



Quick answers to common problems

# MySQL Admin Cookbook

99 great recipes for mastering MySQL configuration  
and administration

# **MySQL Admin Cookbook**

99 great recipes for mastering MySQL configuration  
administration

**Daniel Schneller**

**Udo Schwedt**



# MySQL Admin Cookbook

Copyright © 2010 Packt Publishing

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

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

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

First published: March 2010

Production Reference: 1080310

Published by Packt Publishing Ltd.  
32 Lincoln Road  
Olton  
Birmingham, B27 6PA, UK.

ISBN 978-1-847197-96-2

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Vinayak Chittar ([vinayak.chittar@gmail.com](mailto:vinayak.chittar@gmail.com))

# Credits

**Authors**

Daniel Schneller  
Udo Schwedt

**Reviewers**

Kai Seidler  
Marc Delisle

**Acquisition Editor**

Sarah Cullington

**Development Editor**

Reshma Sundaresan

**Technical Editors**

Pallavi Kachare  
Bhupali Khule  
Aaron Rosario

**Copy Editor**

Lakshmi Menon

**Editorial Team Leader**

Akshara Aware

**Indexer**

Rekha Nair

**Project Team Leader**

Lata Basantani

**Project Coordinator**

Shubhanjan Chatterjee

**Proofreader**

Chris Smith

**Graphics**

Geetanjali Sawant

**Production Coordinator**

Melwyn Arun D'sa

**Cover Work**

Melwyn Arun D'sa

# About the Author

**Daniel Schneller** works as a software developer, database administrator, and general IT professional for an independent software vendor in the retail sector. After successfully graduating from the University of Cooperative Education in Heidenheim, Germany with a degree in Business Computer Science, he started his career as a professional software developer, focused on the Microsoft technology stack. In 2002, he started focusing on enterprise-level Java development and has since gained extensive knowledge and experience implementing large scale systems based on Java EE and relational databases, especially MySQL since version 4.0.

Currently, he is mostly involved with the ongoing development of framework-level functionality, including customization and extension of an ORM-based persistence layer. He is involved in different open source projects such as FindBugs, Eclipse, and Chakra, and infrequently blogs about Java, MySQL, Windows, Linux, and other insanities at <http://www.danielschneller.com>.

---

When I first was asked by Packt Publishing whether I would be interested in writing a book about MySQL on Christmas Eve 2008 little did I know how much work, stress, but also what a lot of fun I was headed for.

Now, that the book is finally done I would like to thank those people without whom getting it done would have been impossible.

First of all, I'd like to thank Udo for agreeing to be my co-author. Without him this whole thing would have taken a lot longer and would have been not half as useful as I believe it has turned out now.

I would also like to thank the team at Packt Publishing—most importantly for noticing and reading my blog, consequently contacting me to get the whole thing started—but also for taking care of schedules, providing support, guidance and feedback, and keeping us on track the whole way.

Last, but by no means least, I want to thank Jenny—for encouraging me to write a book in the first place, and then making sure I never ran out of tea, cookies, or motivation on the countless evenings I spent sitting in front of the computer writing code.

**Udo Schwedt** has over ten years of experience in the IT industry as a professional developer and software architect. He is head of the Java architecture team and depu... the Java development department at the IT service provider for Germany's market... the Do-It-Yourself sector.

He has been fascinated by computers since his childhood, and taught himself the programming during his school years. After graduating from school, he began his studies at the RWTH Aachen, Germany, which he finished with a summa cum laude degree in computer science, minoring in psychology with a focus on software ergonomics.

Udo started his career as a professional C, C++, and Java developer in a software company that delivers leading solutions in the financial online transaction processing sector. In 2008, he joined his current employer as a Java framework developer for a large-scale international project, where he met Daniel. In the course of the project, he gained extensive experience using MySQL in a professional context.

For both **Daniel** and **Udo**, the common project involved the design and implementation of a database infrastructure solution for a Java-based merchandise management software system with tens of thousands of clients. The evaluation of different database systems and the realization of the infrastructure made it necessary for them to delve into MySQL beyond its typical utilization scenarios. The resulting decentralized multi-platform environment with more than 500 decentralized MySQL server instances with more than 5,500 replication slaves bears challenges not covered by the standard MySQL documentation.

---

To the Packt Publishing team: Thank you for critiques, encouragement, and organization.

To Daniel: Thank you so much for your confidence in me. I still feel honored that you asked me to co-author this book—you should know better by now!

To my parents: Thank you for supporting me from the very start and ever since.

To Katharina, Johannah, and Frida: Thank you for your support and all your patience—I love you!

---

# About the Review

**Kai Seidler** was born in Hamburg in 1970. He graduated from the Technical University of Berlin with a Diplom Informatiker degree (Master of Science equivalent) in Computer Science. In the 90s he created and managed Germany's biggest IRCnet server irc.de, and co-managed one of the world's largest anonymous FTP server ftp.cs.tu-berlin.de. He professionally set up his first public web servers in 1993. From 1993 until 1996 he was a member of Projektgruppe Kulturräum Internet, a research project on net culture and new media. In 2002, he co-founded Apache Friends and created the multi-platform Apache web server bundle XAMPP. Around 2005, XAMPP became the most popular Apache distribution worldwide. In 2006, his third book, *Das XAMPP-Handbuch*, was published by Addison Wesley.

Currently he's working as a Technology Evangelist for web-tier products at Sun Microsystems.

**Marc Delisle** is a member of the MySQL Developers Guild—which brings together MySQL community developers—because of his involvement with phpMyAdmin. He started contributing to this popular MySQL web interface in December 1998, when he made the first multi-language version. He has been actively involved with this software project since 2001 as a developer and project administrator.

Marc has worked since 1980 at Cegep de Sherbrooke, Québec, Canada, as an application programmer and network manager. He has also been teaching networking, security, and PHP/MySQL application development. Marc lives in Sherbrooke with his wife and two sons, spending time with their four children.

Marc authored the first ever Packt Publishing book, *Mastering phpMyAdmin for Efficient MySQL Management*, and its revised editions. He also wrote *Creating your MySQL Database: Practical Design Tips and Techniques*, again with Packt Publishing.

---

I would like to thank the fine team at Packt for their support in reviewing this book.

# Table of Conte

## Preface

---

### Chapter :1 Replication

---

Introduction

Setting up automatically updated slaves of a server based on a SQL du

Setting up automatically updated slaves of a selection of tables based  
on a SQL dump

Setting up automatically updated slaves using data file copy

Sharing read load across multiple machines

Using replication to provide full-text indexing for InnoDB tables

Estimating network and slave I/O load

Limiting network and slave I/O load in heavy write scenarios using  
the blackhole storage engine

Setting up slaves via network streaming

Skipping problematic queries

Checking if servers are in sync

Avoiding duplicate server IDs

Setting up slaves to report custom information about themselves  
to the master

### Chapter :2 Indexing

---

Introduction

Adding indexes to tables

Adding a fulltext index

Creating a normalized text search column

Removing indexes from tables

Estimating InnoDB index space requirements

Using prefix primary keys

*Table of Contents* —

**Finding duplicate indexes**

**Chapter :3 Tools**

**Introduction**

**Transferring connection settings between different machines using a network share**

**Sorting MySQL GUI Tools' stored connections**

**Automatically creating stored connections**

**Adding custom graphs to MySQL Administrator**

**Displaying query results page by page and with scrolling using the MySQL command-line client**

**Extracting information from verbose output using the MySQL command-line client**

**Specifying a default pager**

**Using a custom prompt to distinguish connections**

**Encrypting a MySQL server connection with SSH**

**Creating an encrypted MySQL console via SSH**

**Using a PuTTY template connection for SSH secured connections**

**Chapter :4 Backing Up and Restoring MySQL Data**

**Introduction**

**Using MySQL Administrator GUI Tool as a frontend for backups**

**Copying all data files to a backup location**

**Creating a SQL dump of all databases**

**Creating a SQL dump of specific databases**

**Creating a SQL dump of specific tables**

**Compressing SQL dumps on-the-fly**

**Rotating and purging binary logs**

**Using replication to perform backups without hurting a production system's performance**

**Restoring data from a dump to a previously backed-up state**

**Performing a point-in-time recovery using the binary logs**

**Chapter :5 Managing Data**

**Introduction**

**Exporting data to a simple CSV file**

**Exporting data to a custom file format**

**Using stored procedures to export repeatedly**

**Importing data from a simple CSV file**

**Importing data from custom file formats**

**Inserting new data and updating data if it already exists**

---

**Deleting all but a fragment of a large table's data**  
**Deleting all data incrementally from large tables**

## **Chapter :6 Monitoring and Analyzing a MySQL Installation**

**Introduction**  
**Checking free InnoDB tablespace**  
**Establishing alerting mechanisms for low remaining tablespace by using triggers**  
**Estimating tablespace requirements**  
**Identifying and changing MySQL variables**  
**Assessing the overall table count**  
**Finding the biggest tables**  
**Finding all columns with a certain name and/or type**  
**Finding all tables referencing each other**

## **Chapter :7 Configuring MySQL**

**Introduction**  
**Setting up a fixed InnoDB tablespace**  
**Setting up an auto-extending InnoDB tablespace**  
**Storing InnoDB data in one file per table**  
**Decreasing InnoDB tablespace**  
**Enabling and configuring binary logging**  
**Configuring the InnoDB redo log**  
**Understanding and configuring important MySQL and InnoDB timeout options**  
**Adjusting table and database name letter case handling for better platform independence**  
**Installing MySQL as a Windows service with custom options**  
**Running multiple MySQL server instances in parallel on a Linux server**  
**Preventing invalid date values from being stored in DATE or DATETIME columns**

## **Chapter :8 MySQL User Management**

**Introduction**  
**Configuring MySQL Administrator to display global privileges and hosts**  
**Defining an alternative user for administrative tasks**  
**Disabling the default accounts**  
**Creating a basic user**  
**Creating an installation user**  
**Creating a read-only account**  
**Defining a specific user for backup**

*Table of Contents* —

**Regaining access to your database in case of lost account information**  
**Avoiding plain text passwords in administrative scripts**

**Chapter :9 Managing Schemas**

**Introduction**

**Adding new columns at specific positions**

**Defining a primary key for a table containing (non-unique) data**

**Allowing individual INSERT statements with "0" values in auto-incrementing columns**

**Globally allowing INSERT statements with "0" values in auto-incrementing columns**

**Choosing a suitable storage engine**

**Improving the performance of ALTER TABLE for InnoDB**

**Using a stored procedure to conditionally add columns or indexes**

**Improving query performance for InnoDB tables with BLOB columns**

**Identifying differences between two schemas**

**Comparing schema revisions using hash values**

**Appendix : Good to Know**

**Introduction**

**Avoiding silent replication disruption on full master disk**

**Maximizing usable memory on 32-bit Windows**

**Using separate temporary directories for multiple MySQL servers on a single machine, preventing conflicts**

**Preventing mysqldump from failing with Error 2013**

**Non-availability of InnoDB may escape monitoring**

**Troubleshooting "Can't start server: Bind on TCP/IP port: No such file or directory" error**

**Choosing character sets**

**Understanding auto-increment values**

# Prefa

MySQL is the most popular open-source database and is also known for its easy setup feature. However, proper configuration beyond the default settings is still a challenge, along with some other day-to-day maintenance tasks such as backup and restoring, performance tuning, and server monitoring.

This book provides both step-by-step recipes and relevant background information on various topics and more. It covers everything from basic to advanced aspects of MySQL administration and configuration. All recipes are based on real-world experience and were derived from proven solutions used in an enterprise environment.

## What this book covers

Chapter 1, *Replication*: In this chapter, you will see how to set up MySQL replication, including load balancing, online backups, and fail-over scenarios. Advanced replication scenarios such as the blackhole engine and streaming slave deployment are discussed beyond the basics.

Chapter 2, *Indexing*: You will be shown how to create, drop, and modify indexes, perhaps the most important means of optimizing your MySQL servers' performance. Fulltext and clustered and non-clustered indexes are compared and presented with their respective strengths and typical use cases. Moreover, you will learn how to identify duplicate indexes which can hinder your servers' performance.

Chapter 3, *Tools*: This chapter will get you acquainted with the MySQL Administrator, MySQL Workbench GUI Tools as well as the MySQL command-line client and how to use it in conjunction with external scripts and tools. You will also see how to create custom diagrams for the MySQL Administrator and share connection profiles between multiple computers.

Chapter 4, *Backing Up and Restoring MySQL Data*: In this chapter, we introduce the various approaches to backing up your database and restoring data again. Advanced techniques such as on-the-fly compression, point in time recovery, avoiding extended lock situations, binlog replication scenarios, and partial backup and restore are also covered.

## Preface

---

**Chapter 5, Managing Data:** You will learn some tricks beyond the basic SQL commands which enable you to delete data in a highly efficient manner and insert data based on database content, and how to import and export data to and from your database.

**Chapter 6, Monitoring and Analyzing a MySQL Installation:** We present approaches for monitoring table space usage, and how to use database metadata to your advantage. Performance bottlenecks and lock contention problems are discussed as well.

**Chapter 7, Configuring MySQL:** This chapter deals with MySQL configuration and how to leverage available settings to their full potential. Table space management, pool sizes, and logging options are discussed along with platform-specific caveats and advanced installation scenarios, such as multiple instances on one server.

**Chapter 8, MySQL User Management:** Management of MySQL user accounts is discussed in detail throughout this chapter. Typical user roles with appropriate privileges and approaches to restricting access sensibly are proposed. You will also learn how to regain access to a database in case the administrative user credentials are lost.

**Chapter 9, Managing Schemas:** This chapter includes topics such as adding and removing columns to and from tables and choosing a suitable storage engine and character set for individual needs. Another recipe covers a technique to add a new primary key column to a table already filled with data. Ways to manage and automate database schema evolution as part of a software life cycle are presented as well. And if you have always missed "IF NOT EXISTS", you will find a solution to this, too.

