out for disaster. Joins are also a good source of trouble if not used properly. If you have the feeling that you know everything about joins and that this section can be skipped, give it a chance. Things might not be as easy as expected.

Create demo data for joins

To show what can go wrong during a join, the best thing to do is to create a simplistic example. In this case, two tables are created:

```
test=# CREATE TABLE a (aid int);
CREATE TABLE
test=# CREATE TABLE b (bid int);
CREATE TABLE
```

Then some rows can be added:

```
test=# INSERT INTO a VALUES (1), (2), (3);
INSERT 0 3
test=# INSERT INTO b VALUES (2), (3), (4);
INSERT 0 3
```

Note that the tables are quite similar but not identical.

For the sake of completeness, here is how common values can be found:

```
test=# SELECT * FROM a, b WHERE aid = bid;
 aid | bid
-----+-----
    2 |    2
    3 |    3
(2 rows)
```

Understanding outer joins

A simple inner join is not what we are most interested in when talking about joins and problems in general. A lot more trouble comes from the so-called outer joins.

Let's look for all values on the left-hand side and match those that are present on the right-hand side:

```
test=# SELECT *
      FROM a LEFT JOIN b
      ON (aid = bid);
 aid | bid
-----+-----
    1 |
    2 |    2
    3 |    3
(3 rows)
```

The query is pretty simple and actually quite risk free as well, but what happens if an `AND` clause is added to the `ON` clause? Many people tend to do that to filter the data and reduce the result:

```
test=# SELECT *
      FROM a LEFT JOIN b
        ON (aid = bid AND bid = 2);
 aid | bid
-----+-----
    1 |
    2 |    2
    3 |
(3 rows)
```

Most people don't expect this result. Keep in mind that the `ON` clause is here to tell PostgreSQL what to join. In a `LEFT JOIN` keyword, the left side is always taken completely. The `ON` clause merely indicates what fits the left-hand side. The scary part here is that the number of rows returned by the query is *not* reduced at all. It has to be that way.

Somebody might argue that this is an easy problem to fix. Why not just move the filter to the `WHERE` clause, as shown here?

```
test=# SELECT *
      FROM a LEFT JOIN b
        ON (aid = bid)
      WHERE bid = 2;
 aid | bid
-----+-----
    2 |    2
(1 row)
```

Only one row is returned now. Then what is the point of using a `LEFT JOIN` keyword here? The outer join is totally useless in this case because the filter strips all the rows created by the outer join.

The trouble with outer joins is that a wrong outer join will hit our support desk and will be nicely covered up as a performance problem. Here is why:

```
test=# SELECT avg(aid), avg(bid)
      FROM a LEFT JOIN b
        ON (aid = bid AND bid = 2);
      avg      |      avg
-----+-----
2.0000000000000000 | 2.0000000000000000
(1 row)
```

The results look correct at first glance. However, the customer expects that `bid = 2` actually reduces the amount of data processed. People wonder why indexes are not used and complain about bad performance. In reality, this is not a performance problem but a logical problem because the query is most likely not what was originally intended by the developer.

A second problem with outer joins has to do with NULL values:

```
test=# SELECT count(aid), count(bid), count(*)
        FROM a LEFT JOIN b
           ON (aid = bid AND bid = 2);
count | count | count
-----+-----+-----
      3 |      1 |      3
(1 row)
```

The `count(bid)` function returns 1 while `count(*)` function will return 3. Keep in mind that `count(*)` will count rows regardless of their contents. However, `count(column)` will count all values that don't happen to be NULL. As there is just one number, the correct answer is 1 here. The `count(*)` function and outer joins, if not used wisely, can result in a deadly mixture—ready to explode!

Before writing an aggregate, always try to fix the outer join first and then think twice. Where shall NULL values be expected? What kind of count is really needed? Is the AND operator inside the ON clause really doing what you think it is doing? Whenever there is AND in the ON clause, be attentive and check twice. The same applies to `count(*)` versus `count(column)`. An error along these lines can affect you in a brutal way.

Let's focus our attention on FULL JOIN now:

```
test=# SELECT *
        FROM a FULL JOIN b
           ON (aid = bid);
aid | bid
-----+-----
  1 | 
  2 |  2
  3 |  3
   |  4
(4 rows)
```

The idea of a FULL JOIN keyword is to find everything on the left and match it with everything on the right. All values from each of the sides will be shown in the result.

After all that you have learned, do you expect the following result?

```
test=# SELECT *
      FROM a FULL JOIN b
      ON (aid = bid AND aid = 2);
 aid | bid
-----+-----
   1 |
   2 |    2
   3 |
     |    4
     |    3
(5 rows)
```

Again, the AND clause will only tell the system what goes together; it does not reduce data—quite the opposite. In this example, the amount of data returned is even increased by one row.

Reading large amounts of data

Some applications require large amounts of data that have to be read by the database. Usually, two major cases can be distinguished:

- Reading large amounts of data and shipping them to the client
- Reading large amounts of data so that the server can process them

In this section, both the cases will be covered because they have slightly different requirements.

Making use of cursors

Cursors are a fundamental concept provided by all reasonable relational databases. However, cursors are often ignored and not used the way they should be, which leads to problems all along the way.

Why would anybody want to use a cursor, and what is it all about anyway? Consider the following example:

```
SELECT * FROM very_large_table;
```

For PostgreSQL, this is no problem at all. It will calculate the result and send it to the client server. However, the client may instantly crash because it will try to store all of the data in the memory. If the table is really large, the client may just not cope with all of the data coming from the server. The client will consume a lot of memory or even die. The biggest problem is that whether your client will survive or not will depend on the amount of data in the table—a sure recipe for disaster, which is also difficult to identify.

In general, your applications should be designed in a way such that they will always work, regardless of the amount of data and regardless of the result sets coming back from the server. A cursor is here to fix this very problem. The idea behind a cursor is to allow the client to keep only as much data around as will be needed at a specific time.

To make use of cursors, the `DECLARE` command has to be used:

```
test=# \h DECLARE
Command:      DECLARE
Description:  define a cursor
Syntax:
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
          CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

In our example, the goal is to read 10 rows without having to keep all 10 rows in the memory at once. Demo data can be created easily:

```
test=# CREATE TABLE t_test AS
      SELECT * FROM generate_series(1, 10) AS x;
SELECT 10
```

In the next step, the data should be read by the application. The most important thing to do here is to wrap the operation inside a transaction (in this book, transactions will also be covered in detail in one of the later chapters). The cursor can then be easily defined:

```
test=# BEGIN;
BEGIN
test=# DECLARE mycur CURSOR FOR SELECT * FROM t_test;
DECLARE CURSOR
```

Note that creating the cursor does not do anything at this point. It merely means that PostgreSQL is now prepared to send a lot of data to you. The real work is done as soon as `FETCH` is called:

```
test=# FETCH NEXT FROM mycur;
x
```

```

---
1
(1 row)

```

`FETCH NEXT` will yield one row from the result set. It is not possible to call `FETCH NEXT` for every row in the result. The trouble with `FETCH NEXT` is that it is really really slow. If you fetch 1 million rows one by one, it means a lot of network overhead. Therefore, it is highly recommended to fetch data in larger buckets. In general, buckets of several hundred to several thousand rows make perfect sense, as memory consumption and network overhead would be in perfect balance. However, don't take this as a fixed guideline. Things depend on your type of application, your data, your network, and so on. It can make sense to experiment.

Here is how more than one row can be fetched at a time:

```

test=# FETCH 3 FROM mycur;
x
---
2
3
4
(3 rows)

```

It is now possible to loop through the entire result as needed.

To finish the operation, it is enough to commit the transaction. All cursors will be closed automatically:

```

test=# COMMIT;
COMMIT

```

It is important to note that after a `COMMIT` statement, the cursor is not there anymore:

```

test=# FETCH NEXT FROM mycur;
ERROR:  cursor "mycur" does not exist

```

By default, a cursor is only visible inside a transaction, and therefore, very long cursor-related operations can also cause issues such as replication conflicts and so on. If you really want a cursor to survive a transaction, you have to create a so-called `WITH HOLD` cursor. In this case, however, you have to take care of cleaning up the cursor again by calling `CLOSE`:

```

test=# \h CLOSE
Command:      CLOSE
Description:  close a cursor
Syntax:
CLOSE { name | ALL }

```

WITH HOLD cursors should be handled with care. The most important observation is that a WITH HOLD cursor has to materialize its result, which can have a lot of impact on the amount of storage required on the database server.

Synchronized scanning

Sometimes, reading is not about shipping data to the client but all about processing data server side. This is exactly when synchronized seq scans come into play.

Let's imagine the following scenario: a 10 TB table contains some old reporting data. It is stored on a server containing a couple of traditional mechanical disks. Let's consider the following query:

```
SELECT error_type, count(*) FROM big_table GROUP BY 1;
```

Assuming that `error_type` has just a couple of different values, PostgreSQL will turn to a seq scan and process the query. The I/O system might deliver data at the rate of 500 MB per second. All disks will be spinning nicely.

Let's assume now that the query shown here has already processed 2 TB of data. Then somebody comes along and fires up a second, similar query. Even if the data has already been read by the previous job, it has definitely fallen out of the cache by now. Therefore, it has to be read once again. But things are even worse; those spinning disks will start jumping between the first and the second query. The more the queries added, the more random the disk behavior. After about 10 queries, the total throughput may drop to approximately 60 MB per second, which means 6 MB per database connection. Clearly, this is not a desirable performance level and is unacceptable.

Synchronized seq scans come to the rescue. Let's assume the first query has already processed 2 TB of the table at the full speed of 500 MB per second. When the second (and similar) query kicks off on the same table, PostgreSQL will make it start reading at around 2 TB. The second query will continue from there. The advantage of synchronizing these scans is that one can benefit from the cache entries generated by the other one. Reading from the disk once can, under good conditions, feed both (or even more) of the scans with data. Thus, performance is not 500 MB per second divided by two minus additional random access per scan, but ideally 500 MB per second multiplied by the number of scans. Running scans in parallel can improve speed dramatically by drastically reducing the I/O bottleneck and turning it slowly into a CPU bottleneck, which is a lot easier to scale.

The important thing is that synchronized sequential scans are on by default, and in general, it is a good idea to ensure that this setting is really set to on:

```
test=# SHOW synchronize_seqscans ;
synchronize_seqscans
-----
on
(1 row)
```

You can enjoy their full potential on the default configuration.

In many cases, it makes more sense to execute things in parallel than in sequence because many analytical jobs may share the same I/O.



Keep in mind that this works only for sequential scans.

Understanding prepared queries

This section is all about prepared queries. What is a prepared query? Consider the following example:

```
SELECT * FROM website WHERE domain = 'www.cybertec.at';
```

PostgreSQL will parse the query to check for syntax errors and so on. Then it will go through the rewrite system and hit the PostgreSQL optimizer, which then comes up with a good plan. Finally, the executor will perform all the steps selected by the query planner. If this type of query is executed over and over again for different domains, the parsing and planning process will also have to run over and over again, which is very time consuming.

Prepared queries try to solve exactly that problem. The idea is that the backend process will cache the plan and reuse it just in case it is needed again. The beauty is that sending the parameters instead of the complete query can be enough. Usually, prepared statements are happening behind the scenes when desired (this is usually done through the driver if the driver is prepared for it). However, you can also prepare queries manually:

```
test=# \h PREPARE
Command:      PREPARE
Description:  prepare a statement for execution
Syntax:
PREPARE name [ ( data_type [, ...] ) ] AS statement
```


As mentioned already, something similar is done by the driver. However, it also works manually:

```
test=# PREPARE myplan(int) AS SELECT $1;
PREPARE
test=# EXECUTE myplan(10);
?column?
-----
      10
(1 row)
```

In this example, a plan called `myplan` that accepts one parameter is created. Parameters passed to a prepared query are numbers such as `$1`, `$2`, `$3`, and so on. To run a plan, `EXECUTE` is used here.

Of course, there is some protocol-level support for optimizing this process, and it is highly recommended to make use of this functionality.

However, in some cases, there can be problems. If a client application creates thousands or even millions of prepared plans, the memory consumption of the backend may grow. To figure out whether your active database connection has created too many prepared statements, a system view can be consulted:

```
test=# \d pg_prepared_statements
View "pg_catalog.pg_prepared_statements"
Column          | Type                | Modifiers
-----+-----+-----
name            | text                |
statement       | text                |
prepare_time    | timestamp with time zone |
parameter_types | regtype[]           |
from_sql        | boolean              |
```

The `pg_prepared_statements` view tells us which statements have been prepared in our database connection.

If you want to remove a statement from a backend, you can use the `DEALLOCATE` command:

```
test=# \h DEALLOCATE
Command:      DEALLOCATE
Description:  deallocate a prepared statement
Syntax:
DEALLOCATE [ PREPARE ] { name | ALL }
```

Another important point is that prepared queries are not only about speed but also about SQL injections. A prepared query is a good way to avoid security loopholes altogether.

Summary

This chapter was all about reading data efficiently. It covered aspects such as cursors as well as synchronized seq scans and prepared plans. Of course, a chapter like this can never be complete, as the field of reading data is simply an infinite one.

The next chapter will cover transactions and locking — two important topics relevant to any relational database system.

5

Getting Transactions and Locking Right

Transactions are a core technique of every professional relational database system. In fact, it is pretty hard to imagine a world without transactions these days. Atomic deletions, proper locking, and all the functionalities provided by a modern relational system are simple and expected from a modern system, and many applications rely on them. Therefore, this chapter is all about transactions and locking.

The following topics will be covered:

- The PostgreSQL transaction model
- Basic locking
- `FOR UPDATE`, `FOR SHARE`, and `NOWAIT`
- Locking tables
- Understanding transaction isolation levels
- Indexes and foreign keys
- Transactions and sequences

The PostgreSQL transaction model

The PostgreSQL transaction model differs quite a lot from other systems. In many cases, these differences are a root source of trouble. In this section, some basic examples have been included to make you aware of the way PostgreSQL works.

The first thing to take into account is that a transaction block has to be started with `BEGIN` statement and finished with a simple `COMMIT` statement. All statements inside the transaction block have to be correct:

```
test=# BEGIN;
BEGIN
test=# SELECT now();
           now
-----
2014-09-22 13:27:24.952578+02
(1 row)

test=# SELECT now();
           now
-----
2014-09-22 13:27:24.952578+02
(1 row)

test=# COMMIT;
COMMIT
```

Note that `now()` will return the transaction time. Therefore it is always the same, regardless of how often it is used. Many people like to use `DEFAULT now()` in their column definitions. Always keep in mind that `DEFAULT now()` sets the column to transaction time, and it stays constant even in a long transaction. This is a handy way to determine whether all rows have been written in the same transaction.

The second important thing to note is that a transaction must be free of mistakes:

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?
-----
1
(1 row)

test=# SELECT 1 / 0;
ERROR:  division by zero
test=# SELECT 1;
ERROR:  current transaction is aborted, commands
        ignored until end of transaction block
test=# SELECT 1;
ERROR:  current transaction is aborted, commands
        ignored until end of transaction block
test=# COMMIT;
ROLLBACK
```

In this example, an error occurs after a successful start of the transaction. The main issue here is that the transaction will never recover, even if properly working statements are issued after the mistake! Remember that a transaction has to be correct from the beginning to the end. Otherwise, all commands after the error will be ignored by the system.