**Appendix, Good to Know:** In this final part of the book you can find several things that might be useful in everyday situations, but did not fit the step-by-step recipe format naturally. They range from choosing character sets to getting the most out of 32 bit address space.

## What you need for this book

This book was written using MySQL versions 5.0 and 5.1. Most recipes will work equally well on either one of these versions. Older versions might work as well, but have not been tested. You can download both versions of the MySQL server from <http://dev.mysql.com>. You will find references to programs and tools not included in the MySQL server distribution. These can be downloaded from their respective websites, named in the recipes. The "GUI Tools"—MySQL Administrator and MySQL Query Browser—which are referenced several times throughout the book—unfortunately have been declared End Of Life shortly before this book was finished. Currently, there is no functionally equivalent successor to these tools. "MySQL Workbench" is the new combined tool recommended on the MySQL website, but it does not offer all features required to apply many of the recipes in this book. We recommend that you download MySQL Administrator and MySQL Query Browser from the MySQL archive area where they are still available. You will find them by just using the links provided in the book.

---

## Who this book is for

This book is for ambitious MySQL users as well as professional data center database administrators. Beginners as well as experienced administrators will profit from this book and get fresh ideas to improve their MySQL environments. Detailed background information will enable them to widen their MySQL horizon.

It does not cover SQL basics, how to install MySQL servers, or how to design a relational database schema. Readers are expected to have a basic understanding of the SQL language and database concepts in general.

## Conventions

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

Code words in text are shown as follows: "Only use qualified statements and `replicate_wild_table` configuration options for intuitively predictable replication!"

A block of code is set as follows:

```
slave> create database sakila;
slave> use sakila;
slave> source /tmp/sakila_master.sql;
slave> CHANGE MASTER TO master_host='master.example.com', master_port=3306, master_user='repl', master_password='slavepass';
slave> START SLAVE;
```

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

```
slave> SHOW SLAVE STATUS\G
***** 1. row *****
...
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
...
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "You will see the famous messages about InnoDB filling up the data files and finally, the **Ready for connection**"

## Preface



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 the book—what you liked or may have disliked. Reader feedback is important for us to help others get the most out of titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com). Please mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a suggestion. Visit the **SUGGEST A TITLE** form on [www.packtpub.com](http://www.packtpub.com) or e-mail [suggest@packtpub.com](mailto:suggest@packtpub.com).

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book on, see our author guide on [www.packtpub.com/author](http://www.packtpub.com/author).

## Customer support

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



### Downloading the example code for the book

Visit [http://www.packtpub.com/files/code/7962\\_Code.zip](http://www.packtpub.com/files/code/7962_Code.zip) to directly download the example code.

---

## 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 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/support>, selecting your book, clicking on the **let us know** 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. We take the protection of our copyright and licenses very seriously. If you come across illegal copies of our works, in any form, on the Internet, please provide us with the address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected piracy.

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](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.



# Replicat

In this chapter, we will discuss:

- ▶ Setting up automatically updated slaves of a server based on a SQL dump
- ▶ Setting up automatically updated slaves of a selection of tables based on a
- ▶ Setting up automatically updated slaves using data file copy
- ▶ Sharing read load across multiple machines
- ▶ Using replication to provide full-text indexing for InnoDB tables
- ▶ Estimating network and slave I/O load
- ▶ Limiting network and slave I/O load in heavy write scenarios using the bla storage engine
- ▶ Setting up slaves via network streaming
- ▶ Skipping problematic queries
- ▶ Checking if servers are in sync
- ▶ Avoiding duplicate server IDs
- ▶ Setting up slaves to report custom information about themselves to the m

## Introduction

Replication is an interesting feature of MySQL that can be used for a variety of pur It can help to balance server load across multiple machines, ease backups, provide workaround for the lack of fulltext search capabilities in InnoDB, and much more.

## *Replication*

The basic idea behind replication is to reflect the contents of one database server (which can include all databases, only some of them, or even just a few tables) to more than one instance. Usually, those instances will be running on separate machines, even though it is not technically necessary.

Traditionally, MySQL replication is based on the surprisingly simple idea of repeating the execution of all statements issued that can modify data—not SELECT—against a single machine on other machines as well. Provided all secondary *slave* machines had identical contents when the replication process began, they should automatically remain in sync. This is called **Statement Based Replication (SBR)**.

With MySQL 5.1, **Row Based Replication (RBR)** was added as an alternative method of replication, targeting some of the deficiencies SBR brings with it. While at first glance RBR may seem superior (and more reliable), it is not a silver bullet—the pain points of RBR are different from those of SBR.

Even though there are certain use cases for RBR, all recipes in this chapter will be about Statement Based Replication.

While MySQL makes replication generally easy to use, it is still important to understand what happens internally to be able to know the limitations and consequences of the choices and decisions you will have to make. We assume you already have a basic understanding of replication in general, but we will still go into a few important details.

## **Statement Based Replication**

SBR is based on a simple but effective principle: if two or more machines have the same initial set of data to begin with, they will remain identical if all of them execute the exact same sequence of statements in the same order.

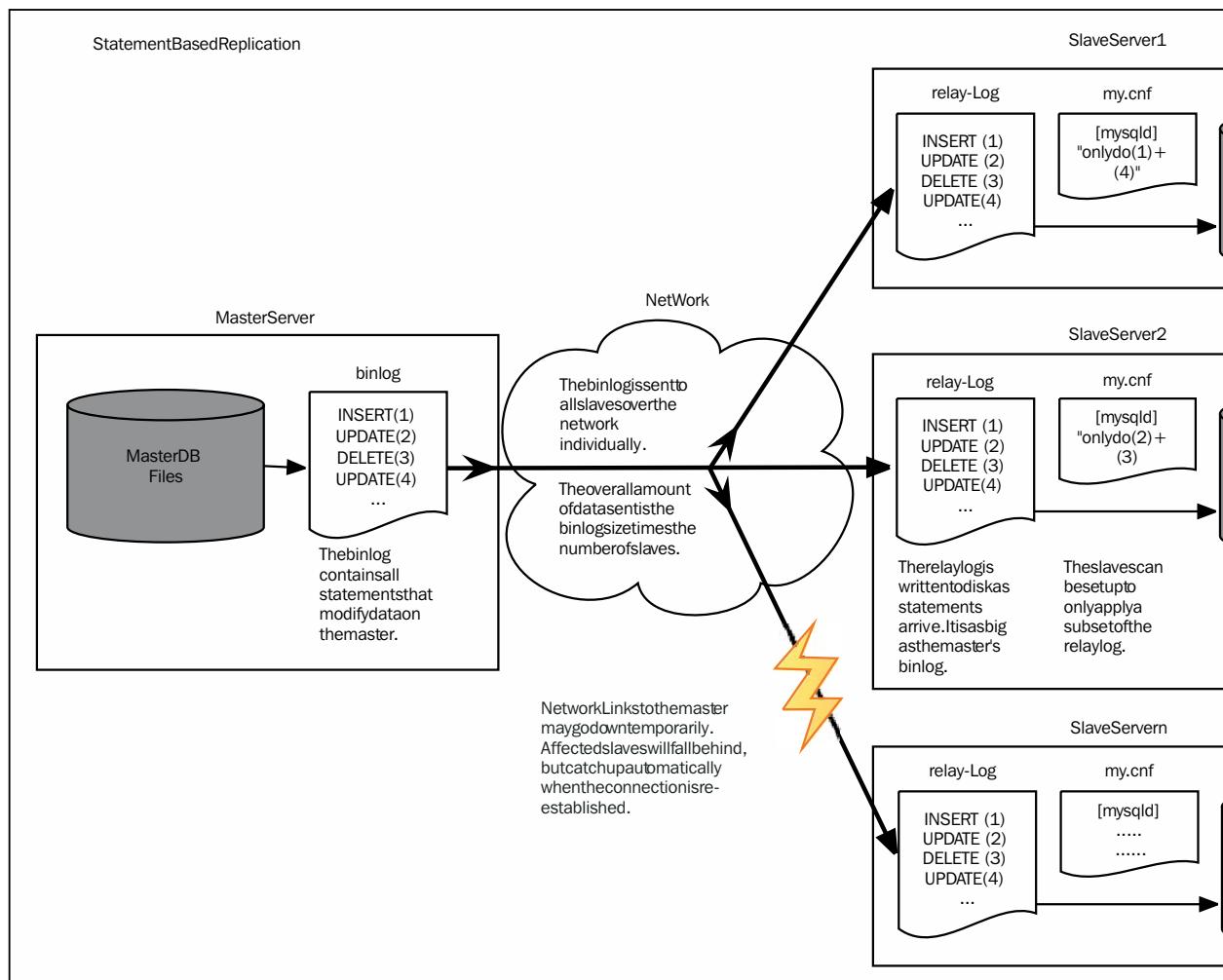
Executing all statements manually on multiple machines would be extremely tedious and impractical. SBR automates this process. In simple terms, it takes care of sending the SQL statements that change data on one server (the master) to any number of additional instances (the slaves) over the network.

The slaves receiving this stream of modification statements execute them automatically, thereby effectively reproducing the changes the master machine made to its data. That way they will keep their local data files in sync with the master's.

One thing worth noting here is that the network connection between the master and slave(s) need not be permanent. In case the link between a slave and its master fails, the slave will remember up to which point it had read the data last time and will continue reading there once the network becomes available again.

In order to minimize the dependency on the network link, the slaves will retrieve the logs (*binlogs*) from the master as quickly as they can, storing them on their local disk files called *relay logs*. This way, the connection, which might be some sort of dial-up, can be terminated much sooner while executing the statements from the local *relay log* asynchronously. The relay log is just a copy of the master's *binlog*.

The following image shows the overall architecture:



## Filtering

In the image you can see that each slave may have its individual configuration on what it executes. Some slaves might execute all the statements coming in from the master, or just a selection of those. This can be helpful when you have some slaves dedicated to special tasks, where they might not need all the information from the master.

All of the binary logs have to be sent to each slave, even though it might then decide to throw away most of them. Depending on the size of the binlogs, the number of slaves, and the bandwidth of the connections in between, this can be a heavy burden on the network, especially if you are replicating via wide area networks.

## *Replication*

Even though the general idea of transferring SQL statements over the wire is rather simple, there are lots of things that can go wrong, especially because MySQL offers configuration options that are quite counter-intuitive and lead to hard-to-find problems.

For us, this has become a best practice:

"Only use qualified statements and `replicate-*--table` configuration options for predictable replication!"

What this means is that the only filtering rules that produce intuitive results are those on the `replicate-do-table` and `replicate-ignore-table` configuration options. This includes those variants with wildcards, but specifically excludes the `all-database` option. `replicate-do-db` and `replicate-ignore-db`. These directives are applied on the master side on all incoming relay logs.

The master-side `binlog-do-*` and `binlog-ignore-*` configuration directives indicate which statements are sent to the binlog and which are not. We strongly recommend not using them, because apart from hard-to-predict results they will make the binlogs unusable for server backup and restore. They are often of limited use anyway as they do not allow individual configurations per slave but apply to all of them.

For these reasons you will not find any use of these options in this book.

## **Setting up automatically updated slaves a server based on a SQL dump**

In this recipe, we will show you how to prepare a dump file of a MySQL master server and use it to set up one or more replication slaves. These will automatically be updated with changes made on the master server over the network.

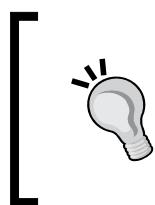
### **Getting ready**

You will need a running MySQL master database server that will act as the *replicator*, and at least one more server to act as a *replication slave*. This needs to be a separate instance with its own data directory and configuration. It can reside on the same machine if you just want to try this out. In practice, a second machine is recommended because this technique's very goal is to distribute data across multiple pieces of hardware, not put an even higher burden on a single one.

For production systems you should pick a time to do this when there is a lighter load on the master machine, often during the night when there are less users accessing the system. Taking the SQL dump uses some extra resources, but unless your server is maxed out, the performance impact usually is not a serious problem. Exactly how long the dump takes

You will need an administrative operating system account on the master and the slave to edit the MySQL server configuration files on both of them. Moreover, an administrative MySQL database user is required to set up replication.

We will just replicate a single database called `sakila` in this example.



### **Replicating more than one database**

In case you want to replicate more than one schema, just add their names to the commands shown below. To replicate all of them, just leave out any database name from the command line.

## **How to do it...**

1. At the operating system level, connect to the master machine and edit the MySQL configuration file with a text editor. Usually it is called `my.ini` and `my.cnf` on other operating systems.
 

```
server-id=1000
log-bin=master-bin
```

If one or both entries already exist, do not change them but simply note them down. The `log-bin` setting need not have a value, but can stand alone as well.
2. On the master machine, make sure the following entries are present and add the `[mysqld]` section if not already there:
 

```
server-id=1000
log-bin=master-bin
```
3. Restart the master server if you need to modify the configuration file.
4. Create a user account on the master that can be used by the slaves.
 

```
master> grant replication slave on *.* to 'rep1'@'%' identified by 'slavepass';
```
5. Using the `mysqldump` tool included in the default MySQL install, create the dump file to set up the slave(s):
 

```
$ mysqldump -uUSER -pPASS --master-data --single-transaction sakila > sakila_master.sql
```
6. Transfer the `sakila_master.sql` dump file to each slave you want to replicate, for example, by using an external drive or network copy.

## Replication

7. On the slave, make sure the following entries are present and added to the [mysqld] section if not present:

```
server-id=1001
replicate-wild-do-table=sakila.%
```

When adding more than one slave, make sure the server-id setting is unique among master and all clients.

8. Restart the MySQL service.

9. Connect to the slave server and issue the following commands (assuming the full content dump was stored in the /tmp directory):

```
slave> create database sakila;
slave> use sakila;
slave> source /tmp/sakila_master.sql;
slave> CHANGE MASTER TO master_host='master.example.com',
       master_port=3306, master_user='repl',
       master_password='slavepass';
slave> START SLAVE;
```

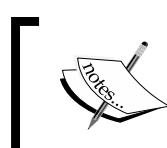
10. Verify the slave is running.