Understanding savepoints

If the conditions we just mentioned cannot be achieved, it is wise to consider using savepoints. Basically, a savepoint is a mechanism used to jump back inside a transaction. It can be used effectively to avoid errors and to ensure success. The following example shows how it works:

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?
-----
      1
(1 row)

test=# SAVEPOINT s1;
SAVEPOINT
test=# SELECT 1 / 0;
ERROR:  division by zero
test=# ROLLBACK TO SAVEPOINT s1;
ROLLBACK
test=# COMMIT;
COMMIT
```

After the initial `SELECT` statement, a savepoint is created. Note that the savepoint, as the name suggests, is used later on to identify the spot to return to. After an error occurs, it is still possible to jump to a specific savepoint. The main advantage is that the transaction can be committed nicely in this case.

 Keep in mind that a savepoint is only valid inside a transaction. It does not exist after a transaction has committed or rolled back.

Understanding basic locking and deadlocks

Now that you have learned about the basic PostgreSQL model followed by the PostgreSQL system, it is time to dig into locking. In many cases, locking turns out to be a bottleneck. So, it is definitely worth inspecting things.

To get started, a simple demo table can be created like this:

```
test=# CREATE TABLE t_test AS SELECT 1 AS id;
SELECT 1
```

The table contains only one row, which is enough to demonstrate some basic concepts of locking. In the first example, the goal is to demonstrate what happens when two users try to increment the value in the table at the same time:

User 1	User 2
BEGIN;	
	BEGIN;
UPDATE t_test SET id = id +1 RETURNING *;	
	UPDATE t_test SET id = id +1 RETURNING *;
	wait ...
COMMIT;	wait ...
	COMMIT;

The goal is to allow two users to increment the value in the table at the same time. The important thing to observe is that the second `UPDATE` statement has to wait until the first `UPDATE` statement has committed. It is important to mention that we are talking about a row level lock and not a table lock here. PostgreSQL only locks those rows affected by the `UPDATE` statement, ensuring that other people are not facing serious bottlenecks.

The second important observation is the result; it is ensured that the outcome at the end of the example is always 3. As soon as the first transaction commits, the second `UPDATE` rereads the now-committed row and increments it. While this may seem like a logical issue, it is not widely known to many people in spite of its wide implications.

Let's expand the example a little:

```
test=# INSERT INTO t_test VALUES (5);
INSERT 0 1
test=# SELECT * FROM t_test;
 id
----
```

```

3
5
(2 rows)

```

Let's perform two updates now:

User 1:	User 2:
BEGIN;	BEGIN;
UPDATE t_test SET id = 4 WHERE id = 3;	UPDATE t_test SET id = 6 WHERE id = 4;
	UPDATE t_test SET id = 4 WHERE id = 3;
UPDATE t_test SET id = 6 WHERE id = 5;	
... deadlock detected ...	
	COMMIT;

The first UPDATE statement of both the users works as expected. The second UPDATE statement of User 2 has to wait for User 1 because the same row is changed. The important part, however, is the final UPDATE. Both users wait on each other, so nobody can win this game. Therefore, PostgreSQL steps in and resolves the hopeless situation. The exact error message is as follows:

```

test=# UPDATE t_test SET id = 6 WHERE id = 5;
ERROR:  deadlock detected
DETAIL:  Process 27988 waits for ShareLock on
        transaction 1131992; blocked by process 28084.
Process 28084 waits for ShareLock on transaction
        1131990; blocked by process 27988.
HINT:   See server log for query details.
CONTEXT:  while updating tuple (0,5) in relation
          "t_test"

```

The deadlock is resolved automatically. All that the user has to do is to catch the error and try again.



Many users think that a deadlock is something nasty, horribly dangerous, and disastrous. Actually, this is not the case. A deadlock is a totally natural thing, and it simply happens once in a while. Don't panic! Instead, think logically about the reasons for it.

How can deadlocks be avoided? Actually, there is no general rule. Personally, I found it useful to try to update data in a sorted order. In many cases, this works well and cures the reasons for a deadlock (which is out of order changes). Besides this, there is not much that can be done to get around this issue.

When talking about deadlocks, many people instantly ask about a PostgreSQL parameter called `deadlock_timeout`:

```
test=# SHOW deadlock_timeout;
      deadlock_timeout
-----
1s
(1 row)
```

There is a general misconception regarding this parameter in the heads of many. The parameter actually tells us how long the system waits before checking for a deadlock. Deadlock detection is fairly expensive, and therefore, PostgreSQL waits for one second before it initializes the check.

Locking in FOR UPDATE mode

In this section you will be introduced to a problem that has bugged generations of database developers – wrong or missing locking.

Let's assume you are reading some data and you definitely intend to change it later on, once you have read it. But what if the data was already gone by the time you were performing your `UPDATE`? I assume it would not do you any good.

In PostgreSQL, reading can occur concurrently, and many people can read the same piece of data at the same time without interfering with each other. Therefore, reading does not provide sufficient protection against other folk who are planning to modify data as well. The result can be nasty. Consider the following example:

```
BEGIN;
SELECT * FROM tab WHERE foo = bar;
UPDATE tab SET foo = "value made in the application";
COMMIT;
```

A transaction like what is shown here is not safe. The reason is simple: What if people execute the same thing concurrently? It may happen that they overwrite each other's changes instantly. Clearly, this would cause major issues.

To prevent this from happening, `SELECT ... FOR UPDATE` comes to the rescue.

Consider the following example:

User 1:	User 2:
BEGIN;	BEGIN;
SELECT * FROM t_test WHERE 1 > 0 FOR UPDATE;	
... do some work ...	SELECT * FROM t_test WHERE 1 > 0 FOR UPDATE;
... do some work ...	wait ...
COMMIT;	
	... returns latest data ...
	... do some work and commit ...

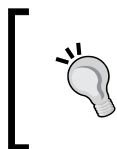
In this example, `SELECT ... FOR UPDATE` locks the rows returned by the query, just like `UPDATE` would've locked those rows. The beauty here is that the second transaction has to wait until the first transaction either issues a `COMMIT` statement or exits. The advantage here is that the second transaction can already build on the changes of the first query, ensuring that nothing is lost.

Avoiding performance bottlenecks

However, if used carelessly, `SELECT ... FOR UPDATE` can cause serious performance issues. Consider the following example:

```
SELECT ... FROM a, b, c, d, e, f, g WHERE ... FOR UPDATE;
```

Let's assume all you want to do is to modify `a`. The rest of the tables are just used as lookup tables. PostgreSQL has no way to figure out that you are planning to modify just one table. Therefore, it has to lock all rows returned by the query, in all tables. While this is a necessary thing for PostgreSQL to do, it is likely to cause nasty contentions involving lookup tables.



Contention means you may witness any of the following: comparatively low CPU usage, little disk wait, bad performance, and so on. If the CPU is not working at its top speed and if nothing is moving forward, you are most likely facing contentions caused by locking somewhere.

There are two approaches to fix this problem. The first is to use `SELECT ... FOR UPDATE NOWAIT`. This will make a query stop if a lock cannot be obtained. It can come in handy if you want to avoid waiting indefinitely. The second option is even more sophisticated. Consider the following example:

```
SELECT ... FROM a, b, c, d, e, f, g WHERE ...  
FOR UPDATE OF a, b;
```

In this case, PostgreSQL knows that only two tables will most likely be updated in the next couple of steps, which will improve locking substantially.

Avoiding table locks

In some cases, `SELECT ... FOR UPDATE` is not enough, and locking an entire table is necessary. To create a table lock, PostgreSQL provides a simple command:

```
test=# \h LOCK  
Command:      LOCK  
Description:  lock a table  
Syntax:  
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [  
NOWAIT ]
```

where lockmode is one of:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE  
| SHARE UPDATE EXCLUSIVE | SHARE  
| SHARE ROW EXCLUSIVE | EXCLUSIVE  
| ACCESS EXCLUSIVE
```

Note that PostgreSQL has eight lock levels here, ranging from `ACCESS SHARE` all the way up to `ACCESS EXCLUSIVE`. These allow you to define locking in a very fine-grained way. Suppose you want to make sure that you are the only person allowed to read and write a table. Then the lock mode needed is `ACCESS EXCLUSIVE`. It perfectly ensures that nobody else can even look at that table. If nobody is supposed to modify a table but reading is perfectly fine, `EXCLUSIVE` is the option of choice.

On the other end of the spectrum, there is a lock held by a simple read called `ACCESS SHARE`. It can nicely coexist with other reads and does not harm others. It only conflicts with `ACCESS EXCLUSIVE`, which is needed by `DROP TABLE` and similar commands at <http://www.postgresql.org/docs/9.4/static>.

Locking a table is sometimes necessary, but it should never be done in a careless way because it can have nasty side effects and block other transactions.



More information about locking and a detailed overview of all potential conflicts can be found at:
<http://www.postgresql.org/docs/9.4/static/explicit-locking.html>.

Transaction isolation

In this section, you will be introduced to an important topic called transaction isolation. In my long career as a PostgreSQL database consultant, I have seen countless scenarios in which transaction isolation has caused enormous problems. Sometimes, people even desperately replaced their hardware to get around issues that were definitely not related to hardware.

Demonstrating read committed mode

So, what is transaction isolation? The idea behind transaction isolation is to give users a tool to control what they see inside a transaction. A report might have different requirements than a simple OLTP application, but let's not get lost in plan theory. Instead, let's look at a real example with some data:

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test VALUES (4), (5);
INSERT 0 2
```

We've got two rows. Now let's see what happens if two transactions battle for the data at the same time:

User 1:	User 2:
BEGIN;	
SELECT sum(id) FROM t_test; sum ----- 9 (1 row)	
	INSERT INTO t_test VALUES (9); INSERT 0 1

User 1:	User 2:
<pre>SELECT sum(id) FROM t_test; sum ----- 18 (1 row)</pre>	
<pre>COMMIT;</pre>	

The first `SELECT` statement returns 9, which is the expected value. While the first transaction is still active, some other user adds some more data. Most people now expect the first transaction to return 9, but this is not the case. Just because user 1 is in a transaction, it does not imply that its view of the data never changes. By default, PostgreSQL uses something called read committed mode. This means that a transaction will see changes committed by somebody else every time a new statement is started.



Note that running statements never change their view of the world just because somebody else makes a change. In read committed mode, a snapshot is taken at the beginning of a statement. Once it is running, it stays as it is.

The fact that the result changes due to some other commit can have serious implications. If a report consists of more than just one query, it can mean that data shown on page 5 does not reflect what it says on page 29 of the report. Of course, such discrepancies don't show up all the time but only sometimes – when somebody happens to commit a change in some other session at the same time.

The support requests reaching us here – at Cybertec Schönig & Schönig GmbH – sound something like this, "Sometimes, PostgreSQL returns the wrong result. We have already updated the database, but sometimes there are still mistakes. Interestingly, this only happens on the live system. We have tested things on the development system and it seems to work just fine. We have even replaced the hardware of our production system because this really feels like a broken RAM. The bug still exists but it seems less likely."

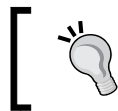
There are a couple of observations to be made from this little (fake) support request: Yes, the "problem" never shows up in the lab. The reason is that there are not as many people in the lab as there are on production. Thus, nobody changes data while you are testing. Upgrading the database certainly does not help because PostgreSQL behaves in the desired way. Upgrading hardware won't change things either, but it makes sense in saying that "the problem" will be less likely because new hardware is usually faster. The faster your hardware, the less likely it is that two transactions run at the same time, thus reducing the odds of causing a problem in the first place.

Using repeatable read

To ensure that a transaction does not change its view of the world, `REPEATABLE READ` mode can be used. Here is a table showing how it works:

User 1:	User 2:
<pre>BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ; BEGIN</pre>	
<pre>SELECT sum(id) FROM t_test; sum ----- 18 (1 row)</pre>	
	<pre>DELETE FROM t_test; DELETE 3</pre>
<pre>SELECT sum(id) FROM t_test; sum ----- 18 (1 row)</pre>	
<pre>COMMIT;</pre>	

In this case, the transaction sticks to the data it sees. The concurrent `DELETE` statement, which removes all data from the table, does not harm the report. In other words, side effects can be avoided by turning to the repeatable read mode.



People often ask if repeatable read has a performance penalty compared to read committed mode; it doesn't. Both isolation levels offer the same throughput. There is no performance bottleneck.

As mentioned before, repeatable read is perfect for all kinds of reports. However, it is not the golden solution for everything. Normal OLTP operations are still best done in the read committed mode to ensure that data is as up to date as possible.

Beyond repeatable read

There is more than just read committed and repeatable read. The highest transaction isolation level supported by PostgreSQL is serializable. The original idea behind the serializable transaction isolation level back in the early days was to totally serialize transactions to avoid all kinds of interaction.

However, executing transactions in a sequential way is clearly not what you would expect from a modern relational database such as PostgreSQL. The problem with sequential execution is scalability. There is no way to scale out if a system executes one thing after another on a single CPU because everything else is just locked. In PostgreSQL, serializable mode is way more sophisticated. In fact, it pretends to do sequential execution, but internally, it executes things as parallel as possible.

What is the purpose of a serializable transaction? Sometimes, freezing your snapshot is not enough. Consider the following example:

```
CREATE TABLE t_prison (guard text);
```

Let's assume that you want to ensure that there is always exactly one guard around (for the sake of the example, no primary key is allowed). Now a transaction may look like this:

```
BEGIN;
SELECT count(*) FROM t_prison;
if count > 1
    delete a guard
COMMIT;
```

What happens if two folks execute this at the same time? Both would see that two people are in the table. Each of them would decide to delete one. If we are unlucky, one would remove "Bob" while the other one would remove "Alice". Voilà! All doors to the prison would be wide open. It does not matter whether repeatable read or read committed is used. Two people can check for the content at the same time, and therefore, see the same data, leading to the same conclusion.

The solution to this problem is serializable mode. It will ensure that only transactions that are independent of each other can run in parallel. Transactions that involve the same data (or same predicates) will conflict with each other.

The beauty of this is that you can basically get around a table lock, thus improving scalability. However, you will need a retry loop in case transactions are aborted due to conflicts.

Inspecting locks

In many cases, locks are a major issue for most system administrators. If a transaction blocks some other operations, it is usually a problem. End users will complain that "something hangs." However, you can rely on the fact that PostgreSQL never hangs without a reason. Usually, it is simple and all about locking.

In this section you will learn how to detect locks and how to figure out who is locking which transaction.

Let's do a simple example considering the following data:

```
test=# CREATE TABLE t_test AS SELECT *
      FROM generate_series(1, 10) AS id;
SELECT 10
```

The goal is to have two concurrent `SELECT FOR UPDATE` operations reading slightly different data. Those two operations will lock each other, and then the goal will be to figure out which operation is blocking which transaction.

The first `SELECT FOR UPDATE` command selects everything larger than 9:

```
test=# BEGIN;
BEGIN
test=# SELECT * FROM t_test WHERE id > 9 FOR UPDATE;
      id
----
      10
(1 rows)
```

The second transaction locks everything larger than 8:

```
test=# BEGIN;
BEGIN
test=# SELECT * FROM t_test WHERE id > 8 FOR UPDATE;
```

Logically, this transaction has to wait for the first transaction. In this case, it is obvious what is going on, but how can it be checked if you don't know?

The first thing to check is always `pg_stat_activity`. It will reveal what is going on and which operations are active:

```
test=# \x
Expanded display is on.
test=# SELECT pid, query, waiting
      FROM pg_stat_activity;[ RECORD1 ]-----
-----
pid      | 4635
query    | SELECT * FROM t_test WHERE id > 9 FOR UPDATE;
waiting  | f
[ RECORD2 ]-----
pid      | 4637
query    | SELECT * FROM t_test WHERE id > 8 FOR UPDATE;
waiting  | t
```



```
[ RECORD3 ]-----
pid      | 4654
query    | SELECT pid, query, waiting
        | FROM pg_stat_activity;
waiting  | f
```

In this example, pid 4637 indicates that the transaction is waiting on something. The waiting=true is a reliable indicator of potential trouble. It means that a transaction is stuck. The next thing to do is to check an additional system view called `pg_locks` view. This view tells us which locks have been granted and which locks haven't been.

The following query detects locks that have not been granted:

```
test=# SELECT locktype, granted, transactionid,
        mode
        FROM pg_locks WHERE granted = 'f';
 locktype | granted | transactionid | mode
-----+-----+-----+-----
transactionid | f | 1812 | ShareLock
(1 row)
```

It seems that somebody is waiting on transactionid 1812. This is an important observation. In the next step, information about transaction 1812 can be retrieved:

```
test=# SELECT granted, transactionid, mode, pid
        FROM pg_locks
        WHERE transactionid = 1812;
granted | transactionid | mode | pid
-----+-----+-----+-----
t | 1812 | ExclusiveLock | 4635
f | 1812 | ShareLock | 4637
(2 rows)
```

There are two processes that have something to do with transactionid 1812. One of them has the lock and the other does not. Obviously, pid 4635 is blocking pid 4637. To solve this problem, the database connection that is causing issues can be terminated:

```
SELECT pg_terminate_backend(4635);
```

But don't do that yet. There is more to be found out.

Whenever a lock is held on a conflicting row, there is an entry made in `pg_locks`:

```
test=# SELECT pid, relation, page, tuple
        FROM   pg_locks
        WHERE   page IS NOT NULL;
 pid | relation | page | tuple
-----+-----+-----+-----
 4637 | 16385    | 0    | 10
(1 row)
```

Obviously, there is a lock on the tenth tuple in the first (0) page in relation 16385. The `pid` indicates that the waiting transaction is waiting on this row.

In many cases, it is even possible to see the row that is causing issues. The so-called `ctid` is an identifier pointing to a row in the storage system. By querying the row, we have good chances (no guarantee) of seeing the row that is causing issues:

```
test=# SELECT ctid, * FROM t_test
        WHERE   ctid = '(0, 10)';
 ctid | id
-----+---
(0,10) | 10
(1 row)
```

The problem is caused by row number 10, which is the very row locked by our first `SELECT FOR UPDATE` command. Inspecting the row that is causing issues can give you vital insights into what is really going on in the system. Before killing a database connection, it is always wise to check out which kind of data is really causing trouble. Sometimes, this can reveal problems in the underlying application.

Summary

In this chapter, you were introduced to locking and transactions. As pointed out, locking is a central issue affecting both scalability as well as correctness of your data. It is important to keep an eye on locking. Especially, transaction isolation is an important issue, but is often forgotten by many developers, which in turn can lead to buggy applications. In addition to that, harsh locking can lead to scalability issues as well as to timeouts.

In addition to what you have seen, locks can be reverse engineered to find out who is waiting for whom.

6

Writing Proper Procedures

After understanding transactions and locking, it is time to focus on writing proper procedures to avoid widespread problems and known issues. The goal is to cover issues that bug many people around the globe. This chapter will introduce some of the most common pitfalls when it comes to writing procedures for PostgreSQL.

The following topics are on our agenda:

- Choosing the right language
- Managing procedures and transactions
- Optimizing procedures for indexing
- Avoiding security problems
- Controlling memory

All of these topics will help you to make sure that your procedures offer high performance at low risk.

Choosing the right language

One of the cool features of PostgreSQL is the privilege for you to write your stored procedures in almost any language you want. Some languages such as SQL, PL/pgSQL, PL/Perl, PL/Python, and PL/Tcl are shipped along with the core of the system. You can also add other languages such as PL/sh, PL/v8, and many more to PostgreSQL to make sure that you always have the language of your choice at hand when you want to write a procedure.



The listing at https://wiki.postgresql.org/wiki/PL_Matrix contains an overview of all procedural languages available.



To enable a procedural language, you can call `CREATE LANGUAGE`.

```
test=# \h CREATE LANGUAGE
Command:      CREATE LANGUAGE
Description:  define a new procedural language
Syntax:
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
           HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR
           valfunction ]
```

To enable PL/Perl, just call this command:

```
CREATE LANGUAGE plperl;
```

If PL/Perl is installed, this will work, and you can write procedures in Perl from now on.

If you decide to add a language, it definitely makes sense to check out the documentation of the language that should be installed and act accordingly.

Trusted versus untrusted

When deciding on a procedural language, you should keep one thing in mind: you have to decide whether to use a trusted or an untrusted language. What does this actually mean? If you load, for example, Perl in untrusted mode (`plperl_u`), the external language is allowed to do anything. This poses a real security threat because a user writing a procedure can actually delete files, send e-mails, or do any other nasty stuff. Of course, in a secure environment, this is not desirable at all. Therefore, it is better to restrict Perl to the so-called taint mode, which does not provide system calls. It offers only the basics of the language and bans any interaction with the outside world. For procedural languages embedded inside PostgreSQL, this is definitely the way to go. Untrusted Perl should only be used in rare, exceptional cases.

Not all programming languages provide you with such a choice. Languages such as PL/sh and the like are available only in untrusted mode, making them unsuitable for ordinary users who do not have superuser permissions.

Managing procedures and transactions

Once you have decided on a procedural language, it is time to get started with real work. The first important fact you have to keep in mind is that PostgreSQL relies heavily on an object-oriented feature called function overloading. This term means that various functions can have the same name but accept different input parameters.

One function may be called `func(int, int)`, while some other function may be called `func(text, text)`. Always keep in mind that the entire signature of the function is what counts:

```
test=# CREATE FUNCTION mysum(int, int)
      RETURNS int AS ' SELECT $1 + $2 '
      LANGUAGE 'sql';
CREATE FUNCTION
test=# CREATE FUNCTION mysum(text, text)
      RETURNS text AS ' SELECT $1 || $2 '
      LANGUAGE 'sql';
CREATE FUNCTION
```

In this case, two functions have been defined. The first will take two integers, and the second takes two text values. Depending on how the function is called, different portions of code will be executed:

```
test=# SELECT mysum(10, 20);
mysum
-----
      30
(1 row)
```

The behavior of the function changes as the input data types change:

```
test=# SELECT mysum('10', '20');
mysum
-----
    1020
(1 row)
```

You are advised to be very careful here.

Using transactions to avoid trouble


Let's discuss a very common problem many people all over the world face. Experience has shown that many people commit their code way too early. Consider the following flow:

1. Somebody created `func(int, int)`.
2. At some point, the programmer finds out that actually, `func(int8, int8)` is needed.
3. The new function is accidentally created alongside the old function.
4. Nobody notices that there are two functions around until it is too late.

This is not a theoretical edge case. It happens all the time all over the world. A mistake like this can cost people dearly.

One way to reduce the odds of such a mistake is the use of transactions. Remember that in PostgreSQL, transactions can even be used along with DDLs (`CREATE FUNCTION`, `ALTER TABLE`, and so on). By (simply) not committing a change while you are working on the code, you can prevent side effects, such as leftovers, and this helps to avoid the problem of cleaning up things during and after development. Writing database code is not like writing code in a programming language. If you remove a function from the code, it is gone. In a database, removing a function from a text file somewhere does not necessarily remove the function from the database system. `BEGIN`, `COMMIT`, or `ROLLBACK` can help to reduce the pain.

[



Here is my personal tip on how to do it:

```
[hs@jacqueline deploy]$ cat START.sql
BEGIN;
\i module_1.sql
\i module_2.sql
```

]

The `module_1.sql` and `module_2.sql` files contain all of the code. Note that there is no `COMMIT` statement. It is only added after all the modules are somewhat ready for the rest of the world. The `\i` function is a very convenient tool used to include modules and load them as parts of one transaction. To those of you who are using the normal Linux command line to develop software, this might be very helpful.

Understanding transactions and procedures

So far, you have seen how transactions and function creation work, but there are other aspects here as well. One important thing is that inside a function, the programmer cannot start or end transactions; this makes sense. Consider the following command:

```
SELECT * FROM func();
```

Let's assume `func()` returns 1 million rows. There is no way to open a transaction after returning 200,000 rows or so. The same applies to commit. You cannot just commit at any place you want inside an operation.

The golden rule is that a function is always a part of a transaction. It never starts or ends transactions. A function always operates within the scope of the statement running it.

However, there is one thing you can take advantage of: a PL/pgSQL function can use so-called exceptions. Practically, an exception can help you to achieve the same thing as with a savepoint. Here is an example:

```
CREATE FUNCTION failure(int) RETURNS int AS $$
    DECLARE
        v_one    ALIAS FOR $1;
    BEGIN
        RETURN v_one / 0;

        EXCEPTION WHEN division_by_zero THEN
            RETURN 0;
    END;
$$ LANGUAGE 'plpgsql';
```

This procedure will definitely cause an error. As you have seen in the previous chapters, an error actually leads to a failing transaction. However, the exception helps us catch the error and ensures that the transaction can be recovered.

Procedures and indexing

In this section, you will learn the basic pitfalls of indexing and procedures. One of the most common issues is that people forget to define their procedures. In PostgreSQL, there are four types of procedures:

- VOLATILE
- STABLE
- IMMUTABLE
- [NOT] LEAKPROOF


A volatile procedure may return different outputs for the same input parameters within the same transaction:

```
test=# SELECT random(), random();
         random          |          random
-----+-----
 0.906597905792296 | 0.368819046299905
(1 row)
```


The `random()` function is supposed to return different values all the time. This is the core purpose of a random generator. Logically, this has implications when it comes to indexing:

```
SELECT * FROM tab WHERE field = random();
```

Can PostgreSQL use an index here? The answer is no, because what will the engine look up in the B-tree? The value is supposed to change by definition for each row. Therefore, indexes are forbidden here. All existing indexes on the column will simply be ignored.

[ VOLATILE is a major source of performance problems because people are left wondering why no indexes are used.]


`STABLE` tells PostgreSQL that this function is providing a more deterministic output than a `VOLATILE` function:

```
test=# BEGIN;
BEGIN
test=# SELECT now();
           now
-----
2014-10-15 11:04:51.555648+02
(1 row)

test=# SELECT now();
           now
-----
2014-10-15 11:04:51.555648+02
(1 row)

test=# COMMIT;
COMMIT
```

Within the same transaction, the function will always return the same output, given the same input. The `now()` function is a classic candidate here because within the same transaction, it will always return the same time (transaction time).

[ The `clock_timestamp()` function is the volatile counterpart of `now()` because it returns the real time.]

IMMUTABLE is the most stable level a function can reach. It basically means that the result is always the same, regardless of the transaction:

```
test=# SELECT cos(10);
      cos
-----
-0.839071529076452
(1 row)
```

The cosine of 10 is always -0.839, regardless of transactional content, age of the programmer, mood of the customer, or the database system in use. It is a law of nature—a fact.

STABLE and IMMUTABLE functions can be indexed, like this:

```
SELECT * FROM tab WHERE field = now()
```

The previously mentioned query is a perfect candidate for using an index on a field. Because `now()` stays constant throughout the transaction, PostgreSQL actually knows what to look up in the index and will therefore consider indexes as an optimization. The same would apply to `pi()`, `sin()`, `cos()`, and so on.

Always define a function properly (if possible) to avoid performance-related problems:

```
test=# CREATE FUNCTION dummy(int) RETURNS int AS
$$
    SELECT $1;
$$ LANGUAGE 'sql' IMMUTABLE;
CREATE FUNCTION
```

LEAKPROOF versus NOT LEAKPROOF

LEAKPROOF is, unfortunately, a bit more complicated than the other three volatility types. However, it is as important as they are.

To understand LEAKPROOF, it is important to dig a little deeper into the working of the planner. Consider the following statement:

```
SELECT * FROM tab WHERE a = 10 AND b = 20;
```

Should the system filter on `a` or on `b` first? Well, it depends. If `b` promises a greater reduction in the amount of data returned, it is better to filter on `b` first to ensure that there is not much data left when the second filter on `a` is applied.

In other words, PostgreSQL is allowed to reorder those quals (restrictions) and decide on the best way to execute those conditions. Why is this relevant? Well, here is an example:

```
CREATE FUNCTION slow_func(int4) RETURNS int4 AS $$
    BEGIN
        EXECUTE 'SELECT pg_sleep(1)';
        RAISE NOTICE 'slow_func: %', $1;
        RETURN $1;
    END;
$$ LANGUAGE 'plpgsql' IMMUTABLE
    COST 10;
```

```
CREATE FUNCTION fast_func(int4) RETURNS int4 AS $$
    BEGIN
        RAISE NOTICE 'fast_func: %', $1;
        RETURN $1;
    END;
$$ LANGUAGE 'plpgsql' IMMUTABLE
    COST 100;
```

We have two functions; one is fast and the other is slow. To prove my point, I assigned more penalty points (costs) to the faster function. All that the two functions do is wait for some time to pass. Here is the demo data:

```
CREATE TABLE t_test (id int4);
INSERT INTO t_test SELECT * FROM generate_series(1, 20);
```

Now a view is created. All it does is return even numbers and call the faster function to see whether it returns zero:

```
CREATE VIEW v AS
    SELECT *
    FROM   t_test
    WHERE  id % 2 = 0
           AND fast_func(id) = 0;
```

In the next step, the view is used and a filter is applied on top of the view:

```
SELECT * FROM v WHERE slow_func(id) = 0;
```

PostgreSQL is facing a couple of choices now: Which function should be called first, the fast one or the slow one? And in which order should the filters be applied?

Here is what happens:

```
\timing
SELECT * FROM v WHERE slow_func(id) = 0;
NOTICE:  slow_func: 2
NOTICE:  slow_func: 4
   *snip*
NOTICE:  slow_func: 20
 id
----
(0 rows)
Time: 10012.580 ms
```

PostgreSQL will go for the slow function first because it promises to be faster due to lower costs.

Now the trouble is as follows: imagine you are not allowed to see the complete content of `t_test`. For some security reasons, you are allowed to see only even numbers. PostgreSQL is allowed to reorder those filters. Who says that `id%2` and `fast_func` are executed first? What if PostgreSQL would decide on calling `slow_func` first? A simple `RAISE NOTICE` function would reveal the data that you are actually not allowed to see. In our case, PostgreSQL figures out that `%2` should be the first operation, but if this was not there, there would be a major security problem.

This is exactly when `LEAKPROOF` comes into play. It actually tells the planner whether a function can be pushed through a so-called security barrier or not.

The next listing shows how a security barrier can be created:

```
CREATE VIEW v WITH (security_barrier) AS
    SELECT *
    FROM   t_test
    WHERE  id % 2 = 0
           AND fast_func(id) = 0;
```

Just add `WITH (security_barrier)` to the definition of the view, and PostgreSQL won't be allowed anymore to push `NOT LEAKPROOF` functions to the view to reorder things. `LEAKPROOF` functions can be reordered freely.

If you are dealing with security-critical data or many different users and procedures, it is definitely important to think about `LEAKPROOF` as against `NOT LEAKPROOF` to avoid nasty surprises.

Understanding memory

When a procedure is written, it makes sense to think for a moment about memory consumption. There are three important issues here:

- Procedures and cursors
- Handling set returning functions
- Assigning memory parameters to functions

In this section, you will be guided through all of these topics.

Procedures and cursors

The first thing to be discussed is the way large result sets should be handled inside a procedure. In normal programming, the general rule is that in the case of large amounts of data, a cursor is needed. The idea behind a cursor is that only small portions of data are fetched and processed at a time.

Languages such as PL/pgSQL provide functionality to open, close, and fetch data from cursors. However, in most cases, you don't actually need to rely on this functionality. The beauty of a procedural language is that internally, everything is a cursor! Here is an example:

```
CREATE FUNCTION process() RETURNS int AS
$$
    DECLARE
        v_rec          RECORD;
    BEGIN
        FOR v_rec IN SELECT * FROM large_table
        LOOP
            -- do something
        END LOOP;

        RETURN 0;
    END;
$$ LANGUAGE 'plpgsql';
```

In this example, the procedure loops through a large result set. PL/pgSQL will automatically use an implicit cursor. You don't have to worry about memory at all because internally, data is processed smoothly without ever materializing completely in the memory.

In short, there is no need to fool around with cursor handling and explicit, manual cursor management.

Handling set-returning functions

While loops are processed nicely by the system, there are functions that do need some attention. I am talking about the so-called set-returning functions. These are functions that actually return more than a row. The set-returning function that is most widely used in this book is `generate_series`:

```
SELECT * FROM generate_series(1, 10) AS x;
```

The main issue here is that PostgreSQL has to calculate the entire function, store its output somewhere, and continue from there. This means that if you call `generate_series` (or a similar function) with a large range, everything has to be kept in the memory as long as things are needed. Dealing with two thousand rows or so is definitely not a problem here, as modern hardware is easily able to handle that. However, if you want to return hundreds of millions of rows, you need to switch to a different technique. You should not return such large result sets with a set-returning function.

In many cases, returning a cursor can help.

Assigning memory parameters to functions

Tuning PostgreSQL also involves modifying memory parameters in `postgresql.conf`. In many cases, setting things globally is quite sufficient. Most applications seen in the field these days are perfectly fine with global settings. However, in some cases, passing `config` parameters directly to a function can make sense.

Consider the following example:

```
CREATE FUNCTION func() RETURNS int AS
$$
    DECLARE
    BEGIN
        -- some code
    RETURN 0;
```