```
slave> SHOW SLAVE STATUS\G
***** 1. row *****
...
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
...
```

## How it works...

Some of the instructions discussed in the previous section are to make sure that the master and slave are configured with different server-id settings. This is of paramount importance for a successful replication setup. If you fail to provide unique server-id values for different server instances, you might see strange replication errors that are hard to debug.

Moreover, the master must be configured to write *binlogs*—a record of all statements that are manipulating data (this is what the slaves will receive).



Before taking a full content dump of the sakila demo database, we create a user account for the slaves to use. This needs the REPLICATION SLAVE privilege.

---

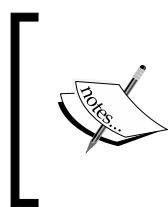
Then a data dump is created with the `mysqldump` command line tool. Notice the parameters `--master-data` and `--single-transaction`. The former is needed for `mysqldump` to include information about the precise moment the dump was created in the resulting output. The latter parameter is important when using InnoDB tables, because then will the dump be created based on a transactional snapshot of the data. Without these statements changing data while the tool was running could lead to an inconsistent dump.

The output of the command is redirected to the `/tmp/sakila_master.sql` file. Since the `sakila` database is not very big, you should not see any problems. However, if you plan to use this recipe to larger databases, make sure you send the data to a volume with sufficient space—the SQL dump can become quite large. To save space here, you may optionally compress the output through `gzip` or `bzip2` at the cost of a higher CPU load on both the master and the slaves, because they will need to unpack the dump before they can load it, of course.

If you open the uncompressed dump file with an editor, you will see a line with a `CHANGE MASTER TO` statement. This is what `--master-data` is for. Once the file is imported into a slave, it will know at which point in time (well, rather at which binlog position) this replication needs to start. Everything that happened on the master after that needs to be replicated.

Finally, we configure that slave to use the credentials set up on the master before and then start the replication. Notice that the `CHANGE MASTER TO` statement used does not include the information about the log positions or file names because that has already been taken from the dump file just read in.

From here on the slave will go ahead and record all SQL statements sent from the master, store them in its relay logs, and then execute them against the local data set.



This recipe is very important because the following recipes are based on this! So in case you have not fully understood the above steps yet, we recommend you go through them again, before trying out more complicated setups.

## See also

- ▶ [Avoiding duplicate server IDs](#)

*Replication* —

## Setting up automatically updated slaves of a selection of tables based on a SQL dump

Often you might not need to replicate everything, but only a subset of tables in a database. MySQL allows exercising fine-grained control over what to replicate and what to ignore. Unfortunately, the configuration settings are not as obvious as they might seem at first.

In this recipe, you will see how to replicate only a few select tables from a database.

### Getting ready

The setup for this recipe is the same as for the previous one, *Setting up automatically updated slaves of a server based on a SQL dump*. Only the configuration options on the slave need to be changed. So instead of repeating everything here, we just present the important differences.

### How to do it...

- Follow the steps of the previous recipe up to the point where the `mysqldump` command is used to extract the initial data set from the master. Use this command instead:

```
$ mysqldump -uUSER -pPASS --master-data --single-transaction sakila address country city > sakila_master.sql
```

- Go on with the steps of the previous recipe up to the point where it tells you to edit the slave machine's configuration. Edit the slave's configuration as follows instead in the `[mysqld]` section:

```
server-id=1001
replicate-wild-ignore-table=sakila.%
replicate-do-table=sakila.address
replicate-do-table=sakila.country
replicate-do-table=sakila.city
```

- Continue with the rest of the instructions as in the *Setting up automatically updated slaves of a server based on a SQL dump* recipe.

## How it works...

The SQL dump file taken on the master is limited to three tables: address, country, and city. The slave's configuration also tells it to only execute statements coming from the master that targets one of these three tables (`replicate-do-table` directives), overtly ignoring any other changes in the `sakila` database (`replicate-wild-ignore-table`). Even though all other statements are still retrieved from the master and stored temporarily in the relay log files on the slave, only those with modifications to one of the tables explicitly configured are actually run. The rest are discarded.

You can choose any subset of tables, but you need to make sure to take Foreign key relationships between tables into account. In this example, the `address` table has a reference to the `city` table via the `city_id` column, while `city` in turn has a reference to `country`. If you were to exclude either one of the latter and your storage engine on the slave was InnoDB, replication would break because of Foreign key violations whenever you insert an address, since its dependencies were not fulfilled.

MySQL does not help you in this respect; you must make sure to identify all table relationships manually before setting up the replication.

## There's more...

In this example, we clearly specified three tables by their full names. There are more options available, not only to include but also to exclude tables. See the MySQL online manual chapter 16.1.3.3 on *Replication Slave Options and Variables* for more information at [http://dev.mysql.com/doc/refman/5.1/en/replication-options-slave.html#option\\_replication\\_slave-replicate-table](http://dev.mysql.com/doc/refman/5.1/en/replication-options-slave.html#option_replication_slave-replicate-table).

## Setting up automatically updated slave using data file copy

Even though replication is designed to keep your data in sync, circumstances might force you to set up slaves afresh. One such scenario might be severely changing the master, making replication too expensive. Using a SQL dump to re-initialize the slaves might be time-consuming, depending on the size of the data set and the power of the slave.

In cases where master and slave databases are the same size anyway (meaning, you don't have filters in place to sync data only partially) and if you can afford a downtime of the database, there is another way of providing slaves with a fresh starting point: copy the master's data files to the slave.



Beware that this approach will lose all data that was changed since the last backup.

*Replication***Getting ready**

To follow along with this recipe you will need privileges to shut down both master and slave MySQL instances and access the data and log directories on both machines. Depending on the size of your database you will have to judge which method of copying will be the best between the machines. If both are part of a local area network, copying via a shared folder or something like FTP will probably be the fastest way. You might, however, need to use other means of data transfer like external hard disks or the like, when only limited bandwidth is available.

Moreover, you will need administrative MySQL user accounts on both sides to execute the necessary statements to control replication.

**How to do it...**

1. Open the master's configuration file with a text editor. Locate the line containing the name of the binlog files. It is located in the [mysqld] section and will look similar to this:

```
[mysql]
...
log-bin=master-bin
...
```

2. Change the value of the log-bin parameter to a different name. In this example, we will use log-bin=new-master-bin because the master MySQL server to start with a new sequence of binlogs upon launch, making a convenient starting point for the replication.

3. Shut down the master MySQL instance.
4. Navigate to the MySQL data directory. The exact location can be found in the master's configuration file. Make sure to find both InnoDB data and log locations.
5. Optionally, copy data and log files to another location locally on the master. This may be faster than copying via network or USB drives, and allows for a quick recovery if the master fails. If you do this, use this temporary location to copy the data in the next step.
6. Copy the data to the slave machine. We recommend a temporary target location on the slave because this allows the slave to continue running for the time the copy is taking. Unless you want to bring along all the user accounts and privilege information from the master to the slaves, you should exclude the mysql folder from the data directory. You also need not copy the binary logs (in this example called master-bin.\*).
7. Stop the master MySQL instance.
8. You can restart the master once the original data and transaction log files have been copied to the slave.

- 
9. the names and sizes of InnoDB data and log files you copied to the slave. have to be entered into the slave's configuration because, otherwise, InnoDB start up. Also, pay attention to an autoextend option, which the last of the might have attached. Make sure you carry over this option, should it be the master. You can also take these values from the master's configuration file.
10. Replace the original slave data and log files with those of the master. Make sure you keep the mysql database directory if you decided not to copy it from the master.
11. Make sure you delete the master.info and any relay-logs from the slave—those do not match the current state and would cause trouble when the slave is restarted.
12. Edit the slave's configuration file to match the names and sizes of the data files you wrote down a minute ago.
13. Start the slave server again. It should come up without any problems using the data files. If not, make sure you got everything right in the config file regarding the names and sizes of the data files.
14. Re-initialize the replication connection. This is rather easy because we altered the master's configuration to log all binary logs that occurred after the snapshot copy was taken to a new series of binlog files using the appropriate host name, user, and password values for your master:
- ```
slave> CHANGE MASTER TO MASTER_HOST='master', MASTER_USER='replica',  
MASTER_PASSWORD='slavepass', MASTER_LOG_FILE='new-master-bin.000001';  
slave> START SLAVE;
```
- As we want the slave to start reading the new-master-bin.000001 file from the beginning, no MASTER\_LOG\_POS has to be specified.

15. Verify whether the slave is running correctly.
- ```
slave> SHOW SLAVE STATUS\G  
***** 1. row *****  
...  
Slave_IO_Running: Yes  
Slave_SQL_Running: Yes  
...
```

*Replication***How it works...**

The principle of this recipe is very simple: replication needs a common starting point for master and slave. What could be better than a 1:1 copy of the original master's data? If the master is shut down during the process, no more writes can happen. Configuring the master to switch with a new binlog file on its next start makes it trivial to point the slave to the right position because it is right at the new file's beginning.

If you cannot change the master binlogs' file names, the process is slightly more complex. First you need to make sure nobody can modify any data for a short period of time. You can do this with a `FLUSH TABLES WITH READ LOCK;` statement. Then issue a `SHOW MASTER STATUS` command and note the values. Now, without closing the client connection or releasing the lock, shut down the master server. Only if the lock is kept while shutting down the master can you be sure no write operations take place and invalidate the binlog position you just gathered.

Copy the data and transaction log files as described above. The remaining steps are the same, except of course, when you issue the `CHANGE MASTER TO` on the slave. Here you will need to insert the `MASTER_LOG_FILE` and `MASTER_LOG_POS` you got from `SHOW MASTER STATUS`.

**There's more...**

The steps described above require you to take down both master and slave databases, not necessarily at the same time. Nevertheless, this might not be an option if you are working with a production system that cannot be shut down easily.

In these cases, you have some other options that are, however, not explained in detail here.

**Conserving data file by using LVM snapshots**

If your data and log files are stored on a logical volume managed by LVM, you can use the snapshot feature to conserve the data files' state once you have got the `SHOW MASTER STATUS` information. As soon as the snapshot has been taken, you can release the lock and proceed as described above, copying not the most current version but the snapshot. Be advised, however, that this approach might take a significant hit on the I/O performance of your master!

**Backing up data using Percona xtrabackup**

At the time of writing, an open-source alternative to the commercially available *innobackupex* tool (available from <http://www.innodb.com/products/hot-backup/>) is under development. While being primarily a backup tool that allows backing up InnoDB data while the server is up and running, the documentation contains a (currently empty) section on setting up a slave from a backup in replication. Experience tells that Percona—the company behind the tool—will add this functionality in the future.

---

behind xtrabackup—is very engaged in the MySQL ecosystem and might very well have completed its set of instructions by the time you read this. To check on the current status of the project go to <https://launchpad.net/percona-xtrabackup>.

## Sharing read load across multiple machines

Often you have a very unequal distribution of read and write operations on a database. Websites usually get many more visitors just browsing and reading contents than adding contributing contents. This results in the database server being mainly busy reading information instead of adding or modifying existing material.

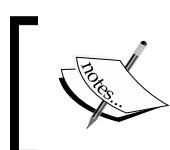
Replication can be used to alleviate scalability issues when your site reaches a certain point and a single machine might reach the limits of its performance reserves.

Unfortunately, MySQL does not offer this load-balancing functionality itself, so you will have to take appropriate actions on the application level.

In this recipe, we will show you the general procedure to follow when sharing read load between two slave machines while still aiming writes at the master. Beware that due to the asynchronous nature of MySQL replication, your application must be able to handle out-of-date results because issuing an `INSERT`, `UPDATE`, or `DELETE` against the master does not mean that you can read the modified data back immediately as the slave might need some time to catch up. Usually, on a local network this should be a couple of seconds at most, nevertheless the application code must be ready for that.

To simplify the scheme, you should design your application to exclusively read from the slaves and only use the master for modifications. This brings the additional benefit of being able to keep the overall system up and running while switching to a read-only mode temporarily backing up the master server. This is not part of this recipe, however.

The example used in this recipe uses three database servers. The sample application is written in Java, using the *MySQL Connector/J* JDBC database driver. Depending on the application platform you are using, syntax and function names will differ, but the general principle is language independent.



The source code shown later has been abbreviated to show only the most relevant portions. You can find the complete file on the book's website.

*Replication* —**Getting ready**

Depending on your application infrastructure, you will need privileges to change its database connectivity configuration and the source code. This is usually a task that requires cooperation with application developers.

To follow along with this example you should be familiar with the Java language and basic constructs.

Moreover, you will need three MySQL servers—one configured as the master and two slaves. They will be referred to as *master*, *slave1*, and *slave2* in this example. Substitute concrete host names appropriately.

You will also need the Java Standard Edition development tools available from <http://java.sun.com>, and the MySQL Connector/JDBC driver available from <http://dev.mysql.com>. Download and install both if you do not already have them.

**How to do it...**

1. Download the file called `MySQLBalancingDemo.java` from the book's website. It contains the following code:

...