```
END;  
  
$$ LANGUAGE 'plpgsql' SET work_mem = '1 GB';  
  
SELECT func();
```

The most important issue is the `SET` statement at the end of the `CREATE FUNCTION` statement. It tells the system that inside this function, 1 GB of `work_mem` integer should be available. Outside the function, other values can be used.

You can specify any of the runtime-alterable settings within the function definition. One example is of time zones. Maybe, you want to run your function according to UTC time. No problem! Just tell the function to do so.

Of course, you can change those settings later on with `ALTER FUNCTION` if it turns out that a change is needed.

Summary

In this chapter, you learned about procedures. The goal was to cover the most common pitfalls when it comes to procedures in a short, compact chapter. Things such as function volatility, trusted and untrusted languages, and memory parameters were discussed.

The next chapter will be all about monitoring PostgreSQL. You will also learn how to detect bottlenecks in the system.

7

PostgreSQL Monitoring

Troubleshooting is virtually impossible if you don't have the information you need to actually track down a problem. For carrying out troubleshooting, data is everything. Without data and without the ability to see what is going on, the situation is practically hopeless.

In this chapter, you will be guided through the following topics:

- Checking the overall database behavior
- Checking for conflicts
- Finding I/O bottlenecks
- Inspecting system internals
- Chasing down slow queries
- External tools

Those topics will help you keep an eye on your system.

Checking the overall database behavior

The first thing to do when approaching a system is to check the overall health and see what is currently going on in the system. Is the system using too much CPU? Or maybe, too much memory? Is the disk I/O fine?

Checking pg_stat_activity

The best place to start is `pg_stat_activity`, a system view listing open database connections. Since PostgreSQL 9.2, `pg_stat_statement` has been extended a little, and a lot more information such as the state of a database connection can be seen:

```
test=# \d pg_stat_activity
          View "pg_catalog.pg_stat_activity"
   Column          |          Type          | Modifiers
-----+-----+-----
 datid             | oid                    |
 datname           | name                   |
 pid               | integer                |
 usesysid          | oid                    |
 username          | name                   |
 application_name   | text                   |
 client_addr       | inet                   |
 client_hostname    | text                   |
 client_port       | integer                |
 backend_start      | timestamp with time zone |
 xact_start        | timestamp with time zone |
 query_start       | timestamp with time zone |
 state_change      | timestamp with time zone |
 waiting           | boolean                |
 state             | text                   |
 backend_xid        | xid                    |
 backend_xmin       | xid                    |
 query             | text                   |
```

Let's go through those columns. The first two columns tell us which database a person is connected to. The first column contains the object ID of the database, and the second column contains the name of the database in plain text. Then information about the user connected is displayed. Again, two fields are used: one with the object ID, and another with the name of the user in plain text.

Then comes the `application_name` field. The `application_name` field can be set freely by the end user (or the application). It is very helpful to debug applications because, if used wisely, it can tell you where a request is generated inside your application. Consider the following example:

```
test=# SET application_name TO 'script.pl:456';
SET
test=# SHOW application_name;
```

```

application_name
-----
script.pl:456
(1 row)

```

As you can see, it is just a plain text name, which can contain anything.

The next three columns (`client_*`) will tell you where a connection originates from. You can detect the hostname, the port, and so on. In case of trouble, it allows you to solve the problem on the box causing the issue.

The `backend_start` and `xact_start` fields tell you when a database connection or the transaction was started.



Keep an eye on very old and long-running transactions. They can have a serious impact on your `VACUUM` policy. Remember that a row can only be cleaned out when it cannot be seen by a single transaction anymore. Long-running transactions can therefore have undesirable side effects.

The next important fields are `query_start`, `state`, `state_change`, and `query`. These four fields have to be seen in combination, and not one by one. The `query_start` column indicates when the last query has started. When the state is set to active, the query is still active at that point of time. When the state stays idle, the query is not active anymore and was terminated when `state_change` was set. Therefore, the difference between `query_start` and `state_change` is the runtime of the previous query in this case. Of course, `query` is the query we are talking about.

After the `waiting` column, which was discussed in the previous chapter, there are `backend_xid` and `backend_xmin`. These two columns give you an insight into the transactional behavior of the system (transaction IDs and so on).

To avoid trouble, the following points can be checked:

- Are there any waiting queries?
- Are there long-running transactions causing `VACUUM` issues?
- Are there no connections at all or is there any excess number of connections?
- Are there queries listed that have taken (or are taking) way too much time?

Given those valuable insights, you can fix a couple of quite frequent issues.

Checking database-level information

Once you are done with `pg_stat_activity`, you might want to turn your attention to `pg_stat_database`. The idea of `pg_stat_database` is to have one row per database:

```
test=# \d pg_stat_database
          View "pg_catalog.pg_stat_database"
   Column      |      Type      | Modifiers
-----+-----+-----
 datid         | oid             |
 datname       | name            |
 numbackends   | integer         |
 xact_commit   | bigint          |
 xact_rollback | bigint          |
 blks_read     | bigint          |
 blks_hit      | bigint          |
 tup_returned  | bigint          |
 tup_fetched   | bigint          |
 tup_inserted  | bigint          |
 tup_updated   | bigint          |
 tup_deleted   | bigint          |
 conflicts     | bigint          |
 temp_files    | bigint          |
 temp_bytes    | bigint          |
 deadlocks     | bigint          |
 blk_read_time | double precision|
 blk_write_time| double precision|
 stats_reset   | timestamp with time zone |
```

There are a couple of important fields here. First of all, there are the object ID and the name of the database. Then the view reveals the number of currently open database connections (`numbackends`). However, there is some more subtle information here that does not seem to be widely used: `xact_commit` and `xact_rollback`. The `xact_commit` and `xact_rollback` fields contain the number of successful and aborted transactions since the start of time, respectively. Why are these fields so important? In a typical application, the ratio of `COMMIT` to `ROLLBACK` tends to stay pretty constant over time. An application may face 5 percent `ROLLBACK` commands over a long period of time. What does it mean if this value is suddenly jumping to 35 percent or more? Most likely, some software update must have introduced some syntax error somewhere. That bug might have been undetected so far. However, if used properly, `pg_stat_database` might give clues that something might be wrong.

The next few fields tell us how much data was read, added, updated, deleted, and so on. This might be interesting to see, but usually, it does not reveal deeply hidden problems. These fields are more of a general guideline, telling you how much is going on in your system.

The next field is all about `conflicts`. More information about conflicts can be found in the next section of this chapter, *Checking for conflicts*.

The `temp_files` and `temp_bytes` fields are two highly important fields. They will tell us whether the database has done extensive I/O through temporary files. When does this happen? Temporary files are written if a lot of data is sorted and so on; for example, `ORDER BY` clauses, `CREATE INDEX`, or some `OVER` clauses lead to heavy sorting. If `work_mem` or `maintenance_work_mem` is not properly set, it can manifest in high I/O activity when it comes to temporary files. Another reason for high temporary I/O is simply bad SQL statements (for example, too much data returned).

There are two more fields that can give you a clue about I/O: `blk_write_time` and `blk_read_time`. The purpose of these two fields is to measure the time spent on writing and reading data to and from the operating system respectively. However, these two fields are not active by default:

```
test=# SHOW track_io_timing;
      track_io_timing
-----
off
(1 row)
```

Unless `track_io_timing` has been activated in `postgresql.conf`, there is no data available. If you are fighting with I/O problems and disk wait, it is definitely recommended to turn this setting on to detect the database that is causing some or all of the I/O trouble.



Note that in some rare cases, timing can decrease performance by a small amount. In most cases, however, this overhead is not relevant.

Finally, `stats_reset` tells us when the statistics has been reset.

Detecting I/O bottlenecks

The `track_io_timing` command can give you clues about the database that is causing most of the load. However, there is more; `pg_stat_bgwriter` contains a lot of information about the system's I/O behavior:

```
test=# \d pg_stat_bgwriter
View "pg_catalog.pg_stat_bgwriter"
      Column      |      Type      |
-----+-----+
checkpoints_timed | bigint         |
checkpoints_req   | bigint         |
checkpoint_write_time | double precision |
checkpoint_sync_time | double precision |
buffers_checkpoint | bigint         |
buffers_clean     | bigint         |
maxwritten_clean  | bigint         |
buffers_backend   | bigint         |
buffers_backend_fsync | bigint         |
buffers_alloc     | bigint         |
stats_reset       | timestamp with time zone |
```

Understanding what is going on in the I/O system can be vital. This understanding also includes checkpoints. In PostgreSQL, there are two reasons for a checkpoint: either the system has run out of WAL segments (a requested checkpoint) or the system has exceeded `checkpoint_timeout` (a timed checkpoint). The `pg_stat_bgwriter` table will tell you what has caused a checkpoint to take place. If all your checkpoints are timed, it means that there is no point in increasing `checkpoint_segments`. If all your checkpoints are requested, raising `checkpoint_segments` and `checkpoint_timeout` might be a good idea to solve a potential performance bottleneck.

Then there are `checkpoint_write_time` and `checkpoint_sync_time`. The `checkpoint_write_time` field is all about the total amount of time that has been spent in the portion of checkpoint processing where files are written to the disk. The `checkpoint_sync_time` field represents the amount of time spent on flushing to the disk during a checkpoint.

The `buffers_checkpoint` field tells people about the number of blocks written during a checkpoint. The `buffers_clean` field tells people how many blocks are written by the background writer.

When the background writer writes blocks from the shared buffers area to the operating system, it does so in rounds. When it stops because it has already written too many blocks per round, we can see that in `maxwritten_clean`. If the background writer cannot keep up with the write load, backends (database connections) are also allowed to perform I/O. The number of blocks written by backend is counted in `buffers_backend`. Sometimes, a backend even has to call `fsync` to flush to the disk. Those flush calls are counted in `buffers_backend_fsync`. Finally, there is `buffers_alloc`, which tells us about the number of buffers allocated.

The key to use `pg_stat_bgwriter` is to figure out which component is doing I/O. In general, the `bgwriter` is supposed to do most of the I/O itself. However, under heavy load, it might be necessary for normal database connections to do some of the work on their own. In PostgreSQL 9.2, a checkpoint process has been added to take away some work from the regular background writer process. Both checkpointer and writer processes write to the same files. However, the checkpointer writes all of the data that was dirty at a certain time (the start of the checkpoint), regardless of how often it was used since dirtied. On the other hand, the background writer writes data that hasn't been used recently, regardless of when it was first dirtied. Neither of the two knows or cares whether the data being written was committed or rolled back or is still in progress.

Checking for conflicts

Once the I/O system has been properly inspected, it makes sense to stop for a minute and see whether there are major conflicts going on in the system:

```
test=# \d pg_stat_database_conflicts
View "pg_catalog.pg_stat_database_conflicts"
      Column      |  Type  | Modifiers
-----+-----+-----
 datid            | oid    |
 datname          | name   |
 confl_tablespace | bigint |
 confl_lock       | bigint |
 confl_snapshot   | bigint |
 confl_bufferpin   | bigint |
 confl_deadlock   | bigint |
```

The `pg_stat_database_conflicts` view informs us about conflicts happening in the system. The view actually lists quite a number of different conflicts. However, not all of them are actually relevant. The most important of them are locking conflicts, snapshot conflicts, and deadlocks. A locking conflict happens if one transaction is waiting for another transaction to complete. A snapshot conflict can happen when a transaction is relying on a snapshot that is way too old (this happens in high isolation levels).

Finally there are deadlocks, which have already been covered extensively in this book.

Chasing down slow queries

After inspecting active queries, checking for I/O problems, and locking, it might be interesting to see which queries are actually causing most of the trouble. Without knowing the actual time-consuming queries, it is pretty hard to improve things in the long run. Fortunately, PostgreSQL provides a module called `pg_stat_statements` that is capable of tracking queries and providing administrators with statistical information about those queries.

To use this module, it has to be enabled:

```
test=# CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION
```

Note that the module has to be enabled inside a database. The system views created by this module will only live in this database for now.

However, we are not done yet:

```
test=# SELECT * FROM pg_stat_statements;
ERROR:  pg_stat_statements must be loaded via
        shared_preload_libraries
```

The module is actually loaded when the postmaster starts up. It cannot be loaded on the fly because the information created must survive the disconnection.

To load the module at startup, change `shared_preload_libraries` in `postgresql.conf` to this:

```
shared_preload_libraries = 'pg_stat_statements'
```

Then the database has to be restarted. Now the system will automatically collect all of the information and present it in a system view:

```
test=# \d pg_stat_statements
View "public.pg_stat_statements"
```

Column	Type	Modifiers
userid	oid	
dbid	oid	
queryid	bigint	
query	text	
calls	bigint	
total_time	double precision	
rows	bigint	
shared_blks_hit	bigint	
shared_blks_read	bigint	
shared_blks_dirtied	bigint	
shared_blks_written	bigint	
local_blks_hit	bigint	
local_blks_read	bigint	
local_blks_dirtied	bigint	
local_blks_written	bigint	
temp_blks_read	bigint	
temp_blks_written	bigint	
blk_read_time	double precision	
blk_write_time	double precision	

For each database (`dbid`), we will see the query and the number of times it was called (`calls`). In addition to that, there is the total time used up by each query, and also the total number of rows returned (`rows`). In many cases, this information is enough:

```
SELECT query, total_time, calls
FROM   pg_stat_statements
ORDER BY 2 DESC;
```

This simple query will reveal the most expensive queries and their total runtime, but there is more. The next couple of columns will tell us about the I/O behavior of the query that is about to be inspected. It is possible to detect issues related to shared buffers as well as the behavior when it comes to local buffers allocated by the backend itself.

Then there are `temp_blks_read` and `temp_blks_written`. These two fields will tell us whether temporary data has to be written to disk or not.

Finally, there is information about I/O timing. Again, `track_io_timing` has to be turned on to gather this information. It is possible to check for each query how much time it has spent on input as well as output.

Notes about overhead

Many people ask about the overhead of this module. It has turned out that this is hard to measure. It is virtually zero. There is no need to be afraid of serious performance degradations when the module is used.

In general, it is useful to always enable `pg_stat_statements` to track slow queries. This gives you a valuable insight into what is happening on your server.

Resetting data

To reset the data collected by `pg_stat_statements`, you can call `pg_stat_statements_reset()`:

```
SELECT pg_stat_statements_reset();
```

PostgreSQL will reset statistics related to `pg_stat_statements`. The rest of the statistics will be unchanged.

Inspecting internal information

Once the basics (locking, slow queries, and so on) are done, our attention can be turned to more sophisticated inspections.

Looking inside a table

As already described in this book, PostgreSQL uses a mechanism called **Multi-Version Concurrency Control (MVCC)**. The beauty of this system is that more than one version of a row can exist at the same time, which is the perfect way to scale up the system. However, this also means that space on the disk might be wasted if too many versions of too many rows exist. This problem is generally called table bloat. If tables grow too large and inefficient, performance problems are likely to arise.

The `pgstattuple` module has been created to detect those problems. Before the module can be used, it has to be loaded:

```
test=# CREATE EXTENSION pgstattuple;
CREATE EXTENSION
```

Basically, `pgstattuple` is easy to use. All you have to do is call the `pgstattuple` function and pass a table name as a parameter. No columns are returned:

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pgstattuple('pg_class');
```

```

-[ RECORD 1 ]-----+-----
table_len      | 65536
tuple_count    | 305
tuple_len      | 56032
tuple_percent  | 85.5
dead_tuple_count | 0
dead_tuple_len  | 0
dead_tuple_percent | 0
free_space     | 5508
free_percent   | 8.4

```

What we see here is that `pgstattuple` returns the total size of the table, the number of rows inside the table, as well as their size. Then come two blocks; the `dead_*` columns provide us with information about the number and size of dead rows in the table. Dead rows can easily be turned into free space by running `VACUUM`.

The main challenge when using `pgstattuple` is to run it for many tables at the same time. It is always possible to run things for each table at a time, but this might be too time consuming. Running it for all tables at once is definitely more convenient.

To achieve that, we can turn to composite types. The `pgstattuple` is actually overloaded and not only can it be called with the name of a table, but it also works with object IDs. The main challenge now is to fetch the list of tables. A system table called `pg_class` can be used here. All tables are identified as `r` (relations) in the `relkind` column. However, the most important part is the decomposition of the data type returned by `pgstattuple`. Simple use of parentheses and an asterisk will work here:

```

test=# SELECT relname, (pgstattuple(oid)).*
      FROM   pg_class
      WHERE  relkind = 'r'
      ORDER BY table_len DESC;
-[ RECORD 1 ]-----+-----
relname      | pg_proc
table_len    | 557056
tuple_count  | 2697
tuple_len    | 525319
tuple_percent | 94.3
dead_tuple_count | 7
dead_tuple_len  | 1553
dead_tuple_percent | 0.28
free_space    | 8540
free_percent  | 1.53
...

```

A long list for each table in the system will be displayed.

The `pgstattuple` module is the perfect tool for detecting table bloat. However, use it with care. It has to read tables entirely. Therefore, using it too often is not a good idea because it causes heavy I/O (similar to a backup).

Inspecting the I/O cache

In addition to looking inside a table, it is also possible to inspect the PostgreSQL I/O cache (shared buffers area) to see what is cached and what is not. In many cases, knowing about the content of the I/O cache can reveal problems related to poor performance caused by too much disk activity.

The `pg_buffercache` view can be activated to inspect the cache:

```
test=# CREATE EXTENSION pg_buffercache;
CREATE EXTENSION
```

Just as was done before, the module has to be activated in order to provide us with information. Again, a system view is provided. It contains all of the information needed:

```
test=# \d pg_buffercache
          View "public.pg_buffercache"
   Column      |  Type   | Modifiers
-----+-----+-----
bufferid       | integer |
reelfilenode    | oid     |
reltablespace  | oid     |
reldatabase    | oid     |
relforknumber  | smallint|
relblocknumber | bigint  |
isdirty        | boolean |
usagecount     | smallint|
```

The main challenge with this special view is that it is a bit hard to read. Most fields are represented in a way that makes it pretty hard for normal users to comprehend what is actually shown. The following list outlines how those fields can be translated into something more readable:

- `reelfilenode`: The name of the data file on disk (`SELECT relname FROM pg_class WHERE reelfilenode = 'this oid';`)
- `reltablespace`: The object ID of the underlying tablespace (`SELECT spcname FROM pg_tablespace WHERE oid = 'this oid';`)
- `reldatabase`: The object ID of the database (`SELECT datname FROM pg_database WHERE oid = 'this oid';`)

With these `SELECT` statements, it will be possible to turn those numbers into readable names.

The next column is `relforknumber` (it is a cached block for the data file, Visibility Map, or Free Space Map), and finally, there is the number of the block in the cached table. This field allows you to see which areas of the table are in RAM and which are not.

The last two columns tell us whether the block is dirty and how often it is pinned. Given all of these fields, you can yourself create an analysis returning exactly the data that you need.

There are many more useful modules floating around on the web. However, the modules in this chapter are, according to my judgment, the most important modules. They are also shipped along with the PostgreSQL core distribution.

Integrating with external tools

When thinking about monitoring in general, most people instantly think of some monitoring tool such as Nagios, Zabbix, or the like. While monitoring tools are an important thing, I hope this chapter has proven that there is more to monitoring than just monitoring tools and pretty graphs.

Still, in this section, the goal is to guide you through some modules commonly used for monitoring with the help of tools. This is not meant to be an introduction to Nagios and other tools but a guideline to what can be used along with which tool.

Using Nagios plugins

The most widely used plugin that can be used in combination with Nagios is the plugin provided by the developers of the Burcardo project at:

http://bucardo.org/wiki/Check_postgres

It is also possible to use those plugins with Shinken and Tanto.

It offers a variety of checks suitable for monitoring different types of instances. The beauty of this plugin is that it can produce not only monitoring data for Nagios but also the output needed for **MRTG (The Multi Router Traffic Grapher)**: <http://oss.oetiker.ch/mrtg/doc/mrtg.en.html>.) as well as Cacti (<http://www.cacti.net/>). Cacti is a nice and simple graphical tool people can use to display what is going on in the system. If you are not willing to dive into Nagios, Cacti is a nice alternative, providing you with what you need in a simple way.

Which checks are provided by the Nagios plugins depends a little on the type of server you are running. Here are some of my favorite checks, which can be applied to pretty much any type of setup:

- `backends`: This tells you about the number of open database connections
- `commitratio`: This checks the commit ratio of all databases and complains when they are too low
- `custom_query`: This is the most powerful check because it allows you to run a generic query capable of returning any output needed
- `database_size`: This returns the database size
- `dbstats`: This returns high-level information about database activity
- `locks`: This checks the total number of locks on one or more databases
- `query_time`: This checks the length of running queries on one or more databases
- `wal_files`: This checks how many WAL files exist in the `pg_xlog` directory

Of course, depending on your application, different or additional checks might be necessary. However, this selection is a good foundation to get started with.

Alternative tools

If you prefer other tools over Nagios or Cacti, there are also proper plugins around. Depending on which tools you like, you might even have to come up with your own plugins to do your setup work as intended.

Zabbix plugins

The Zabbix plugin is called `pg_monz`. More information about this plugin can be found at http://pg-monz.github.io/pg_monz/index-en.html. It basically provides everything you need to integrate Zabbix with PostgreSQL and fetch all of the important information.

pganalyze-collector

The `pganalyze-collector` is a command-line tool that collects information about Postgres databases as well as queries executed on them. All of the data is converted to **JSON (JavaScript Object Notation)** and can be used by a tool of your choice for processing.

The module creates information about CPU, memory, RAM, tables, schemas, databases, queries, and so on.

This module can be downloaded from:

<https://github.com/pganalyze/pganalyze-collector>.

pg_view – a simple alternative

The `pg_view` is another tool that can be used to monitor PostgreSQL. The `pg_view` was developed by a major German retail company called Zalando, and it can be used free of charge. It is essentially a command-line tool, showing nicely condensed information on a simple command-line-based GUI.

More information on this tool can be found at https://github.com/zalando/pg_view.

Summary

In this chapter, monitoring was introduced. Some onboard tools, such as `pgstattuple`, `pg_buffercache`, and `pg_stat_activity` were presented. In addition to that, some external tools and plugins that can be used along with PostgreSQL were covered.

The next chapter is all about backups and replication. Major pitfalls in this area will be covered.

8

Fixing Backups and Replication

In this chapter, you will be guided through replication and learn about simple, basic backups. The following topics will be covered:

- Using `pg_dump` for text dumps
- Creating custom format dumps
- Performing binary base backups
- Understanding point-in-time recovery
- Setting up asynchronous replication
- Upgrading to synchronous replication

This chapter provides a practical, straightforward introduction to backups and replication. The focus is on simple guidelines.

Using `pg_dump`

The `pg_dump` command is one of the most important commands in PostgreSQL. It can be used to create textual as well as binary backups. Every system administrator will get in touch with this command once in a while. Here is how it works.

Creating textual dumps

Creating a textual dump is how most administrators digging into PostgreSQL get started. This is an essential, yet easy-to-manage, task.

The idea behind `pg_dump` is simple; it connects to the database that should be saved and transforms the content to a text format. Here is an example of its use:

```
pg_dump test > /backup/dump.sql
```

The simplest form of a backup is to send the SQL output created by `pg_dump` directly to a plain text file.

It is important to mention that a dump is always consistent. Internally, the dump is a large transaction in isolation level **repeatable read**. A dump represents a snapshot of data, which means that if you start a dump and it takes an hour to complete, then the changes in that one hour will not be included in the dump. This is because it takes a snapshot of data when you start and takes the backup from that.

In the next example, the dump is restored in a database called `new_test`:

```
psql new_test < /backup/dump.sql
```



Keep in mind that all you are doing is replaying text.

In general, a dump is a non-blocking operation. However, DDLs might conflict with some reads. In general, it is a good idea to be careful with DDLs and dumps, as they might exclude each other.

Taking care of blobs

Blobs are not automatically included in a dump. In some cases, this leads to trouble. Therefore, it is recommended to always use the `-b` (`--blobs`) option. If `-b` is activated, `pg_dump` will include those blobs in textual formats and ensure that your dump is complete in every aspect.

Handling passwords

In the following example, we have assumed that a connection has been set to trust. No passwords have been used so far. However, this is far from reality. In a real setup, there is always some sort of security mechanism.

In many cases, a file called `.pgpass`, located in the user's home directory, can be used. Alternatively, the `.pgpass` file can also be referenced by setting the `PGPASSFILE` environment variable.

This file should contain lines of the following format:

```
hostname:port:database:username:password
```

Just create the desired entries. No password prompt will be shown anymore.

Why doesn't PostgreSQL just provide some sort of `--put-password-here` parameter? The reason is that any password passed to a program, such as `pg_dump` or `psql`, will automatically show up in the process table. This has to be avoided at any cost.

Creating custom format dumps

In many cases, a plain textual dump is not what you would really want to use. The main problems with textual dumps are:

- They tend to be fairly large
- It is pretty hard to extract just a subset of data

A custom format dump is exactly what comes to the rescue here. It is basically a compressed dump that includes a **table of contents (TOC)**. The beauty of a custom format dump is that it is easily possible to extract just a subset of tables, a single index, or maybe a single procedure. In addition to that, it allows the replay process to be scaled out to more than one CPU.

Here is how a custom format dump can be created:

```
$ pg_dump test -Fc > /tmp/dump.fc
```

The `-Fc` option is the only option that has to be added to create a custom format dump.

Once the dump has been created, it is possible to take a look at the content of the dump file:

```
$ pg_restore --list /tmp/dump.fc
*snip*
175; 1259 16447 TABLE public a hs
176; 1259 16450 TABLE public b hs
174; 1259 16445 SEQUENCE public seq_a hs
172; 1259 16436 TABLE public t_test hs
173; 1259 16442 TABLE public x hs
*snip*
```

The `--list` option will return the list of objects in the dump. It is now possible to selectively restore data, instead of replacing the entire database.

Here is how a single table can be extracted from the dump again:

```
pg_restore -t t_test /tmp/dump.fc
```

The SQL representation of the data will be displayed on screen. To load the data into a database once again, a simple pipe or `-d` can be used:

```
$ pg_restore -t t_test /tmp/dump.fc | psql xy
$ pg_restore -t t_test /tmp/dump.fc -d xy
```

Making use of many CPUs

So far, only one CPU has been used to replay the dump. This is true for the text dump as well as the custom format dump. However — especially in the case of large databases — it is hard to replay stuff within a reasonable amount of time. Building indexes is expensive and takes a long time during the replay process.

The `-j` option allows custom format dumps to be replayed using more than one CPU. Here is how it works:

```
$ pg_restore -j 4 /tmp/dump.fc -d xy
```

This example assumes that data will be replayed into a database called `xy`. In this case, PostgreSQL should try to use up to four CPU cores concurrently to perform the dump.



Note that many cores are not possible when there is just a single table. It will only work in the case of more than one table.

Managing point-in-time recovery

The `pg_dump` utility is more of a traditional form of creating a backup. It makes sense for a small amount of data, but it tends to have its limitations as soon as the amount of data grows beyond a certain limit. Don't get me wrong; `pg_dump` works perfectly even with terabytes of data. However, let's assume you've got a dump of a 10 TB beast! Does it really make sense to replay a 10 TB database from a dump? Just consider all the indexes that have to be built, and consider the insane amount of time it will take to do that. It definitely makes sense to use a different method. This method is called **point-in-time recovery (PITR)**, or simply xlog archiving. In this section, you will learn about PITR in detail.

How PITR works

The idea behind PITR is to take a snapshot of the data and archive the transaction log created by PostgreSQL from then on. In case of a crash, it is then possible to restore any given point in time after the initial backup (base backup) has finished. The beauty of this concept is that end users won't lose too much data in case of a crash.

In case the system crashes, administrators can return to their last base backup and replay as much data as necessary.

Preparing PostgreSQL for PITR

To configure PostgreSQL for point-in-time recovery, only three settings, which can be found in `postgresql.conf`, are important: `wal_level`, `archive_mode`, and `archive_command`. The `wal_level` setting tells the system to produce enough of a transaction log for recovery based on `xlog`. By default, PostgreSQL only produces enough `xlog` to repair itself in case of a crash. This is not enough.

The `archive_mode` setting tells the system to archive its transaction log in a safe place. The `archive_command` setting will tell the system how to do that.

For getting started with transaction log archiving, the first thing to do is to actually create an archive (ideally on a remote box). For the sake of simplicity, `xlog` will be archived to `/archive` in this example:

```
mkdir /archive
```

Then `postgresql.conf` can be changed:

```
wal_level = archive
archive_mode = on
archive_command = 'cp %p /archive/%f '
```

Now the database can be restarted to activate those settings.

Once those settings are active, the transaction log will gradually accumulate in `/archive` directly. Whenever a 16 MB segment has been filled up by the system, it will be sent to the archive using `archive_command`. As changes come in, our `/archive` directory might begin to look like this:

```
$ ls -l
total 131072
-rw----- 1 hs wheel 16777216 Dec  8 12:24 000000010000000000000001
-rw----- 1 hs wheel 16777216 Dec  8 12:26 000000010000000000000002
-rw----- 1 hs wheel 16777216 Dec  8 12:26 000000010000000000000003
-rw----- 1 hs wheel 16777216 Dec  8 12:26 000000010000000000000004
```

By now, the transaction log is already archived properly.

Taking base backups

Once those `xlog` files are sent to the archive, it is possible to take a simple base backup. There are basically two ways of doing this:

- `SELECT pg_start_backup / pg_stop_backup`: The traditional way
- `pg_basebackup`: The modern way

In this chapter, both methods will be described. In the case of PITR, the traditional way will be used.

Here is how it works. First of all a simple SQL command has to be called as the superuser:

```
test=# SELECT pg_start_backup('some label');
pg_start_backup
-----
0/7000028
(1 row)
```

If this takes a while for you, don't worry. Behind the scenes, the system will wait for a checkpoint to happen and then return a transaction log position. Actually, nothing fancy happens behind the scenes. All it does is to tell the system where to restart replaying `xlog`.

Once this function has completed, it is possible to copy the entire database instance to a safe place. Keep in mind that this can be done during production, and it is not necessary to shut down the system during this operation.

Also, don't worry too much that files might vanish or appear during the backup process. This is a natural thing. Keep in mind that no change in `xlog` ever happens that cannot be found.

Consider the following example:

```
mkdir /base
cd $PGDATA
cp -Rv * /base/
```

You can use a simple `copy`, `rsync`, `ftp`, or any other means. Just ensure that the entire directory has been backed up nicely.

Once this is done, the backup process can be stopped again:

```
test=# SELECT pg_stop_backup();
NOTICE:  pg_stop_backup complete, all required WAL
         segments have been archived
 pg_stop_backup
-----
 0/70000B8
(1 row)
```

Remember that all of those operations can happen while the system is still active. No downtime is needed.

After the backup, a `.backup` file will show up in the `/archive` directory. It contains some important information:

```
$ cat 00000001000000000000000007.00000028.backup
START WAL LOCATION: 0/7000028
    (file 00000001000000000000000007)
STOP WAL LOCATION: 0/70000B8
    (file 00000001000000000000000007)
CHECKPOINT LOCATION: 0/7000028
BACKUP METHOD: pg_start_backup
BACKUP FROM: master
START TIME: 2014-12-08 12:33:50 CET
LABEL: some label
STOP TIME: 2014-12-08 12:39:46 CET
```

The `.backup` file shows when the backup has been created. It will also tell us the oldest `xlog` file needed to perform the replay process. In this case, it is the `00000001000000000000000007` file. Everything older than this file can be safely deleted from the server.

Replaying xlog

If the system is actively used, more transaction log will accumulate over time in `/archive`:

```
-rw----- 1 hs wheel          294 Dec  8 12:39 000000010000000000000007.0
00000028.backup
-rw----- 1 hs wheel 16777216 Dec  8 12:45 000000010000000000000008
-rw----- 1 hs wheel 16777216 Dec  8 12:45 000000010000000000000009
-rw----- 1 hs wheel 16777216 Dec  8 12:45 00000001000000000000000A
```

Let's assume now that the server we want to protect has died and the base backup needs to be recovered using PITR.