```
Connection conn = driver.connect("jdbc:mysql://master:3306,slave1:3307,slave2:3308/sakila?user=testuser&password=testpass&LoadBalance=true", null);

conn.setReadOnly(false); // target the MASTER

rs = conn.createStatement().executeQuery(
    "SELECT @@server_id;" );
rs.next();
System.out.println("Master: " + rs.getString(1));

conn.setReadOnly(true); // switch to one of the slaves

rs = conn.createStatement().executeQuery(
    "SELECT @@server_id;" );
rs.next();
System.out.println("Slave: " + rs.getString(1));
conn.close();
```

- 
2. Compile the file using the `javac` compiler. Alternatively, an integrated development environment like Eclipse or Netbeans can take care of this for you:

```
$ javac -cp mysql-connector-java-5.1.7-bin.jar MySQLBalance.java
```

3. Run the sample application and see how it automatically distributes the requests between the two slaves:

```
$ java -cp .:mysql-connector-java-5.1.7-bin.jar MySQLBalance
Master: 1000
Slave: 13308
```

## How it works...

You just compiled and ran a small program that demonstrates round-robin load balancing.

The first line of output is the master's `server-ID` setting, because the first connection was *not* set to read only. The connection is then declared to be targeted at the slave using `setReadOnly(true)`. The next query will then return the server ID of the part of the cluster it was balanced to. You might need to run the demo a few times to see a different result used because the algorithm that balances the load does not strictly toggle each time; it might direct a few connections against the same slave.

## There's more...

While the JDBC driver makes it relatively easy to use read load balancing across several slaves, it only helps you take the first step on the way. You must take care that the application knows which connection to use for write operations and which for read. It must also deal with slaves and master possibly being slightly out of sync all the time. Concentrating this special logic in a class of its own, is advisable to limit the effect on the rest of the application.

## Working with connection pools

When working with connection pooling, be sure to initialize any connection you get from the pool in the correct mode using the `setReadOnly()` method, to be sure you know what state it is in. You might be handed a connection that was set to the wrong mode when it was put into the pool.

## Working on other programming environments

In development environments not using Java, you might have to take care of managing the connection cycling between slaves yourself. Independent of the actual language or environment you are using, a good practice is to channel all database operations through a set of functions.

## *Replication*

INSERT, UPDATE, and DELETE operations, always connecting to the master and a function going to the slaves for data reads.

In case you need to select something back that you just wrote and cannot allow for replication lag, you might also provide a read function querying the master machine. You should use this sparingly, however, because it definitely counteracts the intention of the master from the read load.

### **Considering efficiency while adding slaves**

Of course, the slaves have to perform write operations as well to keep up with the means their performance is not fully available for reads, limiting scalability.

So adding more slaves does not proportionally improve performance of the overall

## **Using replication to provide full-text indexing for InnoDB tables**

The InnoDB storage engine is the one most commonly used nowadays because it more enterprise-level features than MyISAM and most other engines. However, InnoDB do have a major drawback: they do not support full-text indexing. This can be a significant obstacle when you have to design any sort of application that relies on atomic operations and must store text data in a searchable manner.

While there are third-party products available to redress this shortcoming, there are may need to refrain from using these and stick to the out-of-the-box functionality. If willing to provide an additional server and make slight adjustments to your application replication can help you provide a full-text index for InnoDB tables indirectly.

This recipe is similar to the one about *Sharing read load across multiple machines* chapter. In contrast, only queries that are targeted at the full-text index need to be run on a slave machine. This will require slight changes to the application code.

### **Getting ready**

To follow along with this recipe, you will need two MySQL servers available—a master and a slave. For testing, these might reside on the same physical machine. In a production environment we do, however, recommend two separate pieces of equipment.

They will be referred to as *master* and *slave* in this example. Substitute your concrete names appropriately.

You will need privileges to change the application source code. This is usually a task

## How to do it...

- On the master, identify the table to which you want to add a full-text index. In this example, we will use the following table definition from a fictional forum application of some sort:

```
CREATE TABLE `posts` (
    `id` int(11) NOT NULL auto_increment,
    `title` varchar(100) NOT NULL,
    `posttext` text NOT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

The `posttext` column contains the text of forum posts. As the table is created with `ENGINE=InnoDB`, we cannot add a full-text index to it.

- On the slave, create the same table, but with a slightly modified definition:

```
CREATE TABLE `posts` (
    `id` int(11) NOT NULL auto_increment,
    `title` varchar(100) NOT NULL,
    `posttext` text NOT NULL,
    PRIMARY KEY (`id`),
    FULLTEXT KEY `FT_text` (`posttext`)
) ENGINE=MyISAM;
```

The storage engine is set to MyISAM, allowing the `FULLTEXT KEY `FT_text` (`posttext`)` definition. Trying to add this on the master would result in an error message.

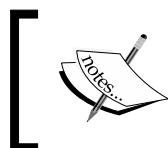
- Make sure the replication rules between master and slave include the slave's IP address.
- application to access the slave when doing full-text queries. It is generally a good practice to concentrate all database access to a dedicated module or application, so that you can easily modify your application's interaction with the underlying database.

## How it works...

In this replication setup, whenever you make changes to the master's `posts` table, those changes will be replicated to the slave, but the target table uses a different storage engine than the master. As SBR simply sends over SQL statements without any information about the origin, the slave will execute the instructions blindly. While this can be a problem in other circumstances because it makes the whole process somewhat fragile, it plays to our advantage here.

## Replication

Upon UPDATE or INSERT to the posttext column the MyISAM engine will update the full-text index appropriately. This enables the application to issue queries using the standard MySQL query syntax against the slave.



An important drawback you must take into account is that you cannot JOIN tables between different MySQL servers!

A workaround is required when you have to, for example, join the posts with a users table via the posts.id column. To implement this you will need to issue two separate queries. The first one using the full-text search on the slave will bring up all posts containing the search terms. From the resulting rows you can then take the id column values and use them in a second query against the master database, substituting the text search with an id.

## There's more...

MyISAM's full-text index has existed for several years, but has not been improved at all over time. If you have many concurrent requests you will notice significant resource usage, limiting scalability.

Over the past few years, several third-party vendors have stepped up with alternatives to the problem of full-text search, offering more features and better performance.

One of those products, offering tight integration with MySQL and PHP, is *Sphinx*—an open-source product available for free from <http://www.sphinxsearch.com>. If you find that the built-in capabilities are either too slow or too limited in other respects to meet your application requirements, you should definitely have a look at it.

## Setting up new slaves in this scenario

You should not simply use a regular SQL dump to initialize the slave, as it will contain a CREATE TABLE statement that specifies InnoDB and does not include the full-text index. Of course, you could change the table type after the import is complete. However, this can be time-consuming. Instead, we recommend you first create the target schema on the slave and make sure the tables in question are created with ENGINE=MyISAM.

Then go ahead and import the data into the table. Only after that, add the full-text index. This is typically much faster than having the index in place beforehand because MySQL will then update it all the way through the bulk insert of rows. This is a very expensive operation compared to the delayed index creation.

## See also

- ▶ *Adding a full-text index* in Chapter 2

# Estimating network and slave I/O load

Especially when using replication over a wide-area network connection with limited bandwidth, it is interesting to be able to predict the amount of data that needs to be transported between master and slaves.

While MySQL does not use the most efficient strategy to deliver data, it is at least relatively easy to calculate the requirements in advance.

This is less of a step-by-step recipe than an annotated walkthrough of the basic formula, but it can be used to estimate the traffic you will have to be prepared for.

## Getting ready

In order to follow along, you must have some key data points available because otherwise there is not much to calculate. You will need:

- ▶ The number of slaves (to be) connected to the master.
- ▶ An idea about the average amount of binlogs written when using the master under regular load. Knowing about peak times can be interesting as well.
- ▶ The bandwidth of the connection between master and slaves. This includes the bandwidth of the network interfaces on the master and, in general, the whole route between them (possibly including Internet connections).

We assume that there are no other network-intensive applications running on the slaves, so that practically all the speed your network card can provide is usable for replication.

In this example, we will keep matters simple, assuming the following:

Data point	Value
Master's Connectivity	Gigabit LAN interface (approx. 100MB/s)
Slaves' Connectivity	2MBit/s DSL line, of which 1MBit/s is assumed available for MySQL. 1MBit/s is equivalent to 125MB/s, which is equivalent to 125,000kb/s, or approximately 100kb/s.
Average amount of binlogs created on master	175MB per hour, approx. 50kb/s.
Number of Slaves	5
Speed of the slowest link in the connection	Master is connected to the Internet via a 2MBit/s line, which is equivalent to 250,000kb/s, or 250kb/s.

*Replication***How to do it...**

1. **Check the master's network speed:** number of slaves with the average amount of binlogs:  $5 \times 175\text{MB}/\text{hour} = 875\text{KB}/\text{second}$  or about 250kb/second. The gigabit connection can handle this easily.
2. **Check individual slaves' network speed:** The 1MB/s DSL line is sufficient for an average amount of data of 50kb/second. Data production is not linear over time—there might be peaks, but there is still plenty of bandwidth available.
3. **Check if the slowest part of the route between master and slaves is the load:** 250kb/second should be no problem for the 10MBit/second Internet connection.
4. Disk I/O load on each slave, caused by the replication process. This is equivalent to the amount of data the master produces. Provided the slave's I/O is not already saturated by other processes, additional 175MB per hour should not pose a problem either.

**How it works...**

Basically, replication simply needs sufficient resources to copy the master's binlogs to the slaves. This really all there is to it. Depending on the network route between them, this can be easily done (say most LANs), or can be tricky (as in cases with slow Internet connections).

In this example, we see there should be no problem on any part of the system, as there is still room for a higher load on each resource. The most limiting factor in this scenario is likely to be the master's outgoing Internet connection. If you add more slaves to the scenario, each new one will add another 50KB per second of outgoing bandwidth. Assuming replication slaves can use the full 1MB/s outgoing speed, which is not very realistic, that part of the route would theoretically service 20 slaves at most. In reality, it will be more like 10 to 15.

**There's more...**

There are two more general considerations you might want to think about when planning a replication setup.

**Handling intermittent connectivity between master and slaves**

If the connection between master and slaves is only available for a limited period of time, the slaves will start to lag behind while disconnected. The slaves will download new data as quickly as possible when the connection is resumed and store it locally in the relay log, then asynchronously executing the statements. So expect higher network load during the connection interruptions.

You will also want to take that into account when there are multiple slaves trying to connect to the master simultaneously.

---

## Enabling compression with the `slave_compressed_protocol` option

Particularly useful for low bandwidth connections between master and slaves is the compression feature for replication traffic. Provided it is switched on on both master slave machines, it can significantly reduce the amount of data that is actually transferred over the network at the cost of increased CPU loads. The master will then send out information in a compressed format.

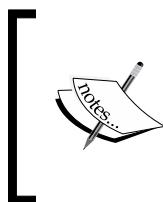
In a simple comparison, measuring the network traffic while creating and loading the sample database, 3180kb of binlogs were created. However, with the compressed option switched on, only about 700KB of data per slave were sent over the network.

To enable compression, add the following line to the `[mysqld]` section in the configuration files on both master and slave:

```
slave_compressed_protocol=1
```

Then restart the servers to enable the feature. Verify whether it was switched on successfully by issuing a `SHOW VARIABLES LIKE 'slave_compressed%';` command on both master and slaves.

You can achieve a similar effect with SSH compression. As we generally do not replicate over the Internet without encryption, that option might even be more appropriate in such scenarios as it does not require any configuration changes to MySQL.



Naturally, the level of compression heavily depends on the data you are handling. If you store JPEG images in BLOB fields, for example, those cannot be compressed much more than they already are!

### See also

- ▶ *Encrypting a MySQL server connection with SSH in Chapter 3*
- ▶ *Creating an encrypted MySQL console via SSH in Chapter 3*
- ▶ *Using a PuTTY template connection for SSH secured connections in Chapter 3*

*Replication* —

## Limiting network and slave I/O load in write scenarios using the blackhole storage engine

If you have a large number of slaves and a rather busy master machine, the network load can become significant, even when using compression. This is because all statements written to the binlog are transferred to all the slaves. They put them in their relay logs and asynchronously process them.

The main reason for the heavy network load is the *filter on the slave paradigm* that MySQL employs. Everything is sent to every one of the slaves and each one decides which statements to throw away and which to apply based on its particular configuration. In the worst case, a slave has to transmit every single change to a database to replicate only a single table.

### Getting ready

The following procedure is based on Linux. So in order to repeat it on Windows, you will need to adapt the path names and a little shell syntax accordingly.

To follow along, you will need a MySQL daemon with the *blackhole* storage engine installed. Verify this with the following command:

```
mysql> show variables like '%blackhole%';
```

Variable_name	Value
have_blackhole_engine	YES

Even though you only strictly need a blackhole-enabled MySQL server on the actual instance, for this example we will be using only a single machine and just a single MySQL version, but with different configuration files and data directories.

In the following steps, we assume you have installed a copy of MySQL in a folder called `blacktest` in your home directory. Modify accordingly if your setup differs.