To make this work, the first thing to do is to come up with a file called `recovery.conf`:

```
$ cat recovery.conf
restore_command = 'cp /archive/%f %p'
recovery_target_time = '2025-04-05 14:32'
```

Basically, just one command is needed here. The `restore_command` function will copy the desired files from the archive to the location needed by PostgreSQL (%p). PostgreSQL will call `restore_command` for one file after the other until it runs out of xlog or until the desired `recovery_target_time` is reached. Actually, the `recovery_target_time` command is redundant. If it is not there, PostgreSQL will recover as far as possible, and not stop at the desired point in time.

Once the `recovery.conf` file is in place, administrators have to ensure that the directory containing the base backup is set to 700:

```
chmod 700 /base
```

Then the server can be fired up (in this example, the log is configured to show up on the screen):

```
$ pg_ctl -D /tmp/base/ start
server starting
$ LOG:  database system was interrupted; last known up
       at 2014-12-08 12:33:50 CET
LOG:  starting point-in-time recovery to
       2025-04-05 14:32:00+02
```

The server mentions that it was interrupted and will tell us the point in time the system is intended to roll forward to.

Then one file after the other is copied from the archive and replayed:

```
LOG:  restored log file "00000001000000000000000007" from archive
LOG:  redo starts at 0/7000090
LOG:  consistent recovery state reached at 0/70000B8
LOG:  restored log file "00000001000000000000000008" from archive
LOG:  restored log file "00000001000000000000000009" from archive
LOG:  restored log file "0000000100000000000000000A" from archive
LOG:  restored log file "0000000100000000000000000B" from archive
```

At some point, PostgreSQL will throw an error because it cannot copy the next file. This makes sense because this chapter has been written in late 2014, and therefore, 2025 cannot be reached easily with the existing `xlog`. But don't worry! PostgreSQL will figure out that there is no more `xlog` and just go live. If a date between the base backup and the last `xlog` file is selected, PostgreSQL will go live without issuing any error:

```
cp: /tmp/archive/00000001000000000000000000C: No such file or directory
LOG:  redo done at 0/BD3C218
LOG:  last completed transaction was at log time 2014-12-08
12:46:11.755618+01
LOG:  restored log file "00000001000000000000000000B" from archive
cp: /tmp/archive/00000002.history: No such file or directory
LOG:  selected new timeline ID: 2
cp: /tmp/archive/00000001.history: No such file or directory
LOG:  archive recovery complete
LOG:  autovacuum launcher started
LOG:  database system is ready to accept connections
```

The recovery process has finished nicely, and the database instance is ready for action. Note that at the end of the process, `recovery.conf` is renamed to `recovery.done`.

Making use of asynchronous replication

While point-in-time recovery is nice, it might not be enough. In many cases, people want an up-to-date standby server that can also serve as a read-only slave. A feature called streaming replication is exactly what is needed in this case.

Setting up streaming replication is easy and can be performed in a handful of steps:

1. Adapt `postgresql.conf` on the master server.
2. Change `pg_hba.conf` to allow remote base backups.
3. Run `pg_basebackup` to get an initial copy.
4. Fire up the slave.

To set up `postgresql.conf`, the following settings are needed:

```
wal_level = hot_standby
max_wal_senders = 5
hot_standby = on
```


The first two settings are mandatory on the master. The `wal_level` field tells the server to create enough `xlog` to allow streaming replication, and `max_wal_senders` will indicate the number of streams the master is allowed to keep open.



If you are planning to run one slave, consider setting `max_wal_senders` to 3 or higher. During streaming, only one wal-sender process will be used, but during the initial copy, up to two connections are required. Setting this too tight can cause unnecessary database restarts.

The third setting changed here is `hot_standby`. Basically, this is a slave setting, and the master will ignore it. The reason it has been set is that in the next step (base backup), the entire instance, including the configuration, will be sent to the standby servers. If this variable has already been set on the master, there is no need to change it later on the slave (which is somewhat convenient).

Once `postgresql.conf` has been changed, it is time to attack `pg_hba.conf`. The important point here is that you have to precisely tell PostgreSQL who is allowed to stream the transaction log and who is not. IPs allowed must be listed in `pg_hba.conf`, like this:

```
host    replication    all    192.168.0.34/32      trust
```

In this case, a slave running on `192.168.0.34` is allowed to fetch the transaction log from the master without having to send a password. For the sake of our example, this is fine.

If you want to test replication on the same box as the master, the following configuration may be useful to you:

```
local    replication    all                                trust
host     replication    all    127.0.0.1/32        trust
host     replication    all    ::1/128             trust
```

Once `postgresql.conf` and `pg_hba.conf` have been changed, the server can be restarted.

Working with `pg_basebackup`

In the next step, it is possible to fetch an initial copy of the database instance. Remember that in the previous section, this very step was performed using `pg_start_backup` and `pg_stop_backup`. In this section, the second method, `pg_basebackup`, will be used.

To prepare the slave for replication, these steps will be necessary:

```
mkdir /slave
chown postgres.postgres slave
chmod 700 slave
```

This example assumes that the server will be running as postgres and that the target directory is /slave.

To create the initial backup, the following command can be executed on the slave:

```
pg_basebackup -D /slave \
-h master.server.com --checkpoint=fast \
--xlog-method=stream -R
```

Ensure that those host names are replaced by your server names or IP.

Here are the meanings of those command-line flags:

- -D: This tells pg_basebackup where to put the initial backup.
- -h: This tells the system where to look for the master.
- --checkpoint=fast: This indicates that the base backup will initiate a checkpoint on the master. This makes sense; otherwise, small backups will take longer than necessary.
- --xlog-method=stream: This will force pg_basebackup to open a second connection and stream enough xlog while the backup is running. The advantage here is that the newly created base backup is already self-sufficient. It can be started straight away. It also ensures that a fresh backup can be started without falling back to using an archive.
- -R: This is a fairly new feature and will create a reasonable `recovery.conf` file for streaming replication.

The default configuration file created by -R will contain two lines:

```
standby_mode = 'on'
primary_conninfo = '...'
```

The `standby_mode = on` setting means that the server should keep streaming even if no xlog is around anymore (it should just wait for more).

The `primary_conninfo` setting points to the master and contains the connection string to the desired master server. These two settings are enough to make streaming work.

Firing up replication

Once the base backup has been created, the server can be started. If, for testing purposes, your master and your slave are in the same box, you can start PostgreSQL on a different port without having to modify the configuration. It works as shown in the next listing:

```
$ pg_ctl -D /slave/ -o "--port=5433" start
```

If your slave has been configured properly, some information will show up in the log:

```
server starting
LOG:  database system was interrupted; last known up at
      2014-12-08 15:07:39 CET
LOG:  creating missing WAL directory
      "pg_xlog/archive_status"
LOG:  entering standby mode
```

To the server, the entire endeavor looks like a crash. It tells us when it feels that it stopped working (that is, at the time of the base backup).

Once PostgreSQL enters standby mode, you are on the right path. PostgreSQL will start replaying stuff, and eventually reach a consistent state:

```
LOG:  redo starts at 0/D000028
LOG:  consistent recovery state reached at 0/D0000F0
```

Always check whether a consistent state has been reached or not. If you have no consistent state, it means there is trouble ahead.

```
LOG:  database system is ready to accept read only
      connections
LOG:  started streaming WAL from primary at 0/E000000
      on timeline 1
```

Also, make sure that you have entered streaming mode.

Voilà! The server is up and running now. To check whether replication is working safely, you can rely on a system view called `pg_stat_replication`. It will contain one line per slave:

```
test=# \d pg_stat_replication
      View "pg_catalog.pg_stat_replication"
      Column      |      Type      | Modifiers
```

-----+-----+-----		
pid	integer	
usesysid	oid	
username	name	
application_name	text	
client_addr	inet	
client_hostname	text	
client_port	integer	
backend_start	timestamp with time zone	
backend_xmin	xid	
state	text	
sent_location	pg_lsn	
write_location	pg_lsn	
flush_location	pg_lsn	
replay_location	pg_lsn	
sync_priority	integer	
sync_state	text	

These fields are highly important, and it is worth explaining them in more detail.

The first field contains the process ID of the process serving the slave on the master. Then come the `usesysid` field and the name of the user connected to the system. This is the user the client is connected as. Then there is an `application_name` field. The purpose of the `application_name` field is to give a connection a real name. This is important for debugging (for normal applications) and, as you will see in the next section, for synchronous replication.

Then there are a couple of fields showing where a connection comes from (`client_*`). Those fields will tell you which slave the entire line is about. The `backend_start` field tells us when the slave started to stream.

After the `backend_xmin` field comes the state the connection is in. If everything is normal, it should be set to streaming. Then there are four location fields. They will tell the administrator how far the slave has already consumed respectively replayed `xlog`.

Finally there is the `sync_state` field. It tells us whether the server is in synchronous or asynchronous replication mode.

For a short, step-by-step guide about streaming replication, refer to http://www.cybertec.at/wp-content/uploads/PostgreSQL_5_min_streaming_replication.pdf.

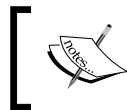
Promoting slaves to masters

Promoting a slave to a master is an easy task. The following command will do it:

```
pg_ctl -D /slave promote
```

The slave will receive the signal and turn itself into a master. The cool thing here is that even existing read-only connections will automatically be turned into read/write connections. There is not even any need to reconnect; PostgreSQL will do all of this for you.

Now the new master is totally independent of the old master. Those two boxes won't synchronize anymore.



[Note that a master cannot be transformed back into a slave easily. A complete resync should be performed to turn the broken, old master into a new slave replica.]

Making replication safer

So far a simple replication setup has been shown. But what happens when a user takes a base backup and does not connect the slave to the master instantly? Nothing prevents the master from recycling its replication log. If the slave comes late, the transaction log needed is not there anymore. The same can happen in the case of slow connections. A user might load data into the master at 200 MB per second but the network can only handle 100 MB per second. The slave falls behind, and again, nothing prevents the master from recycling its `xlog`. To solve this kind of problem, a simple mechanism is there — `wal_keep_segments`.

If `wal_keep_segments` is set to, say, 100 in `postgresql.conf`, it means that the master will keep $100 * 16 \text{ MB}$ more of transaction log around than it would need to repair itself in case of a crash. This little buffer gives the slave some room to breathe, and it defines the maximum amount of data by which the slave is allowed to fall behind the master without losing sight of the master. My personal recommendation is to always set `wal_keep_segments` to a reasonable value to ensure that the slave can always keep up with the master in the event of excessively slow connections, reboots on the slave side, and so on.

One more issue that can be observed in many cases is the appearance of conflicts. In PostgreSQL, the slave can be used to serve read-only transactions. It can happen from time to time that the slave does something that conflicts with the intentions of the master. As an example, let's assume that the slave is running a long `SELECT` statement. In the meantime, somebody on the master performs a `DELETE` statement and instantly fires a `VACUUM` command to clean up those rows. The `SELECT` statement on the slave might still need this data for its purposes. However, the `xlog` tells the system to clean up those rows. Thus, a conflict has happened. The slave will now wait for `max_standby_streaming_delay = 30s` before it simply kills the connection on the slave, if the transaction in doubt does not finish earlier.

To prevent most conflicts, it is possible to set `hot_standby_feedback = on`. This will make the slave send its oldest transaction ID (`xmin`) to the master. The master will then know that there is still a query running on the slave observing certain data. Therefore, `VACUUM` will be delayed until the conflict cannot happen anymore.

Switching to synchronous replication

While asynchronous replication is sufficient in 90 percent of cases, many people ask for synchronous replication. The idea behind synchronous replication is that a transaction is only valid if it has been accepted by at least two servers. Sounds easy? It is! To configure PostgreSQL for synchronous replication, only two settings are necessary.

The first thing to configure is the slave side. All that has to be done is to add an `application_name` field to `primary_conninfo` in `recovery.conf`.

A configuration setting might look like this:

```
primary_conninfo = 'host=master.server.com user=postgres
port=5432 application_name=some_name'
```

The slave will now register itself on the master as `some_name`. The master will now check its `synchronous_standby_names` field. The first entry matching in `synchronous_standby_names` will be considered synchronous; the rest will be considered asynchronous.

So, in short, here's what you have to do:

1. Add an `application_name` field to the `primary_conninfo` in `recovery.conf` on the slave.
2. Add this very name to `synchronous_standby_names` on the master.

However, synchronous replication comes with two issues. The first issue is obvious; performance of short transactions will drop because network latency can be a problem.

The second issue is related to availability. The golden rule is that synchronous replication should never be done with only two servers. Remember that PostgreSQL ensures that a transaction is considered to be safe only if it has been approved by at least two servers. If you got two servers and one is lost, PostgreSQL cannot live up to its promise. The result is that PostgreSQL will not accept writes anymore until a second server is provided, which can take the data and make the system happy!

It is highly recommended to use at least three servers that are redundantly connected to each other.

Handling timelines

If you've got only two servers, timelines are really not an issue. However, if a setup grows larger, understanding timelines can be vital to success. What is this all about? When a server is promoted from a slave to a master, it increments its timeline. This is done to ensure that in a large setup, the right `xlog` is consumed by each server.

In the case of two servers, this is not an issue. If everything is fine, both servers will start in `timeline 1`. In case the master dies, the remaining server will switch to `timeline 2`, and everything will be fine.

But what about a setup consisting of three servers? In case the master crashes, one of those remaining servers will be the next master. But who should be the next master? That is for you to decide. You cannot just take a random server. It is important to check which server has received how much transaction log, and decide on the most advanced server. This is how you can figure it out:

```
SELECT pg_last_xlog_replay_location();
```

But don't promote the most advanced slave instantly. That will make the server jump to the next timeline. What about the remaining boxes? They will sit idle in `timeline 1`, and the replication will be broken.

So, before a server is promoted, reconnect all slaves to the new desired master (by changing `primary_conninfo` and signaling the server), and promote the new master *after* that. It will ensure that the new master will jump to `timeline 2` and replicate this change to all slaves so that those slaves can easily jump to `timeline 2` along with the master.

Summary

In this chapter, PITR and streaming replication were covered in brief. All the basic concepts related to those techniques were outlined, and a practical introduction to those topics was presented.

In the next chapter, you will see problems related to hardware disasters and hardware failure.

9

Handling Hardware and Software Disasters

In this chapter, you will be introduced to issues related to hardware and software failure. Even expensive hardware is far from perfect, and it may fail from time to time. Troubles could occur in the form of failing memory, broken hard drives, or damaged filesystems, kernel-related issues and so on. In this chapter, some of the most common troubles will be covered.

Here are some of the important topics we will cover:

- Checksums – Preventing silent corruption
- Zero out damaged pages
- Dealing with index corruption
- Dumping single pages
- Resetting the transaction log
- Power out related issues

Checksums – preventing silent corruption

When reading books about high availability and fault tolerance, in general, I always get the impression that most systems work on the assumption that crashes are atomic. What do I mean by that? Let's imagine there are two servers; one server has crashed and the other takes over. Unfortunately, crashes are not like this in many cases. In the real world, crashes are often far from atomic. Trouble is likely to build up gradually until things go really wrong. Consider memory corruption; when the memory goes wrong, it might not lead to an instant crash, and even when it does, the system might restart again without problems before troubles return. In many cases, a problem that accumulates silently is way more dangerous than a simple crash. The main danger in the case of silent corruption is that problems could gradually spread in the system without anybody noticing.

To prevent problems related to silent corruption caused by memory consumption, PostgreSQL provides block-level checksums. By default, data is written as it is – one block at a time. However, with block checksums enabled, PostgreSQL will check each 8k block on I/O to see whether it is still fine or not.

To enable data checksums, it is possible to run `initdb` using the `-k` option. This will set the configuration variable for the entire instance.

To check whether a database instance has data checksums enabled, you can run the following command (assuming that the database instance is living in `/data`):

```
$ pg_controldata /data | grep -i checks
```

The command will return 0 or 1, depending on whether the setting has been turned on or off.

At this point, there is no possibility to turn checksums on after `initdb`. There is no way to change the configuration in production.

Zeroing out damaged pages

Once in a while, things do go wrong even if all precautions have been taken. A filesystem might lose some blocks here and there or a disk might simply lose a couple of sectors. The following might happen on the PostgreSQL side in this case:

```
test=# SELECT count(*) FROM t_test;
ERROR:  invalid page in block 535 of relation
        base/16384/16436
```

If a block (or a couple of blocks) has fallen victim to a problem in the filesystem, PostgreSQL will error out and tell the end user that the query cannot be completed anymore.

In the scenario outlined here, you can be certain of one thing: some data has been lost in the storage system. It is important to point out that loss of data is virtually never caused by PostgreSQL itself. In most cases, we are talking about broken hardware, broken filesystems, or some other memory-related problem that has spread its blessings to your data files.

If storage-related problems arise, the general rule is, "don't touch stuff; back up stuff!" Before anything is done that might make the problem even worse, try to create a filesystem snapshot, a tar archive, or any kind of backup. In many cases, the good old "dd" can do a good job, and it can definitely be worthwhile to attempt to create a clone of the underlying device on your system.

Once everything has been rescued on the system, the next step can be performed. PostgreSQL provides a feature that allows you to zero out blocks that are not faulty. In the preceding example, the query could not have been completed because the file contained some bad data. By setting `zero_damaged_pages` to on, it is possible to tell PostgreSQL to zero out all broken blocks. Of course, this does not rescue lost data, but it at least helps us retrieve something from the system:

```
test=# SET zero_damaged_pages TO on;
SET
```

Once the value has been set, the query can be executed again:

```
test=# SELECT count(*) FROM t_test;
WARNING:  invalid page in block 535 of relation
          base/16384/16436; zeroing out page
count
-----
      42342
(1 row)
```

This time, warnings will pop up for each broken block. Luckily, just one block is broken in this case. It is quickly replaced by some binary zeroes, and the query moves on to the next (proper) blocks.

Of course, `zero_damaged_pages` is dangerous because it eliminates bad data. So what choice do you have? When something is so broken, it is important to rescue at least what is there.

Dealing with index corruption

Once in a while, an index may blow up. In most cases, broken indexes are caused by hardware failures. Once again, the most suspicious thing in the case of index corruption is the RAM. So, always check your system memory in case an index has blown up.

In general, the same rules apply: take a snapshot first, and then simply recreate those broken indexes. `REINDEX` can help in this case:

```
test=# \h REINDEX
Command:      REINDEX
Description:  rebuild indexes
Syntax:
REINDEX { INDEX | TABLE | DATABASE | SYSTEM }
         name [ FORCE ]
```

Keep in mind that `REINDEX` needs a `SHARE` lock, which ensures that no writes can happen. In some setups, this could turn out to be a problem. To avoid locking, it makes sense to turn to `CREATE INDEX CONCURRENTLY`. A concurrent index build takes longer than a normal build. However, it avoids a nasty table lock that blocks writes. It can be very beneficial to many applications.

In general, an index-related problem is usually not as bad as a bad block inside a table. Remember, however, that the index itself does not contain data and data cannot be recreated easily.

Dumping individual pages

Advanced users who really want to inspect their broken systems in detail can focus their attention on `pageinspect`. It allows them to inspect a single data or index page on the disk. This requires a fair amount of knowledge about the inner working of PostgreSQL, but it can give valuable insights into what might have gone wrong in a system.

To make use of the module, first it has to be installed. It works like this:

```
test=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

Once the module is in place, a table can be inspected. To do so, a couple of functions are available. The most important function is `get_raw_page`. It will be needed for subsequent inspections:

```
get_raw_page(relname text, fork text, blkno int)
returns bytea
```

The `get_raw_page` function returns a `bytea` field containing the content of the page. The first parameter tells us the relation name we want to inspect. The second parameter needs the so-called relation fork. In PostgreSQL, a table does not only consist of data files. In all, there are three types of files belonging to a table: `main` (the table itself), `fsm` (information about free space in the so-called Free Space Map), and `vm` (the so-called Visibility Map). Depending on the branch of the table you want to inspect, this second parameter has to be set to the proper value. Finally, there is the last parameter. It tells PostgreSQL which block to dump.

If you want to extract the first page of `pg_class` (a system table), try the following:

```
test=# SELECT * FROM get_raw_page('pg_class',
    'main', 0);
```

Of course, the output of this function is too long to list. It is an 8k block after all!

Extracting the page header

Once the page has been extracted, the page header can be inspected. To do so, just call `page_header`:

```
test=# \x
Expanded display (expanded) is on.
test=# SELECT * FROM page_header(get_raw_page('pg_class', 'main', 0));
-[ RECORD 1 ]-----
lsn          | 0/E6604D8
checksum     | 0
flags        | 1
lower        | 340
upper        | 904
special      | 8192
pagesize     | 8192
version      | 4
prune_xid    | 0
```

What is the meaning of those fields? The following listing contains an overview:

- `pd_lsn`: This identifies `xlog` record for the last change to this page
- `pd_checksum`: This is the page checksum, if set
- `pd_flags`: This is a set of flag bits
- `pd_lower`: This is an offset to start of free space
- `pd_upper`: This is an offset to end of free space
- `pd_special`: This is an offset to start of a special space

- `pd_pagesize_version`: This gives the size in bytes and the page layout version number
- `pd_prune_xid`: This is the oldest XID among potentially prunable tuples on the page



Keep in mind that not all of this information is there for every type of object; for example `pg_special` does not give anything useful for a normal data file because there is no special space there.

If those numbers show something odd, it is a good indicator that something has gone wrong.

In the next step, individual tuples (rows) can be inspected. To do this, `heap_page_items` is available:

```
test=# SELECT *
      FROM heap_page_items(
          get_raw_page('pg_class', 'main', 0)
      );
-[ RECORD1 ]-----
lp          | 1
lp_off      | 7992
lp_flags    | 1
lp_len      | 197
t_xmin      | 931
t_xmax      | 0
t_field3    | 5
t_ctid      | (0,1)
t_infomask2 | 29
t_infomask  | 11019
t_hoff      | 32
t_bits      | 11111 ... 0000
t_oid       | 2619
```

In this example, a couple of rows are returned.

Resetting the transaction log

In this section, resetting the transaction log will be covered. Before we get started, I want to personally issue a warning: don't take this lightly. Resetting the transaction log is a harsh thing to do. It almost always leads to some data loss, and it does not guarantee that your data will still be fully consistent. Resetting `xlog` is the last thing to consider when things go south.

The same rules as we covered before apply here: always take a snapshot of the filesystem or shut down the database, and create a binary copy of the data directory before using `pg_resetxlog`. Let me stress my point. In my 15-year long career as a PostgreSQL consultant, I have had to do this only a handful of times. Usually, this can be resolved in some other way.

However, if PostgreSQL does not start up anymore because the `xlog` is broken, `pg_resetxlog` can come to your rescue. Here is how the syntax works:

```
$ pg_resetxlog --help
pg_resetxlog resets the PostgreSQL transaction log.
```

Usage:

```
pg_resetxlog [OPTION]... DATADIR
```

Options:

```
-e XID EPOCH      set next transaction ID epoch
-f                force update to be done
-l XLOGFILE       force minimum WAL starting location
                  for new transaction log
-m MXID,MXID      set next and oldest multitransaction
                  ID
-n                no update, just show what would
                  be done (for testing)
-o OID            set next OID
-O OFFSET         set next multitransaction offset
-V, --version     output version information, then exit
-x XID            set next transaction ID
-?, --help       show this help, then exit
```

There are a couple of settings here. The most important ones are `-o`, `-x`, `-e`, `-m`, `-O`, and `-l`. These settings allow you to go to a certain transaction ID, set an OID, and so on.

In some cases, it might happen that `pg_resetxlog` complains that the `pg_control` file is not valid. The control file is vital because it contains information about block sizes, checkpoints, toast sizes, and so on. To see which data is in the control, try the following snippet:

```
pg_controldata $PGDATA    # or replace $PGDATA with
                           # your PostgreSQL data
                           # directory
```


The `pg_controldata` file will write a lot of information to the screen. If `pg_resetxlog` cannot find a valid control file, it is necessary to use `-f`. Then it tries to fix things even if the control file is long gone or is corrupted.

Once `pg_resetxlog` has been executed, it is usually possible to start the database again (unless it is really doomed). Ensure that a backup is taken instantly and that data is checked for sanity. This includes verifying that all foreign keys are okay and that data can be dumped, or restored, nicely.

Power-out-related issues

Hardware is not always a problem. It might easily happen that you face something as simple as a power out. In general, PostgreSQL can easily survive power outs. The transaction log takes care of all failure-related issues, and with the help of `xlog`, it is always possible to bring the database back to a consistent state. It is safely possible to pull the plug and rely on the quality of PostgreSQL.

However, there are some nasty cases that can lead to even nastier troubles. On commit, PostgreSQL relies on the fact that data can be flushed to the disk. A system call, `fsync`, is used to ensure that everything relevant is forced to the disk. However, what happens if `fsync` cannot do its job? What if data cannot be (or data is not) flushed to the disk? If your system is up and running, everything is fine. The data will make it to the disk at some point, and life is good! However, what if a crash happens? The situation is grimmer. In short, anything can happen. If PostgreSQL does not know anymore what has been safely written to disk and what is not, any kind of corruption may occur.

Why would `fsync` fail? Things can always go south if hardware fails. However, there is a more common issue here – virtualization. Not all pieces of virtualization software will flush data to the disk by default. For performance reasons, some systems such as VirtualBox will default to optimizing `fsync` by just writing in a more lazy way. The results are likely to be disastrous, not only for PostgreSQL but also for the entire virtual machine.

The importance of `fsync` cannot be stressed enough. If flushing to the disk is somewhat broken, the entire database instance is at the mercy of your electricity company. Surely, not flushing to the disk is a lot faster, but it can also mean that your data will be lost much faster.

In case data is lost in a virtualized setup, always try to figure out whether `fsync` is really doing what it is supposed to do on the underlying hardware component.

Summary

In this chapter, the basic and most common problems were discussed. This includes hardware failure as well as problems related to broken indexes, broken `xlog`, and a lot more.

In the next chapter, everything you learned in this book will be summed up and presented in one extensive example.

10

A Standard Approach to Troubleshooting

In this chapter, the goal is to sum up some of the approaches you have seen in this book and combine them into one overall approach for typical everyday scenarios. Customers may come up with any or all of the following statements:

- "My database is slow"
- "CPU usage and I/O are going through the roof"
- "It has not always been so slow"
- "Sometimes, replication fails"
- "I'm adding new hardware"

The goal here is to provide you with a systematic approach to the problem and point to solutions. Note that the example outlined in this chapter is a typical case faced by clients around the globe.

Getting an overview of the problem

Before getting started, the most important thing is to get an overview of the problem. Assuming that there is no corruption around, the first thing to do is to take a detailed look at the content of `pg_stat_activity`. As already outlined in this book, `pg_stat_activity` contains a list of all open database connections. There are a couple of things to look out for here:

- Is the number of open database connections reasonable?
- Are there many idling transactions? Is there a certain query showing up again and again?
- Are there queries that are obviously running too long?

Many middleware components open an insane number of database connections. Remember that each useless connection takes away a bit of memory, which could have been used for something more productive, such as caching. However, the number of database connections alone does not usually impose a dramatic overhead. The number of active snapshots is way more dangerous because active snapshots directly translate into a decrease in speed. An active snapshot causes some internal overhead. The benchmark at http://www.cybertec.at/max_connections-performance-impacts/ will prove this point.

Once the number of connections has been checked, it makes sense to see how many of those are idling. Open transactions that have been idling for quite some time can be a danger. Why? The `VACUUM` command can only clean up dead rows if there is no transaction around anymore that is capable of seeing the data. Old, idle transactions can delay the cleanup of rows, and bad performance is likely to be the consequence. Even if the `VACUUM` command is executed on a regular basis, things are likely to stay slow because `VACUUM` is simply not able to do its job. Therefore, it can make sense to keep an eye on unreasonably long transactions.

But there is more; if the same query shows up again and again in `pg_stat_activity`, it is already worth taking a look at this query and ensuring that it performs nicely. The same applies to queries that are already showing high execution times and show up in the listing.

Attacking low performance

After this basic inspection, it is time to get rid of a very common problem: wrong and missing indexes.

Reviewing indexes

At this point, most of you may say, "indexing? Come on! What is the point? We know that; let's work on real stuff!" From experience, I can tell you with absolute certainty that broken indexing is the single most common cause of bad performance in the world. So, before focusing on anything else in the system, it always makes sense to carry out a safety check to ensure that indexing is definitely okay.

What is the best way to check for missing indexes? In my daily life as a consultant, I use this query:

```
test=# SELECT relname, seq_scan, seq_tup_read,
        idx_scan AS idx,
        seq_tup_read / seq_scan AS ratio
FROM   pg_stat_user_tables
```

```

WHERE seq_scan > 0
ORDER BY seq_tup_read DESC
LIMIT 10;
relname | seq_scan | seq_tup_read | idx | ratio
-----+-----+-----+---+-----
t_user  | 4564324 | 18563909843245 | 2 | 4067176
...
(10 rows)

```

The output of the query shows the following fields: the name of the table, the number of sequential scans that have occurred on the table, the number of rows found in those sequential scans, the number of index scans used, and the average number of rows read during a sequential scan.

Given the listing here, the next question is, why would anybody want to read 4 million rows 4.5 million times? There is no sane case that justifies that such a large table should be read millions of times. Just imagine what it means for the overall load of your system if a table containing 4 million rows is read over and over again. It is like reading the entire phone book to find a single number.

In most cases, the missing index is already obvious when looking at the table structure. In this example, the user table might be used for authentication, but the e-mail address used by people to sign in might not be indexed.



Keep in mind that sequential scans are not always bad and they can never be avoided completely. It is just important to avoid a large number of expensive and pointless sequential scans.

Experience has shown that in most systems showing bad performance, indexes are missing. It is very likely that a simple review of your indexing strategy will solve most of your problems.

On the other hand, it is also possible that too many indexes cause performance problems. An index is not always a solution to a problem—it can also do harm. Such performance problems are not as obvious as missing indexes. In addition to that, the use of too many indexes can usually be tolerated much longer than missing indexes (depending on the type of application, of course).

Fixing UPDATE commands

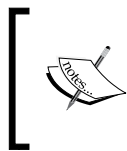
Once indexes have been fixed, a second and very common problem can be attacked. Many people forget about a thing called `FILLFACTOR`. Missing `FILLFACTOR` settings can cause performance problems in the case of update-intensive workloads.

How can those problems be detected? Here is a query showing how it works:

```
SELECT schemaname, relname, n_tup_ins,  
       n_tup_upd, n_tup_hot_upd, n_tup_del  
FROM   pg_stat_user_tables  
ORDER BY n_tup_upd DESC;
```

This query returns those tables that are facing most of the `UPDATE` commands in the system. Now the important ratio is the ratio of `n_tup_upd` to `n_tup_hot_upd`. Let's assume a row is updated. If the page containing the row is full, the row has to be copied to some other block. Additional, expensive I/O is the logical consequence. If a proper `FILLFACTOR` setting (for example, `FILLFACTOR = 70` for 70 percent) has been chosen, a so-called `HOT-UPDATE` command will be performed. In short, the copy of the row can be put in the same block as the original row. A `HOT-UPDATE` command is way cheaper than a traditional, normal `UPDATE` command. Therefore, tables facing many `UPDATE` commands need a reasonably low `FILLFACTOR` option to speed up things and to reduce I/O.

The query shown in this section helps identify those tables that need special attention. It makes sense to work through the list top down and set `FILLFACTOR` on indexes and tables.



Keep in mind that the perfect `FILLFACTOR` option can never be calculated. However, you can make an educated guess and come up with a reasonably good setting. If the `FILLFACTOR` has to be lowered, 75 percent can be a good start.

To change the `FILLFACTOR` to 70 percent, the following command can be used:

```
ALTER TABLE foo SET (FILLFACTOR=70);
```

If you want to change the `FILLFACTOR` of an index, use this:

```
ALTER INDEX foo_index SET (FILLFACTOR=70);
```



Keep in mind that this setting does not have an immediate effect on the physical layout of the table; it is merely a hint. However, the system will try to reach the desired state over time.

Once indexes and `FILLFACTORS` have been defined, the majority of problems should have been fixed.

Detecting slow queries

Next, an important step can be performed—finding slow queries. Some people prefer to check for slow queries first. However, I personally prefer to do some basic checks such as the ones shown up to here because those checks don't require restarts. Activating `pg_stat_statements` needs a restart, and therefore, it makes sense to activate it after picking the low-hanging fruit.

When activating `pg_stat_statements` in `postgresql.conf`, as outlined earlier in the book, it makes a lot of sense to set `track_io_timing` to `true`. Enabling `track_io_timing` will provide users with detailed information about which query has performed which kind of I/O.

To figure out what is going on in the system, the following queries may be useful.

Find the most time-consuming queries, like this:

```
SELECT (SELECT datname
        FROM   pg_database
        WHERE  dbid = oid),
       query, calls, total_time
FROM   pg_stat_statements AS x
ORDER BY total_time DESC;
```

The most time-consuming queries will show up at the top of the listing and can be fixed one at a time. However, in some cases, total execution time is not your problem. What if I/O is the main limiting factor? In this case, the following fields need special attention:

<code>temp_blks_read</code>		<code>bigint</code>	
<code>temp_blks_written</code>		<code>bigint</code>	
<code>blk_read_time</code>		<code>double precision</code>	
<code>blk_write_time</code>		<code>double precision</code>	

If the `work_mem` settings are too low, it can happen that many temporary files are written to disk, resulting in expensive disk writes, which can easily be avoided. Those `temp_*` are a good indicator of the query that causes disk wait.

The `blk_*` settings will give insight into how much I/O is needed by a query and how much time is spent in I/O.

Working through the most expensive queries is essential to track down those last bottlenecks remaining on your systems. If `postgresql.conf` is configured in the same way, going through the basic steps outlined in this chapter will remove a great portion of the most common performance problems.

The problems remaining usually need a more detailed look and maybe changes inside the application. Everything covered so far can be fixed using PostgreSQL's onboard means.

Fixing common replication issues

Given the goals set for this chapter, the last topic to be done is the customer's complaint about replication, which tends to break from time to time. In PostgreSQL, replication never breaks for technical reasons; only misconfiguration can lead to unexpected behavior. Here are the most common issues:

- Replication stops and falls behind
- Queries fail on the slave for some reason

Let's attack one problem at a time.

Fixing stopped replication

One of the most common problems is that at some point, replication just stops. The slave will start to fall behind, and the result coming back from a slave will be old and outdated. There are a couple of reasons for this kind of behavior:

- Constant lack of bandwidth
- Replication conflicts

In many cases, bandwidth is a serious issue. If two servers are connected using a 100 MB interconnect, it can easily happen that the slave falls behind and eventually dies if the write load is constantly higher than 10 MB per second. Keep in mind that the master only keeps a couple of `xlog` files around, which are needed to recover in case of a crash of the master. In the default configuration, it does not queue transaction logs forever just to handle problems on the slaves. If the load is too high on a systematic basis, this would not help anyway.

In PostgreSQL, the slave requests the transaction log it needs from the master. If the master does not have the desired `xlog` file any more, the slave cannot continue. To many users, this looks like a replication failure, which it really is not. It is clearly desired behavior. To reduce the problem straight away, two options are available:

- Set `wal_keep_segments`
- Use replication slots (a crash-safe feature introduced in PostgreSQL 9.4 that can be used to avoid premature removal of `xlog`)

In general, it is always recommended to use `wal_keep_segments`, unless replication slots are used. Otherwise, it is too risky and a slave is likely to fall much behind its master. When configuring `wal_keep_segments`, be gracious and don't use settings that are too low. A nice `xlog` retention pool can really pay off.



Keep in mind that if you are constantly using more bandwidth than is available, `wal_keep_segments` won't help.

Replication slots are a nice alternative to `wal_keep_segments`. In the case of a normal setup using `wal_keep_segments`, a master will keep as much `xlog` as configured by the administrator. But how can the administrator know this? Usually, there is no way to know.

A replication slot can help because it keeps `xlog` around as long as it is needed. Here is how it works:

```
postgres=# SELECT * FROM   pg_create_physical_replication_slot('rep_
slot');
 slot_name | xlog_position
-----+-----
 rep_slot  |
```

It is easy to check for existing replication slots:

```
postgres=# SELECT slot_name, slot_type FROM pg_replication_slots;
 slot_name | slot_type
-----+-----
 rep_slot  | physical
(1 row)
```

In the replica, the name of the replication slot has to be added to `recovery.conf`:

```
primary_slot_name = 'rep_slot'
```



Keep in mind that if the slave goes down, the master might fill up. Therefore, it is essential to monitor this kind of setup.

Fixing failed queries

Another important issue is related to failed queries. It can happen that queries are cancelled on the slave due to replication conflicts. If you are working on a system that constantly kicks queries, it can be very helpful to check out `hot_standby_feedback`.

The idea is to let the slave report its oldest transaction ID from time to time. The master can then react and ensure that conflicts are prevented by delaying `VACUUM` on itself. In reality, setting `hot_standby_feedback` on the slave leaves us with the same behavior as if the query had actually run on the master (not performance-wise, of course).

If the people working on those slave systems can be trusted, it usually makes sense to turn `hot_standby_feedback` on. If users on the slave are not considered to be competent enough, it is safer to stick to the default value — off.

Summary

In this chapter, a typical customer request was outlined and discussed step by step. The goal was to systematically outline the steps to approach an existing system that faces typical everyday problems.



Keep in mind that my personal strategy is not suitable for every purpose but it helps to tackle a typical troubled system anywhere in the world.

Index

Symbol

7th normal form 18, 19

A

arrays

versus normalization 19, 20

asynchronous replication

about 107, 108

firing up 110, 111

pg_basebackup, working with 108, 109

safety 112, 113

slaves, promoting to masters 112

B

backend process (BE) 10

binary trees

URL 24

bit fields

versus boolean fields 15, 16

C

Cacti

about 95

URL 95

checksums

adding, to database instance 8

cidr

versus text 16, 17

circle

versus text 16, 17

clock_timestamp() function 76

columns

grouping 11, 12

conflicts

checking 89, 90

contention 61

cursors 47-50, 80, 81

custom format dumps

creating 101

multiple CPUs, using 102

D

data

large amounts, reading 47

resetting 92

database

checking 83

pg_stat_activity, checking 84, 85

pg_stat_database, checking 86, 87

database instance

about 3

checksums, adding to 8

data type

finding 13

deadlocks 58-60

Debian packages

installing 4

distance operator 34

E

encoding related issues

preventing 8, 9

error message, PostgreSQL 9.2

DETAIL 5

- FATAL 5
- HINT 5
- execution plan** 22

F

- floating point**
 - versus numeric 14
- floating-point unit (FPU)** 14
- foreign keys**
 - managing 29
- FOR UPDATE mode**
 - locking in 60, 61
- full-text search**
 - fixing 35
 - not using 35-38

G

- geometric data**
 - indexing, GiST used 30
- GIN index** 38
- GiST**
 - used, for indexing geometric data 29, 30
- GiST index** 29

I

- index corruption**
 - dealing with 120
- indexes**
 - foreign keys, managing 29
 - geometric data indexing, GiST used 29, 30
 - in PostgreSQL 21, 22
 - missing indexes, detecting 26
 - reviewing 128, 129
 - simple 22-24
 - trouble, avoiding 25, 26
 - troubleshooting 29
 - useless indexes, removing 27, 28
 - working 24
- inet**
 - versus text 16, 17
- installation process, PostgreSQL**
 - about 2
 - Debian packages, installing 4
 - RPM packages, installing 3

- internal information**
 - inspecting 92
 - I/O cache, inspecting 94, 95
 - table, inspecting 92-94
- I/O bottlenecks**
 - detecting 88
- issues** 124

J

- JavaScript Object Notation (JSON)** 96
- joins**
 - about 43
 - demo data, creating 44
 - outer joins 44-47

K

- kernel issues**
 - about 5
 - fixing 7
- kernel parameters**
 - adjusting, for Linux 6
 - adjusting, for Mac OS X 7
- kernel variables**
 - SEMAP 6
 - SEMMNI 6
 - SEMMNS 6
 - SEMMSL 6
 - SEMMVMX 6
- KNN** 34, 35

L

- language**
 - selecting 71, 72
- LEAKPROOF**
 - versus NOT LEAKPROOF 77-80
- LIKE queries**
 - advanced 32, 33
 - handling 30
 - simple 31, 32
- Linux**
 - kernel parameters, adjusting for 6
- lock**
 - about 58
 - inspecting 66-69
 - URL 63

M

Mac OS X

- kernel parameters, adjusting for 7

memory consumption, issues

- about 80
- cursors 80, 81
- memory parameters,
 - assigning to functions 81
- procedures 80, 81
- set-returning functions, handling 81

memory issues

- about 5
- fixing 5, 6

monitoring tools

- Nagios plugins, using 95, 96
- pganalyze-collector 96, 97
- pg_view 97
- using 95
- Zabbix plugin 96

Multi Router Traffic Grapher (MRTG)

- about 95
- URL 95

Multi-Version Concurrency Control (MVCC) 92

N

Nagios plugins

- checks 96
- using 95

Nearest neighbor search. *See* KNN

normalization

- 7th normal form 18
- about 13
- deciding on 17
- versus arrays 19, 20

NOT LEAKPROOF

- versus LEAKPROOF 77-80

NULL

- about 41, 42
- and storage 43
- in action 42, 43

NULL bitmap 43

numeric

- versus floating point 14

O

outer joins 44-47

P

pages

- damaged pages, zeroing out 118, 119
- header, extracting 121
- individual pages, dumping 120

performance

- about 128
- indexes, reviewing 128, 129
- slow queries, detecting 131
- UPDATE commands, fixing 130

pganalyze-collector

- URL 97
- using 96

pg_basebackup

- working with 108, 109

pg_dump

- textual dumps, creating 99, 100
- using 99

pg_monz

- about 96
- URL 96

pg_view

- URL 97
- using 97

point

- versus text 16, 17

point-in-time recovery (PITR)

- about 102
- PostgreSQL, preparing for 103
- working 103
- xlog, replaying 105, 106

PostGIS project

- URL 29

PostgreSQL

- base backups, taking 104, 105
- documentation, URL 6
- indexes 21, 22
- installing 2
- preparing, for PITR 103
- URL 62

PostgreSQL transaction model

- about 55-57
- FOR UPDATE mode, locking in 60, 61
- performance bottlenecks, avoiding 61, 62
- savepoints 57
- table locks, avoiding 62

postmaster

- killing 10

prepared queries 51, 52

procedural language

- enabling 72
- URL 71

procedures

- about 80, 81
- and indexing 75-77
- and transactions 74, 75
- IMMUTABLE 77
- managing 72, 73
- NOT LEAKPROOF 77
- STABLE 76
- VOLATILE 75

Q

queries

- checking 90, 91
- data, resetting 92
- overhead 92
- slow queries, fixing 131

R

random() function 76

read committed mode

- demonstrating 63, 64

repeatable read

- using 65

replication, issues

- failed queries, fixing 134
- fixing 132, 133

RPM packages

- installing 3

S

savepoints 57

scanning

- synchronized scanning 50, 51

semaphores 6

sequential scan 23, 24

silent corruption

- preventing 118

SQL, execution

- Executor 23
- Optimizer stage 23
- Parser stage 22
- Rewrite system stage 22

synchronous replication

- switching to 113, 114

T

table

- inspecting 92-94
- locks, avoiding 62

template pollution

- avoiding 9

text

- versus cidr 16
- versus circle 16
- versus inet 16
- versus point 16
- versus varchar 13, 14

textual dumps

- blobs, handling 100
- creating 99, 100
- passwords, handling 100

timelines

- handling 114

transaction isolation

- about 63
- read committed mode,
 - demonstrating 63, 64
- repeatable read 65, 66
- repeatable read, using 65

transaction model, PostgreSQL

- about 56, 57
- deadlocks 59, 60
- locking 58
- savepoints 57

transactions

- and procedures 74, 75
- log, resetting 122, 123
- managing 72, 73
- using 73, 74

Trigrams 33

troubleshooting

about 127, 128

low performance 128

trusted language

versus untrusted language 72

U

UPDATE commands

fixing 130

V

varchar

versus text 13, 14

version number, PostgreSQL

deciding on 1, 2

Z

Zabbix plugin

using 96



Thank you for buying Troubleshooting PostgreSQL

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 that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

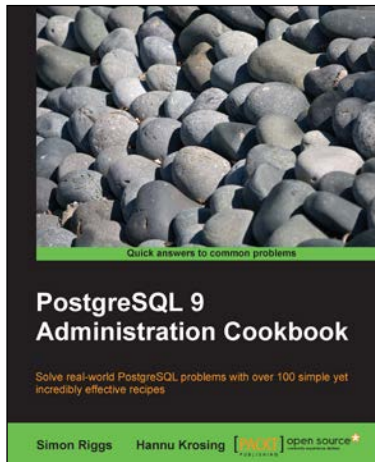
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

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, then please 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.



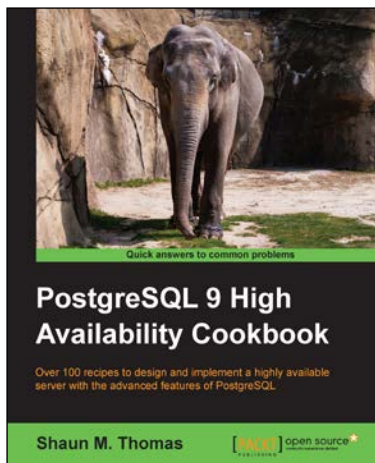
PostgreSQL 9 Administration Cookbook

ISBN: 978-1-84951-028-8

Paperback: 360 pages

Solve real-world PostgreSQL problems with over 100 simple yet incredibly effective recipes

1. Administer and maintain a healthy database.
2. Monitor your database ensuring that it performs as quickly as possible.
3. Tips for backup and recovery of your database.



PostgreSQL 9 High Availability Cookbook

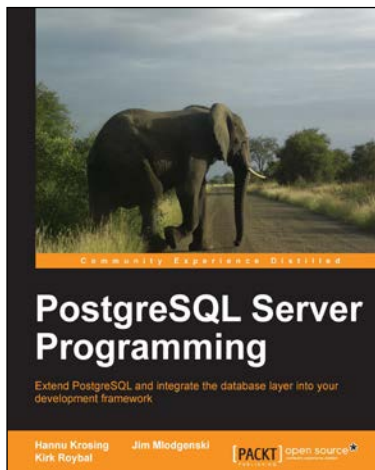
ISBN: 978-1-84951-696-9

Paperback: 398 pages

Over 100 recipes to design and implement a highly available server with the advanced features of PostgreSQL

1. Create a PostgreSQL cluster that stays online even when disaster strikes.
2. Avoid costly downtime and data loss that can ruin your business.
3. Perform data replication and monitor your data with hands-on industry-driven recipes and detailed step-by-step explanations.

Please check www.PacktPub.com for information on our titles



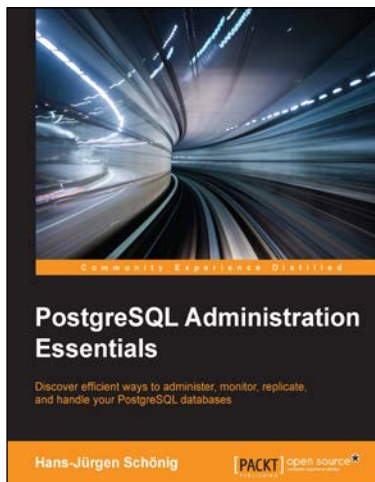
PostgreSQL Server Programming

ISBN: 978-1-84951-698-3

Paperback: 264 pages

Extend PostgreSQL and integrate 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 favorite programming language.
3. Step-by-step tutorial with plenty of tips and tricks to kick-start server programming.



PostgreSQL Administration Essentials

ISBN: 978-1-78398-898-3

Paperback: 142 pages

Discover efficient ways to administer, monitor, replicate, and handle your PostgreSQL databases

1. Learn how to detect bottlenecks and make sure your database systems offer superior performance to your end users.
2. Replicate your databases to achieve full redundancy and create backups quickly and easily.
3. Optimize PostgreSQL configuration parameters and turn your database server into a high-performance machine capable of fulfilling your needs.

Please check www.PacktPub.com for information on our titles