## How to do it...

1. Create three distinct data directories—one for the master, one for the slave engine, and one for a slave.

```
~/blacktest$ mkdir data.master
~/blacktest$ mkdir data.slave
~/blacktest$ mkdir data.black
```

2. Into each of those, copy the MySQL account files.

Ideally, you should take an empty one from a freshly downloaded distribution, and make sure you do not accidentally copy users you do not want.

```
~/blacktest$ cp -R data/mysql data.master
~/blacktest$ cp -R data/mysql data.slave
~/blacktest$ cp -R data/mysql data.black
```

3. Configure the master instance. To do so, create a configuration file called my.cnf and make sure that it contains the following settings:

```
[client]
port      = 3307
socket    = /home/ds/blacktest/master.sock

[mysqld_safe]
socket    = /home/ds/blacktest/master.sock

[mysqld]
user      = mysql
pid-file = /home/ds/blacktest/master.pid
socket    = /home/ds/blacktest/master.sock
port      = 3307
basedir   = /home/ds/blacktest
datadir   = /home/ds/blacktest/data.master
tmpdir    = /tmp
language  = /home/ds/blacktest/share/mysql/english

bind-address = 127.0.0.1

server-id   = 1
log-bin     = /home/ds/blacktest/master-bin.log
```

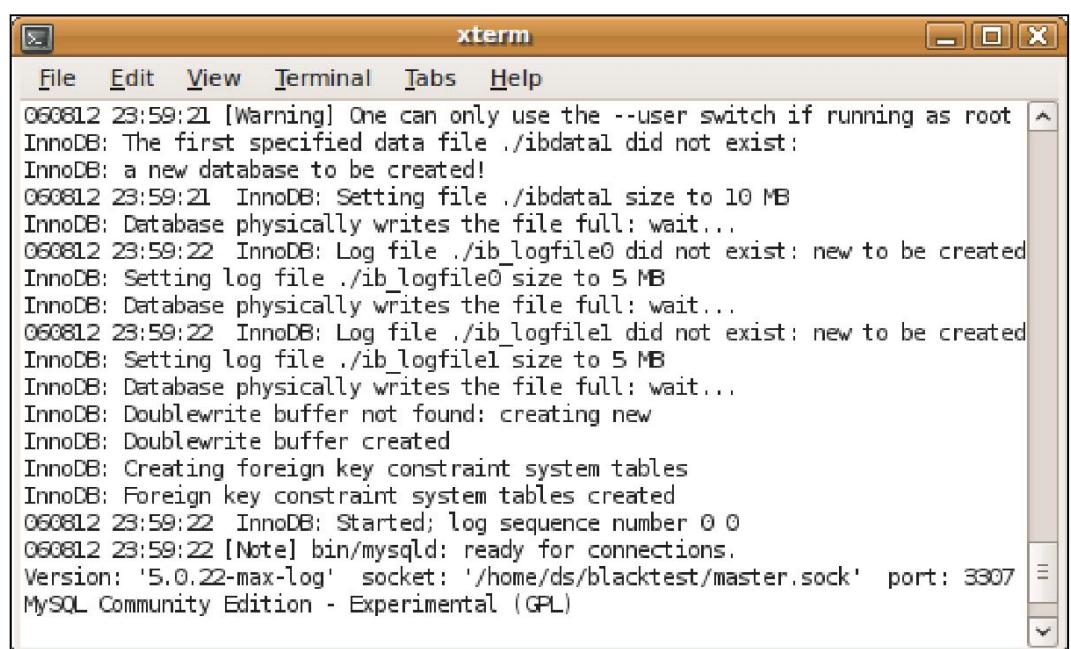
Everything that is specific to the master instance has been highlighted—the

*Replication*

4. Start the master daemon for the first time to make sure everything works. I recommend a dedicated window for this daemon. For example:

```
~/blacktest$ xterm -T MASTER -e bin/mysqld \
>           --defaults-file=my.master \
>           --console &
```

This will start the daemon in the background and show its output in a new window.



The screenshot shows an xterm window titled "xterm" with the following log output:

```
File Edit View Terminal Tabs Help
060612 23:59:21 [Warning] One can only use the --user switch if running as root
InnoDB: The first specified data file ./ibdata1 did not exist:
InnoDB: a new database to be created!
060612 23:59:21 InnoDB: Setting file ./ibdata1 size to 10 MB
InnoDB: Database physically writes the file full: wait...
060612 23:59:22 InnoDB: Log file ./ib_logfile0 did not exist: new to be created
InnoDB: Setting log file ./ib_logfile0 size to 5 MB
InnoDB: Database physically writes the file full: wait...
060612 23:59:22 InnoDB: Log file ./ib_logfile1 did not exist: new to be created
InnoDB: Setting log file ./ib_logfile1 size to 5 MB
InnoDB: Database physically writes the file full: wait...
InnoDB: Doublewrite buffer not found: creating new
InnoDB: Doublewrite buffer created
InnoDB: Creating foreign key constraint system tables
InnoDB: Foreign key constraint system tables created
060612 23:59:22 InnoDB: Started; log sequence number 0 0
060612 23:59:22 [Note] bin/mysqld: ready for connections.
Version: '5.0.22-max-log' socket: '/home/ds/blacktest/master.sock' port: 3307
MySQL Community Edition - Experimental (GPL)
```

The warning about the --user switch can be ignored for now. Should you get a message very similar to the one above (especially concerning the **reconnections** part) go back and find the error in your setup before going on. The error messages issued by MySQL are rather verbose and bring you back pretty soon.

5. Insert some test data to be able to verify the correct function of the filter later. Connect to the master instance just started and create some tables and data.

```
~/blacktest$ bin/mysql -uroot -S master.sock --prompt='master>'
```

```
master> CREATE DATABASE repdb;
master> USE repdb;
master> CREATE TABLE tblA (
    ->         id INT(10) PRIMARY KEY NOT NULL,
    ->         label VARCHAR(30)
    ->     ) ENGINE=InnoDB;
master> CREATE TABLE tblB (
    ->         name VARCHAR(20) PRIMARY KEY NOT NULL,
```

---

```

master> INSERT INTO tblA VALUES
      ->      (1, 'label 1'),
      ->      (2, 'label 2'),
      ->      (3, 'label 3');

master> INSERT INTO tblB VALUES
      ->      ('Peter', 55),
      ->      ('Paul', 43),
      ->      ('Mary', 25);

```

Inserting this data already creates binlog information. You can easily verify looking at the file system. The master-bin.000001 file should have grown around 850 bytes now. This might vary slightly if you did not enter the commands above with the exact same number of spaces—the binlog will store commands exactly as you typed them. For example, we will only replicate changes to the master but ignore anything that happens to table tblA. We will assume that tblB will be written by an application on the slave. So the table should be present, and the slave should ignore it, on the slaves to avoid key collisions.

## 6. Create a user account on the master for the filter to connect to.

```

master> GRANT REPLICATION SLAVE
      -> ON *.* 
      -> TO 'repblack'@'localhost'
      -> IDENTIFIED BY 'blackpass';

```

## 7. Configure the filter (blackhole) instance with a configuration file named my.cnf. It contains at least the following :

```

[client]
port          = 3308
socket        = /home/ds/blacktest/black.sock

[mysqld_safe]
socket        = /home/ds/blacktest/black.sock

[mysqld]
log-slave-updates
skip-innodb
default-storage-engine=blackhole

user          = mysql
pid-file     = /home/ds/blacktest/black.pid
socket        = /home/ds/blacktest/black.sock

```

## Replication

```

datadir          = /home/ds/blacktest/data.black
tmpdir           = /tmp
language         = /home/ds/blacktest/share/mysql/english

bind-address     = 127.0.0.1

server-id        = 2
log-bin           = /home/ds/blacktest/black-bin.log
relay-log         = /home/ds/blacktest/black-relay.log

```

Notice that all occurrences of *master* have been replaced with *black!*

Moreover, the `server-id` setting has been changed and the `log-slave-updates`, `skip-innodb`, and `default-storage-engine` options have been added. The second one prevents this instance from creating `ibdata` table space files, which would not be used later anyway. The last one specifies which storage engine to use when a `CREATE TABLE` statement does not explicitly specify one or if the specified engine is not available. We will come back to this soon.

8. Make sure this instance basically works by starting it the same way as before (you will not see the InnoDB messages here, of course).

```

~/blacktest$ xterm -T BLACK -e bin/mysqld \
>           --defaults-file=my.black \
>           --console &

```

9. Create a set of dump files from to set up both the blackhole filter and an example slave. The details on what the two and in which ways they are different will be explained later. Use these to create the files needed:

```

~/blacktest$ bin/mysqldump -S master.sock -uroot \
>           --master-data \
>           --single-transaction \
>           --no-create-info \
>           --ignore-table=repdb.tblA \
>           repdb > master_data.sql

```

```

~/blacktest$ bin/mysqldump -S master.sock -uroot \
>           --no-data \
>           repdb > master_struct.sql

```

- 
10. Connect to the filter server, create the database, make it the default database finally, import the structure information created before:

```
~/blacktest$ bin/mysql -uroot -S black.sock --prompt='bla'
```

```
black> CREATE DATABASE repdb;
black> USE repdb;
black> source master_black.sql;
```

At this point we now have the structure of the master transferred to the filter adapted to use the blackhole engine for all the tables.

11. Set up the replication between master and filter engine. We need to know the exact position from where the filter will start replicating information from the previously taken data dump like this:

```
~/blacktest$ head -n 30 master_data.sql | grep 'CHANGE MA'
```

Write down that information; we will need it in a moment.

12. Modify the `my.black` configuration file to contain the following in the [mysqld] section:

```
replicate-ignore-table=repdb.tblA
replicate-do-table=repdb.tblB
```

This is a very simple filter setup; in a real application scenario these rules probably be more complex.

13. Restart the filter engine to activate the new configuration.

```
~/blacktest$ bin/mysqladmin -uroot -S black.sock shutdown
```

```
~/blacktest$ xterm -T BLACK -e bin/mysqld \
>           --defaults-file=my.black \
>           --console &
```

14. Reconnect the client connected to the blackhole engine. To do this enter a `SELECT 1;` command.

15. Execute the following command to hook up the filter to the master. Be sure to enter the same values you wrote down a moment ago in the statement:

```
black> CHANGE MASTER TO
      -> master_host='localhost',
      -> master_port=3307,
      -> master_user='repblack',
      -> master_password='blackpass',
```

*Replication* —

16. Retrieve information required to set up the filter/slave portion. Write down of the SHOW MASTER STATUS command, they will be needed later:

```
black> FLUSH LOGS;
black> SHOW MASTER STATUS;
+-----+-----+-----+
| File | Position | ... | ... |
+-----+-----+-----+
| black-bin.000003 | 98 | | |
+-----+-----+-----+
```

17. Start the slave thread on the filter engine and verify that everything is working correctly.

```
black> START SLAVE;
black> SHOW SLAVE STATUS \G
***** 1. row *****
Slave_IO_State: Waiting for master to send event
Master_Host: localhost
Master_User: repblack
Master_Port: 3307
Connect_Retry: 60
Master_Log_File: master-bin.000001
Read_Master_Log_Pos: 1074
Relay_Log_File: black-relay.000003
Relay_Log_Pos: 236
Relay_Master_Log_File: master-bin.000001
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
...
Replicate_Do_Table: repdb.tblB
Replicate_Ignore_Table: repdb.tblA
...
Last_Error:
Skip_Counter: 0
Exec_Master_Log_Pos: 1074
Relay_Log_Space: 236
...
Seconds_Behind_Master: 0
```

At this point we have successfully established a replication connection between the master database and the blackhole-based filter instance.

18. Check that nothing has yet been written to the filter's binlogs. Because we issued a FLUSH LOGS command on the filter instance, there should be nothing in the most recent binlog file. Verify this as follows:

```
~/blacktest$ bin/mysqlbinlog black-bin.000003
```

```
ds@ubuntu810: ~/blacktest
File Edit View Terminal Tabs Help
~/blacktest$ bin/mysqlbinlog black-bin.000003
/*!40019 SET @@session.max_insert_delayed_threads=0*/;
/*!50003 SET @OLD_COMPLETION_TYPE=@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
# at 4
#060813 0:44:06 server id 2 end_log_pos 98      Start: binlog v 4, server v 5.0.
22-max-log created 060813 0:44:06
# Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
# End of log file
ROLLBACK /* added by mysqlbinlog */;
/*!50003 SET COMPLETION_TYPE=@OLD_COMPLETION_TYPE*/;

ds@ubuntu810:~/blacktest$
```

19. Test the filter setup with some statements issued on the master.

```
master> UPDATE repdb.tblA
      -> SET label='modified label 3'
      -> WHERE id=3;
master> INSERT INTO repdb.tblB
      -> VALUES ('John', 39);
```

We would expect to see the INSERT in the binlog file of the filter instance, but not the UPDATE statement, because it modifies `tblA`, which is to be ignored.

20. Verify that the rules work as expected by having another look at the filter instance's binlog.

```
~/blacktest$ bin/mysqlbinlog black-bin.000003
```

<img alt="Screenshot of a terminal window titled 'ds@ubuntu810: ~/blacktest'. The window shows the output of the command 'bin/mysqlbinlog black-bin.000003'. The output is as follows:
 - /\*!40019 SET @@session.max\_insert\_delayed\_threads=0\*/;
 - /\*!50003 SET @OLD\_COMPLETION\_TYPE=@COMPLETION\_TYPE,COMPLETION\_TYPE=0\*/;
 - # at 4
 - #060813 0:44:06 server id 2 end\_log\_pos 98 Start: binlog v 4, server v 5.0.
 - 22-max-log created 060813 0:44:06
 - # Warning: this binlog was not closed properly. Most probably mysqld crashed writing it.
 - # at 98
 - #060813 0:46:30 server id 1 end\_log\_pos 204 Query thread\_id=1 exec\_time=0 error\_code=0
 - use repdb;
 - SET TIMESTAMP=1155422790;
 - SET @@session.foreign\_key\_checks=1, @@session.sql\_auto\_is\_null=1, @@session.unique\_checks=1;
 - SET @@session.sql\_mode=0;
 - /\*!\C latin1 \*/;
 - SET @@session.character\_set\_client=8,@@session.collation\_connection=8,@@session.collation\_server=8;
 - INSERT INTO repdb.tblB VALUES ('John', 39);
 - # End of log file
 - ROLLBACK /\* added by mysqlbinlog \*/;</pre>

*Replication*

This looks precisely as expected—the INSERT is present, the UPDATE is not to be seen.

21. Set up the configuration of a slave using the my.cnf file:

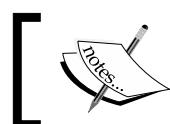
```
[client]
port      = 3309
socket    = /home/ds/blacktest/slave.sock

[mysqld_safe]
socket    = /home/ds/blacktest/slave.sock

[mysqld]
user      = mysql
pid-file = /home/ds/blacktest/slave.pid
socket    = /home/ds/blacktest/slave.sock
port      = 3309
basedir   = /home/ds/blacktest
datadir   = /home/ds/blacktest/data.slave
tmpdir    = /tmp
language  = /home/ds/blacktest/share/mysql/english

bind-address = 127.0.0.1

server-id   = 3
relay-log   = /home/ds/blacktest/slave-relay.log
```



Notice that all occurrences of *master* have been replaced with *slave*!

Again the `server-id` setting has been changed and the `log-slave-updates`, `skip-innodb`, and `default-storage-engine` options that were part of the master instance's configuration are *not* included. Also, the `log-bin` parameter has been removed because changes on the slave need not be recorded separately.

22. Start up the slave engine. You will see the familiar messages about InnoDB loading the data files and finally, the **Ready for connections** line:

```
~/blacktest$ xterm -T SLAVE -e bin/mysqld \
>           --defaults-file=my.slave \
>           --console &
```

- 
23. Then connect a client to the slave and create the database.

```
~/blacktest$ bin/mysql -uroot -S slave.sock --prompt='slave>'
```

```
slave> CREATE DATABASE repdb;
slave> USE repdb;
```

At this point, the slave is set up and has an empty repdb database.

24. Fill up the slave database with the initial snapshot of the master. We need to copy the files here. The details of why are explained further down in the *How it works* section.

```
slave> source master_struct.sql;
```

```
...
```

```
slave> source master_data.sql;
```

```
...
```

25. Verify that you can find the data from the master on the slave now by doing the following:
- \* FROM first table repdb.tblA and then repdb.tblB.

```
ds@ubuntu810: ~/slavetest
File Edit View Terminal Tabs Help
slave> SELECT * FROM repdb.tblA;
Empty set (0.00 sec)

slave> SELECT * FROM repdb.tblB;
+-----+
| name | age |
+-----+
| Mary | 25 |
| Paul | 43 |
| Peter | 55 |
+-----+
3 rows in set (0.00 sec)

slave>
```

The first SELECT shows no records because tblA was excluded from the replication. Table tblB contains the three records we inserted on the master.

26. Create a replication user account on the filter instance for the slave.

```
black> GRANT REPLICATION SLAVE
      -> ON *.* 
      -> TO 'repslave'@'localhost'
      -> IDENTIFIED BY 'slavepass';
```

27. Connect the slave to the filter engine. Be sure to insert the correct values for MASTER\_LOG\_FILE and MASTER\_LOG\_POS in the statement.

*Replication*

```
slave> CHANGE MASTER TO
      -> master_host='localhost',
      -> master_port=3308,
      -> master_user='repslave',
      -> master_password='slavepass',
      -> master_log_file='black-bin.000003',
      -> master_log_pos=98;
Query OK, 0 rows affected (0.01 sec)
```

28. Start the slave and verify that it starts

```
slave> START SLAVE
slave> SHOW SLAVE STATUS \G
***** 1. row *****
Slave_IO_State: Waiting for master to send event
...
Relay_Master_Log_File: black-bin.000003
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
...
Seconds_Behind_Master: 0
```

29. As soon as the previous step is complete, the replication should already have replicated the record for **tblB** on the slave and inserted the new (**"John"**, **39**) record. Verify it like this:

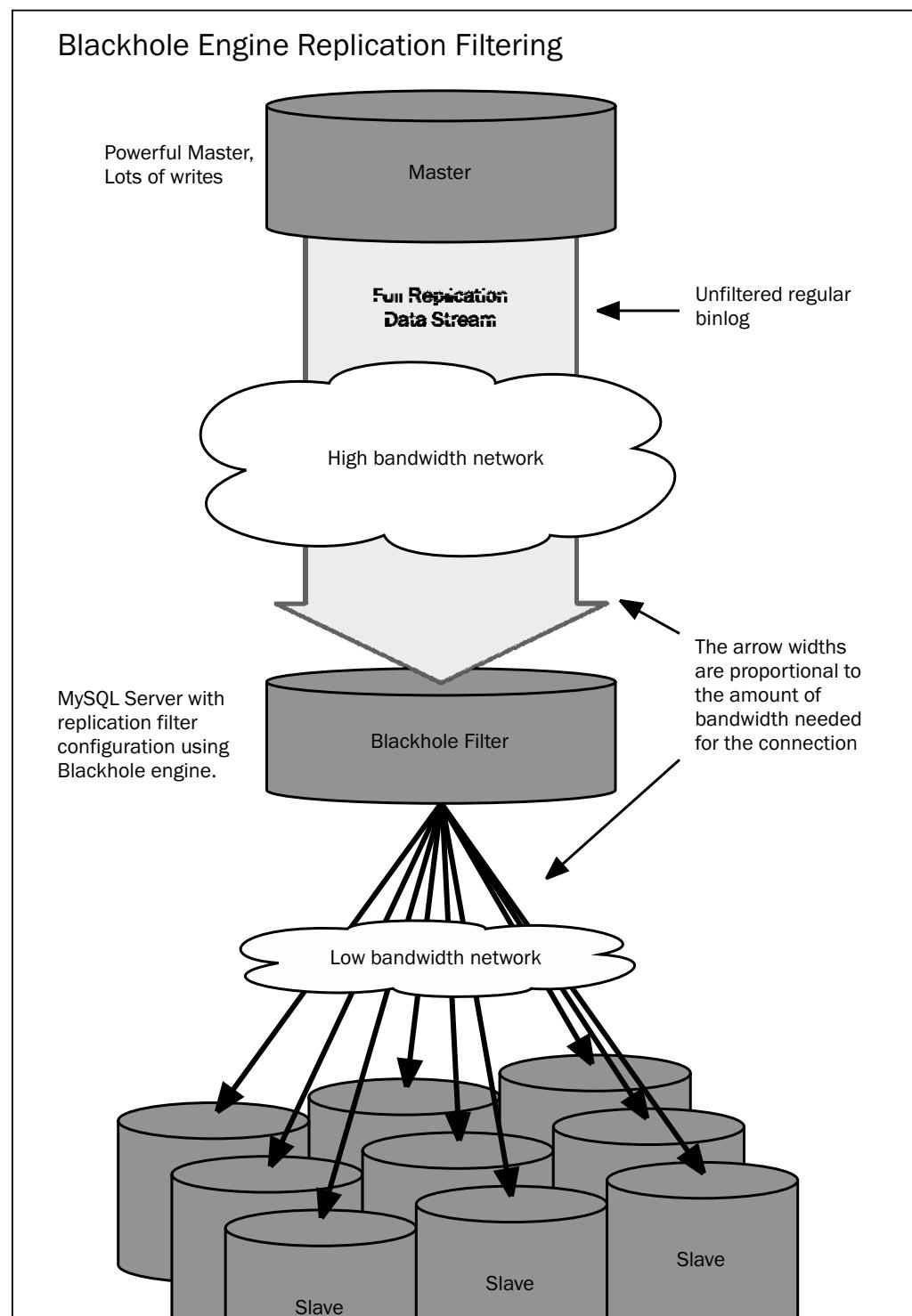
```
slave> SELECT * FROM repdb.tblB;
```

name	age
John	39
Mary	25
Paul	43
Peter	55

Apparently, the replication works. You can now try to modify some data on the master and check if the results match on the slave. Anything you do to modify **tblB** on the master should be reflected on the slave. Remember to use fully qualified statements; other changes will not match the replication rules.

## How it works...

Though MySQL did not implement a *filter on the master* feature literally, another way similar things was provided. While MyISAM and InnoDB implement ways of storing data on disk, another engine was created that is basically an empty shell. It just answers `INSERT`, `UPDATE`, or `DELETE` requests coming from the SQL layer above. `SELECT` always return an empty result set. This engine is suitably called the blackhole storage engine as everything you put into it just vanishes.



## *Replication*

In the upper part you see the main master server. All modifying statements are written into the master's binlog files and sent over the network to subscribed slaves. In this case, there is only a single slave: the filter server in the middle. The thick arrow in between them represents a large amount of data that is sent to it.

In the lower part of the picture, there are a number of slaves. In a regular setup, a single arrow would be drawn from the master to each of those—meaning that the same number of replication data would be sent over the network multiple times. In this picture, the filter server is configured to ignore statements for certain tables. It is also configured to write the statements received from a replication master to its own binlogs. This is done to prevent regular slaves from writing replicated statements to their own binlogs again. The filter server's binlogs are much smaller than those of the main master because a large number of statements have been left out. This would normally have taken place on each and every regular slave.

The regular slaves are configured against the filter server. That means they only receive the pre-filtered stream of statements that have made it into the filter's binlogs through the `replicate-ignore-*` and `replicate-do-*` directives. This is represented by the arrows in the picture.

Because slaves can go offline for extended amounts of time, binlogs could easily reach dozens of gigabytes in a few days. With the much smaller filtered binlogs you can now purge the large main master's binlogs as soon as you have made a full backup, instead of freeing more space than is needed by the additional filter instance.

## **Other storage engines than InnoDB**

Be advised that if you are using a different storage engine than InnoDB for your tables (especially MyISAM), you will need to do a little more tweaking. This is because the example relies on MySQL's being very lenient concerning errors in many cases. We added the `skip-innodb` option into the `my.cnf` config file. This means that InnoDB will not be available at runtime. Because the `master_struct.sql` dump file contains CREATE TABLE ... ENGINE=InnoDB statements, MySQL falls back to the default storage engine we configured to be the blackhole engine.

If you are using MyISAM tables, there is no need for the server to automatically change table type because MyISAM is always available (MySQL stores its user information and other things in MyISAM tables). So you would need to adapt the `master_struct.sql` file before sourcing it into the filter server. I recommend using sed, like this:

```
~/blacktest$ sed -e 's/ENGINE=InnoDB/ENGINE=BLACKHOLE/g' \
>      master_struct.sql > master_black.sql
```

This will replace all occurrences of the InnoDB engine with the blackhole engine and save the result into a new file. Please keep the original file, too, as it will be needed for the filter server.

# Setting up slaves via network streaming

If you need to reset one or more slaves regularly, say every morning before business begins, importing a SQL dump might take too much time, especially if the slaves are low-end machines without a sophisticated I/O subsystem.

In this recipe, we will present a way to set up a MySQL slave with minimal I/O load on the hard drive and the network adapter of the slave. The example assumes a Linux-based machine; however, you should be able to apply this to Windows as well, but you will have to download some free tools most Linux distributions come with out of the box.

The general idea is to have a more powerful machine, which can be the master if needed, allow, to prepare a complete set of data files for the slaves and later stream them to the slave's disk from a web server.

## Getting ready

To try this out, you will need a master server with at least one slave. Additionally, a machine with a web server installed is required. Depending on your setup, the master server may not be suitable for this task. In the example that follows, we will assume that the master server has a web server running.

## How to do it...

1. Set up a fresh temporary MySQL daemon with a configuration similar to the master.
2. Dump the data from the master with the `--master-data` option and feed it into the temporary server.
3. Shut down the temporary server and compress the archive.
4. Transfer the archive to the slaves.
5. Adapt the slaves to accept connections from the master.
6. Run the slaves and let them connect and catch up with the master.

## How it works...

This recipe is based on the fact that you can quite easily copy MySQL's data files (including InnoDB table space files) from one machine to another, as long as you copy them to a slave. You first create a ready-to-go set of slave data files on a relatively powerful machine and then move them to multiple slaves with weaker hardware. Usually, those files will be bigger than a standard SQL dump file that is usually used for slave setups. But no parsing and processing is done on the target system. This makes the whole thing mostly network and linear disk I/O bound.

## Replication

---

The idea behind this concept is to relieve the individual slaves from importing SQL themselves. As their hardware is rather slow and MySQL only supports single thread slave SQL execution, this can be very time consuming. Instead, we use the master resources temporarily as a single *power-slave* and let it handle the process of import, then provide any number of identical slaves with its data files. This will reduce the load on the master and allow the other slaves to simply unpacking some files.

While this does not really save anything in terms of bytes that need to be written to the slave's disk, the access pattern is much more sensible. The following table compares the data transfers for a regular SQL import from local disk and the proposed alternative for a gzipped SQL file, which will lead to approximately 2GB of InnoDB table space files:

Regular SQL Import	Prepared Data File Deployment
Linear write 60MB download to local disk	Download 60MB, directly streamed to data files, written linearly
Linear write 2GB initial creation of InnoDB data files	n/a
Linear read 60MB SQL.gz, interleaved with random write 2GB to data files	n/a
<b>4GB total read/written randomly</b>	<b>2GB linear write</b>

Importing a SQL file from the local hard disk means there are continual seeks between the current position in the SQL text file and the server's data files. Moreover, as the database schema may define lots of indexes, there is even more random disk write activity when executing simple `INSERT` statements.

In contrast unpacking ready-made InnoDB table spaces (or MyISAM table files for that matter) is basically just linear writing.

## Temporary daemon

The SQL dump needs to be executed at least once. So, we set up a temporary MySQL daemon with a stripped down configuration that is close to the actual slaves—meaning the parameters that affect the storage files must match the slaves to create compatible table spaces.

Every time you want to prepare such a new slave installation image, the temporary daemon should be started with an empty data directory. While not strictly necessary, we prefer to delete the table space and transaction log files every time because it allows for better compression rates later.

The data files should be created close to the size that will be needed, maybe a little smaller to prevent the need for them to grow. Nevertheless, specify the last data file to be non-extending. Otherwise the process of importing the SQL data may lead to filling the data directory with many small files.

---

Also, you should allow InnoDB to add larger chunks to the last data file if needed (up to 8MB). Extending the files is associated with some overhead, but using bigger chunks reduces the impact on the I/O subsystem. You should be fine with 50MB or 100MB. The bigger the chunk size is, the less often InnoDB will have to extend the file. See the manual section on InnoDB configuration for more info.

## Dumping master data

Once you have the temporary daemon running, use the `mysqldump` tool with the `--compact`, `--no-data` and `--single-transaction` options to create a dump of the database(s) you want to replicate. In order to save time and disk space, you may find it useful to pipe the output directly through the `mysql` command-line client and feed it into the target temporary server.

## Shutting down and compressing

You can now shut down the temporary server. Compress the data directory. Depending on what you want to configure permissions, you may include or exclude the `mysql` schema. Make sure that you have the temporary server set up with as low permissions as possible and do not include the `mysql` schema along.

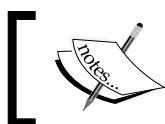
For compression, you should not use the ZIP format. It contains a catalog of all files and cannot be decompressed at its very end; so piping it through a decompression program on the fly will not work. Instead, we use a gzipped tarball. This allows us to download and to pipe the data stream to `tar` and then `gunzip` before directing it to disk.

## Transferring to the slave and uncompressing

On the slave we suggest `curl` as a download tool. It is important that the tool you choose is able to output the downloaded file directly to standard out. With `curl` that is quite simple, as it is its default behavior. It also handles files larger than 2GB, which some versions of `wget` have problems with. The command line should look similar to this:

```
curl http://the.server/mysql_data.tgz | tar -C /the/target/data
```

`curl` will download the file and pipe it to `tar` to decompress into the target data directory.



Do not miss the final `-` at the end of the command!

You will find that on a local area network, downloading and unpacking will be considerably faster than having MySQL to first create the empty data file and then import the SQL dump file for reasons stated above.

## *Replication*

### **Adjusting slave configuration**

When the data files have reached their destination on the slave, you may need to adjust slave settings. This especially depends on whether you copied fixed size data files (in which case you can prepare the config file in advance) or used the autoextend option of the InnoDB table space file. In that case, you could write a little script that takes a template my.cnf file with your basic settings and replaces some placeholders for the data file-related settings. One of those is the size of the last InnoDB data file from the archive. It will be a fixed size file on the slave. Another file will then be added at the first slave start.

### **Connecting to the master**

One last thing that needs to be done is to read the master's current binlog file name and position from the `master.info` file. This is required because once the slave server has started you will need to provide correct credentials for the replication user. You must explicitly tell the slave which master host to connect to. Unfortunately, when issuing the `CHANGE MASTER TO` command on the slave, which includes a master host name, all information about previous master binlogs—the corresponding offset—is discarded (see MySQL manual, chapter 12.6.2.1 *CHANGE MASTER TO* Syntax at <http://dev.mysql.com/doc/refman/5.1/en/change-master-to.html>).

Therefore, you will need to tell the slave again where to begin replication.

One possible solution is to read the contents of the `master.info` file that was brought with the data files into a bash script array and inject the values into the statement:

```
arr = ( $(cat master.info) )
mysql -e "CHANGE MASTER TO master_host='the.master.server',
          user='replication_user', master_password='the_password',
          master_log_file='${arr[2]}', master_log_pos=${arr[3]}"
```

The format of the `master.info` file is described in the MySQL manual.

### **Starting the slave**

As soon as you issue a `START SLAVE` statement, the slave will connect to the master and begin to catch up with whatever has happened since the time when the dump was taken.

## **Skipping problematic queries**

There are occasions where something goes wrong and a problem prevents one or more servers from updating. The reasons for this can be several, but most often some discrepancy between the master's and the slave's data set will cause a statement to fail. A statement that failed on the master will also fail on the slave if it was not included in the binlog. If the slave fails to update a row, it will skip the statement and move on to the next one.

This is where the basic principle of assuming master and slave being equal becomes too simple. It can lead to a potentially long series of statements executing on the slave on a different set of data than the master has. Depending on how long this goes unnoticed, the master and slave can drift out of sync unnoticed, until a statement cannot be executed successfully on the slave—for example because a foreign key constraint fails on the slave.

Fortunately, not every problem stems from such a serious error, which can often only be repaired by resetting the affected slaves to a known good state.

Often a slave stops the replication because a record to be inserted is already present, resulting in key uniqueness violation error. This is especially likely when (accidentally or on purpose) you are working on the master and the slaves, modifying data on both sides even to fix a replication problem.

In this recipe, we will show you how to skip one or more problematic queries—meaning instructions replicated from the master that will not execute correctly on the slave.

## Getting ready

We will demonstrate the skipping of problematic queries in a contrived error scenario. To test this for yourself, you will need two MySQL servers set up as master and slave, being in sync. As an example, we will use the `sakila` sample database to demonstrate an `INSERT` that fails on the slave because it was previously inserted manually by accident.

## How to do it...

1. Connect to the master using a MySQL client. Make `sakila` the default schema.
2. With a second client, connect to the slave. Make `sakila` the default schema as well.
3. On the slave, enter the following command to insert a new row:

```
slave> INSERT INTO category (name) VALUES ('Inserted On Slave');
```

```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
slave> INSERT INTO category (name) VALUES ('Inserted On Slave');

slave> SELECT * FROM category WHERE name='Inserted On Slave';
+-----+-----+
| category_id | name           | last_update   |
+-----+-----+
|          17 | Inserted On Slave | 2009-07-22 19:22:21 |
+-----+-----+
1 row in set (0.00 sec)

slave>
```

*Replication*

In this case, the **category\_id** column was automatically set because it is auto-incrementing. At this point, the master and the slave are already different because this record does not exist on the master.

4. On the master, insert a new record:

```
master> INSERT INTO category (name) VALUES ('Inserted On Master');
```

```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
master> INSERT INTO category (name) VALUES ('Inserted On Master');

master> SELECT * FROM category WHERE name='Inserted On Master';
+-----+-----+
| category_id | name           | last_update   |
+-----+-----+
|      17 | Inserted On Master | 2009-07-22 19:23:18 |
+-----+-----+
1 row in set (0.00 sec)

master>
```

You can see that the master also picked **17** as the **category\_id**. It has been written to the binlog and has by now probably been replicated to the slave.

5. Have a look at the replication status:

```
slave> SHOW SLAVE STATUS \G
***** 1. row *****

Slave_IO_Running: Yes
Slave_SQL_Running: No
...
Seconds_Behind_Master: NULL
```

You can see that the replicated insert has failed. No more statements replicated from the master will be executed (Slave\_SQL\_Running: No).

6. Repair the damage by making sure the slave records a failure:

```
slave> UPDATE category SET name='Inserted On Master' WHERE category_id=17;
```

Now the data on master and slave are identical again.

7. Tell the slave that you want to skip the (one) INSERT statement that cannot be replicated from the master and that cannot be executed:

```
slave> SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1;
```

8. Start the slave SQL thread again and check the replication status:

```
slave> START SLAVE;
```

---

```
slave> SHOW SLAVE STATUS \G
***** 1. row *****
...
Slave_IO_Running: Yes
Slave_SQL_Running: Yes
...
Seconds_Behind_Master: 0
```

You can see that the replication is up and running again.

## How it works...

When the slave data was out of sync in such a way that a statement from the master failed (in this case because of a duplicate identity column value), the slave server stopped executing any more SQL statements, even though in the background they were still being received from the master and stored in the relay logs. This is what the `Slave_IO_State` and `Slave_SQL_Running` columns from the output of `SHOW SLAVE STATUS` say.

MySQL does this to give you a chance to look at the problem and determine if you can fix the situation somehow. In this very simple example, the solution is simple because we can easily bring the slave's data back in sync with the master by modifying the record in question to match the contents that were sent from the master and then skip the `INSERT` statement received from the master using the `SQL_SLAVE_SKIP_COUNTER` global variable. This will skip the problematic `INSERT` statement from the relay logs, when the slave is next told to start. In our case, we can simply delete the problematic `INSERT`.

After that the replication is back in sync, as master and slave are now based on identical sets again, allowing the following statements to be replicated normally.

## There's more...

Another solution in this particular case could have been to delete the record on the master and then restart the replication with `START SLAVE`. As the `INSERT` from the master has not yet been executed, replication would continue as if nothing had happened.

However, under more realistic circumstances, when confronted with a situation like this, we might not have a chance to delete the row on the slave due to foreign key constraints. This is because we immediately took care of the problem and we were sure that in the meantime no programs could have written to the slave, possibly creating new references to the row that were we able to remove it.

Depending on your application architecture and how fast you noticed the problem, the process might have been writing data to the slaves to tables that are designed for replication.

## Replication

While in this simple case, we would be sure that the replication setup is now back again, you often will not be able to tell for certain at which point in time a replication originated. `INSERT` statements of duplicate keys will immediately cause an error to apparent. `UPDATE` or `DELETE` statements will often succeed in executing, but would have different effects on the slave than on the master, when they were previously out of sync.



Problems like this can corrupt the data on your slaves silently for extended periods of time. When you find out in the end, it is often too late to recover without resorting to setting up the slave afresh.

When in doubt, we recommend to first use `mk-table-checksum` and `mk-table-sync`, described in the *Checking if servers are in sync* recipe in this chapter, or more generally to set up the slave from a well-known good state to be completely sure!

## Checking if servers are in sync

As MySQL cannot detect if two servers are in sync (that is they contain the same replicated data), one would often like to verify that master and slave are still working on identical sets to be sure no corruption has occurred yet.

For this purpose, the excellent Maatkit suite of programs (see <http://www.maatkit.org>) contains a handy tool called `mk-table-checksum`. It automatically calculates checksums of tables on one or more servers, which can then be compared. Should the checksums differ, then the table in question is not identical on the machines involved in the check.

The servers involved need not necessarily be a replication master and slaves, but any set of servers you wish to compare. `mk-table-checksum` has an additional option to means of checking the special case in replication environments to see if a master and its slaves are in sync. See the *There's more...* section at the end of this recipe for more information about this feature.

### Getting ready

Maatkit is written in Perl. While on most Unix-like systems this scripting language is installed by default or can be easily downloaded and set up, Windows users will not be lucky in general. If you are stuck with Windows, you might want to take a look at ActiveState's mature Perl implementation for Windows.

Moreover, you are definitely going to need the Maatkit `mk-table-checksum` tool. You can get it directly from <http://www.maatkit.org>. Also, download the `mk-checksum` companion tool and put it in the same directory as `mk-table-checksum`.

In this example, we will compare two MySQL servers that differ in the `sakila` database's `country` table located on machines called `serverA` and `serverB`.

You will need to have user accounts for both machines that have permission to connect and execute statements remotely.

The command lines in this recipe might change with newer versions of Maatkit, as it is still in active development. Double-check with the online manual that the instructions provided here are still current before trying them out.

## How to do it...

1. On a command shell prompt, enter the following line, assuming `mk-table-checksum` is in the current directory and executable:

```
$ ./mk-table-checksum h=serverA,u=userA,p=passwordA
h=serverB,u=userB,p=passwordB | ./mk-checksum-filter
```

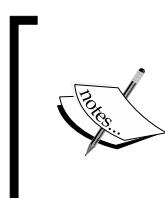
2. Check the output of this command (formatted and abbreviated)

Database	Table	Chunk	Host	Engine	Count	Checksum
sakila	country	0	serverA	InnoDB	NULL	2771817
sakila	country	0	serverB	InnoDB	NULL	3823677

Notice the last column: The checksums do not match—the tables are not identical.

## How it works...

`mk-table-checksum` connects to all servers listed on the command line and calculates checksums for all tables. Identical table contents result in identical checksums. Since checksums from two servers do not match for any given table, there must be a difference in their contents. The `mk-checksum-filter` tool removes all lines from the output that indicate a checksum mismatch.



It is important to know that the checksums are different if you employ different versions of MySQL across servers. In this case, a different checksum might just be the result of the different versions!

`mk-table-checksum` offers several algorithms for checksumming, each with different speeds and different levels of resilience against certain kinds of data differences that cancel each other out, leading to identical checksums, but for different data. The following table summarizes the available algorithms and their characteristics.

*Replication***There's more...**

Due to the asynchronous nature of MySQL replication, executing the `checksum` statement remotely from a single machine may not yield reliable results. This is because the database might already contain modifications that have not been executed by each slave.

To compensate, `mk-table-checksum` offers a special mode to check slaves and calculate checksums from them. Instead of executing the calculations remotely, the statements to do so are written into the master's binlog and then sent off to the slaves via the regular replication mechanism. This ensures that each slave will calculate the checksum at the correct time with respect to the transaction order. The results are then stored in a table on the slave that can be retrieved with a second command remotely later on. To use this feature, you need a user with the `CREATE TABLE` privilege to create a table for this purpose on the slaves.

For more details, see the `--replicate` and `--create-replicate-table` options in the Maatkit online manual.

**Avoiding duplicate server IDs**

A key configuration item in any replication setup is server IDs. They must be unique across all participating master and slave machines. Unfortunately, there is no official way to ensure uniqueness reliably. Instead, when you introduce duplicates by mistake, strange behavior may result. Generally, this happens when cloning the machines from an image.

Most importantly, on the master server you will not see any indication of the problem. The problem arises only on the slaves without clearly stating the root cause of the problem. See the *There's more...* section of this recipe for more details.

**Getting ready**

The `server-id` setting does not carry any meaning in and of itself, but is only used to internally distinguish servers from each other. Generally, administrators setting up multiple servers enter sequential or random values for this field. This requires a list of server IDs already issued, preferably including the host name. As with most things in life that are done manually, maintaining this list is likely to become a burden and will be forgotten.

Instead, you can assign server IDs based on features of the individual machines that are usually unique already, for example, the network interface's MAC address or the IP address, which should remain reasonably fixed for any server machine as well.

IP addresses are usually shown in a dotted notation of four numbers between 0 and 255. Because MySQL requires `server-id` to be specified as single decimal value, you will need to convert it first:

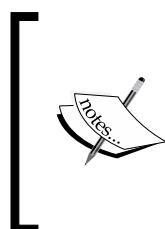
## How to do it...

1. Determine your server's IP address. Make sure not to use the loop-back similar pseudo-interface. In this example we assume an IP address of 10.0.0.16.
2. Convert the address to hexadecimal. Mostly any calculator application can do this. You enter each of the four numbers in decimal mode and then switch to hex mode. Just replace each individual decimal value with its hexadecimal counter-part. For the address above you will come up with: 0a . 00 . 9f . 16
3. Append the bytes (that is just remove the dots between them) and convert to decimal by switching modes:  $0a009f16_{\text{HEX}} = 167812886_{\text{DEC}}$
4. Insert that final value as the server ID in the [mysqld] section of configuration file:

```
[mysqld]
server-id=167812886
```

## How it works...

The IP address serves to uniquely identify a network interface (and therefore a machine) on a network. We leverage this uniqueness by recycling the IP address as the server ID. Most operating systems will issue a warning when an IP address conflict is detected, which indirectly points to a replication problem as well.



Of course, traditional IPv4 addresses (those usually noted in the above notation) are only unique in their respective subnet. That means you should not rely on this recipe alone for your server IDs if master and slave machines are located in different locations from a network topology point of view!

## There's more...

The IP address is only one possible unique value you can use. Anything that you can fit into the valid numeric range of the `server-id` setting can be used. Ideally that value would never change over the lifetime of a server, much like a good Primary key, just not for a database record, but the server as a whole.

You could use any sort of serial number your hardware vendor already assigns to the server if it is purely numeric and fits the valid range of 4 bytes. However, this ties you to the idea of uniqueness, which you cannot verify reliably. Alternatively, the last 4 bytes of the server's MAC address (those are 6 bytes long, starting with a vendor specific prefix)

*Replication*

## Recognizing symptoms of duplicate server IDs

Despite all care, errors can happen and duplicate server-ids can be issued. Unfortunately MySQL will not tell you explicitly when you have non-unique server-ids in your replication setup. While on the master, you will not see any evidence in the log files that something is wrong, slaves will show strange behavior and issue seemingly unrelated error messages in their log files in short succession:

```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
mysql> show slave hosts \G
***** 1. row *****
    Server_id: 13307
        Host: slave_1701.example.com
        Port: 3306
  Rpl_recovery_rank: 0
      Master_id: 1000
mysql>
```

Of course, the names of machines, log files, and positions will vary, but the message is clear: a sudden assumed shutdown of the master, followed by immediate retries and failing again is a strong indication of a problem with replication server-ids.

## Setting up slaves to report custom information about themselves to the master

When you issue a SHOW SLAVE HOSTS command on a replication master server, you will see a list of slaves connected, provided that they are set up correctly.

Unfortunately, by default they are not, so unless you specifically configure them to do so, the slaves will not register themselves with the master. In a default setup you might not even see any slave in the output of the above command or in the Replication Status pane in the MySQL Administrator at all, even though there might be several configured against this master.

In this recipe, we will show you how to configure a slave to tell its master some details that might come in handy when troubleshooting.

 Please note that due to a bug in MySQL the output of the SHOW SLAVE HOSTS command is not always reliable! Sometimes it will report hosts being available when in fact they are currently not. The only way that seems to fix an erroneous display is to stop and start the master server.

This effectively makes this feature unsuitable for the purpose of the actual monitoring of the current health of the slaves. It, nevertheless, provides a way to gather some basic inventory information on their location and some other details described below.

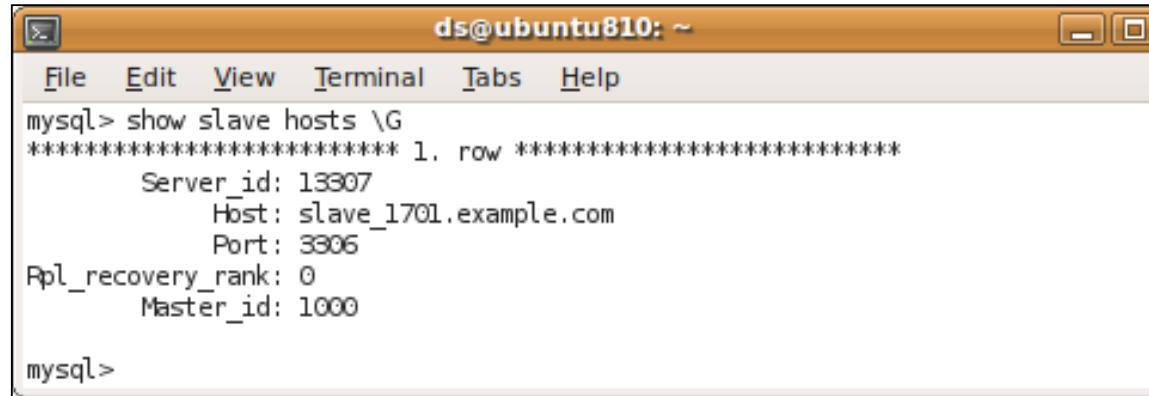
## Getting ready

To follow along, you will need sufficient operating system privileges to modify the slave configuration file (`my.cnf` or `my.ini` depending on your operation system). To activate that status on the master you will need a MySQL user there as well.

## How to do it...

1. Shut down the MySQL service.
2. Open its configuration file in a text editor.
3. Make sure the following line is present in the `[mysqld]` section:  
`report-host=slave_1701.example.com`
4. Save the changes.
5. Restart the MySQL service.
6. On the master, issue this command to verify your change was successful:

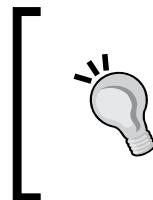
```
mysql> show slave hosts \G
```



```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
mysql> show slave hosts \G
***** 1. row *****
    Server_id: 13307
        Host: slave_1701.example.com
        Port: 3306
    Rpl_recovery_rank: 0
        Master_id: 1000

mysql>
```

You might of course see many more slaves here, depending on how they are configured.



Should you ask yourself what the `Rpl_recovery_rank` line in the output means, you may simply ignore it. It seems it was introduced some years ago but never put to active use.

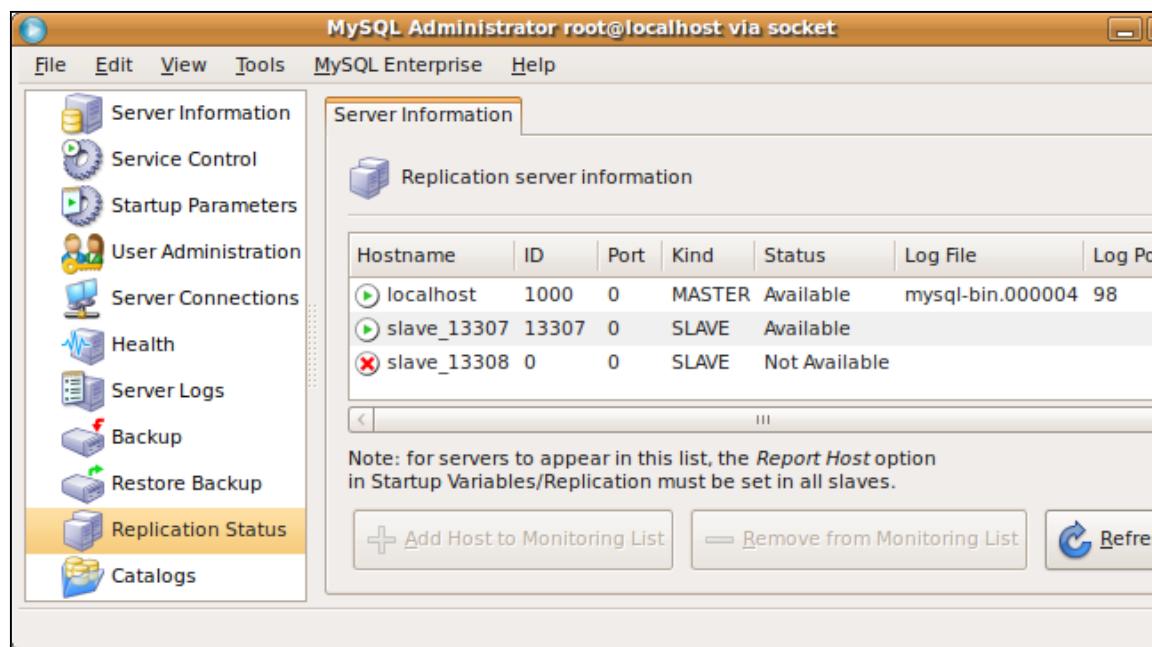
## How it works...

Usually, slaves do not report any details about themselves when they connect to the master. By adding some options in their configuration you can, however, make them announce their presence.

## Replication

We strongly recommend setting up all your slaves to register with the master, especially when you are dealing with many masters and slaves. This can be very helpful to keep track of things.

MySQL Administrator allows you to remember all slaves it has seen once and display a warning on its **Replication Status** pane when a machine previously known does not appear later. This particular piece of information is not reliable, however; see the warning in this recipe's introduction for more information.



## There's more...

The general idea behind the `report-host` setting is to give an idea about how to access the slave machine via the network. As the slave might be located behind some sort of NAT router, its IP address might not be very helpful. So, in general, it can be helpful for the slave to report a fully qualified domain name that can be used to reach it, if applicable.

However, it is by no means mandatory to do so. If you do not intend to access the slaves remotely, you might just enter any other piece of information you like to see in the `report-host` command mentioned in this recipe or the MySQL Administrator pane. As you can see in the previous screenshot, I set up the slaves to report back a name suffixed with their `server-id` value. Doing so works around a bug in MySQL Administrator that knows how to remember the slaves it has seen before, but sometimes forgets their `server-id`.

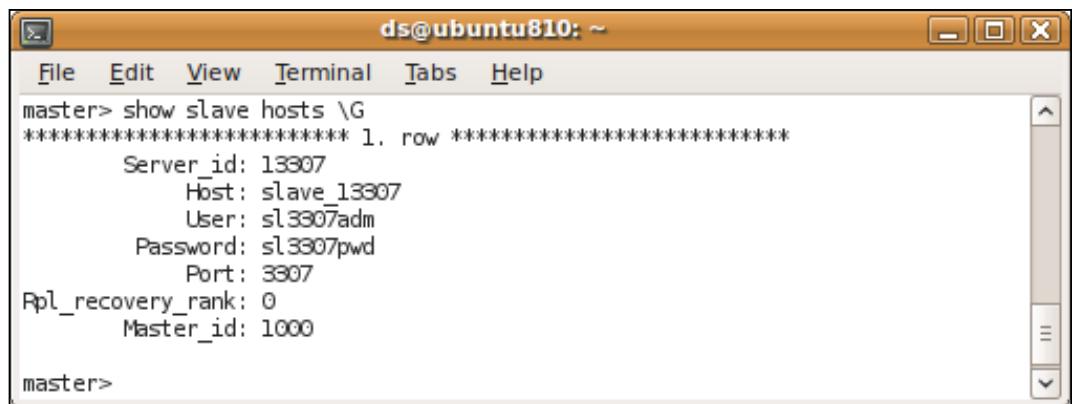
Showing slaves currently unavailable is a feature of MySQL Administrator; the `SHOW SLAVE HOSTS` command will not mention them at all. To leverage this you must click the **Add to Monitoring List** button for each slave once it is connected. Otherwise, they will not appear in the list even when they are not connected.

---

Apart from the `report-host` configuration setting there are three more options you know about:

<b>Setting</b>	<b>Description</b>
<code>report-port</code>	Informs about the port that must be used to reach the slave domain name reported by report-host. This can be sensible if forwarding has been set up.
<code>report-user</code>	Report a username that can be used to connect to the slave recommended to use!
<code>report-password</code>	Report a password that can be used to connect to the slave recommended to use!

For completeness, this is what the output of `SHOW SLAVE HOSTS` will look like if you follow our advice and configure the slave to report a set of login credentials and the master has been started with the `show-slave-auth-info` option:



```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
master> show slave hosts \G
***** 1. row *****
    Server_id: 13307
        Host: slave_13307
        User: sl3307adm
        Password: sl3307pwd
        Port: 3307
    Rpl_recovery_rank: 0
        Master_id: 1000
master>
```

While the `report-port` setting might be useful, we strongly suggest you refrain from using the `report-user` and `report-password` options for security reasons.



Even though the master server will only display these values when it is started with the `show-slave-auth-info` option, it is still very risky to send login credentials over the network in this manner. You should always use more secure ways to exchange login information!



# Index

In this chapter, we will cover:

- ▶ Adding indexes to tables
- ▶ Adding a fulltext index
- ▶ Creating a normalized text search column
- ▶ Removing indexes from tables
- ▶ Estimating InnoDB index space requirements
- ▶ Using prefix primary keys
- ▶ Choosing InnoDB primary key columns
- ▶ Speeding up searches for (sub)domains
- ▶ Finding duplicate indexes

## Introduction

One of the most important features of relational database management systems—being no exception—is the use of indexes to allow rapid and efficient access to the amounts of data they keep safe for us. In this chapter, we will provide some useful you to get the most out of your databases.

## *Indexing*

### **Infinite storage, infinite expectations**

We have got accustomed to nearly infinite storage space at our disposal—storing everything from music to movies to high resolution medical imagery, detailed geographical information, or just plain old business data. While we take it for granted that we hardly ever run out of space, we also expect to be able to locate and retrieve every bit of information we want in an instant. There are examples everywhere in our lives—business and personal:

- ▶ Your pocket music player's library can easily contain tens of thousands of songs yet can be browsed effortlessly by artist name or album title, or show you a list of the top 10 rock songs.
- ▶ Search engines provide thousands of results in milliseconds for any arbitrary search term or combination.
- ▶ A line of business application can render your sales numbers charted and grouped on a map, grouped by sales district in real-time.

These are a few simple examples, yet for each of them huge amounts of data must be combed to quickly provide just the right subset to satisfy each request. Even with the immense speed of modern hardware, this is not a trivial task to do and requires some clever techniques.

### **Speed by redundancy**

Indexes are based on the principle that searching in sorted data sets is way faster than searching in unsorted collections of records. So when MySQL is told to create an index on one or more columns, it copies these columns' contents and stores them in a sorted manner. The remaining columns are replaced by a reference to the original table with the unsorted data.

This combines two benefits—providing fast retrieval while maintaining reasonably efficient storage requirements. So, without wasting too much space this approach enables you to create several of those indexes (or *indices*, both are correct) at a relatively low cost.

However, there is a drawback to this as well: while reading data, indexes allow for very fast speeds, especially in large databases; however, they do slow down writing operations. In the course of **INSERTs**, **UPDATEs**, and **DELETEs**, all indexes need to be updated in addition to the data table itself. This can place significant additional load on the server, slowing down all operations.

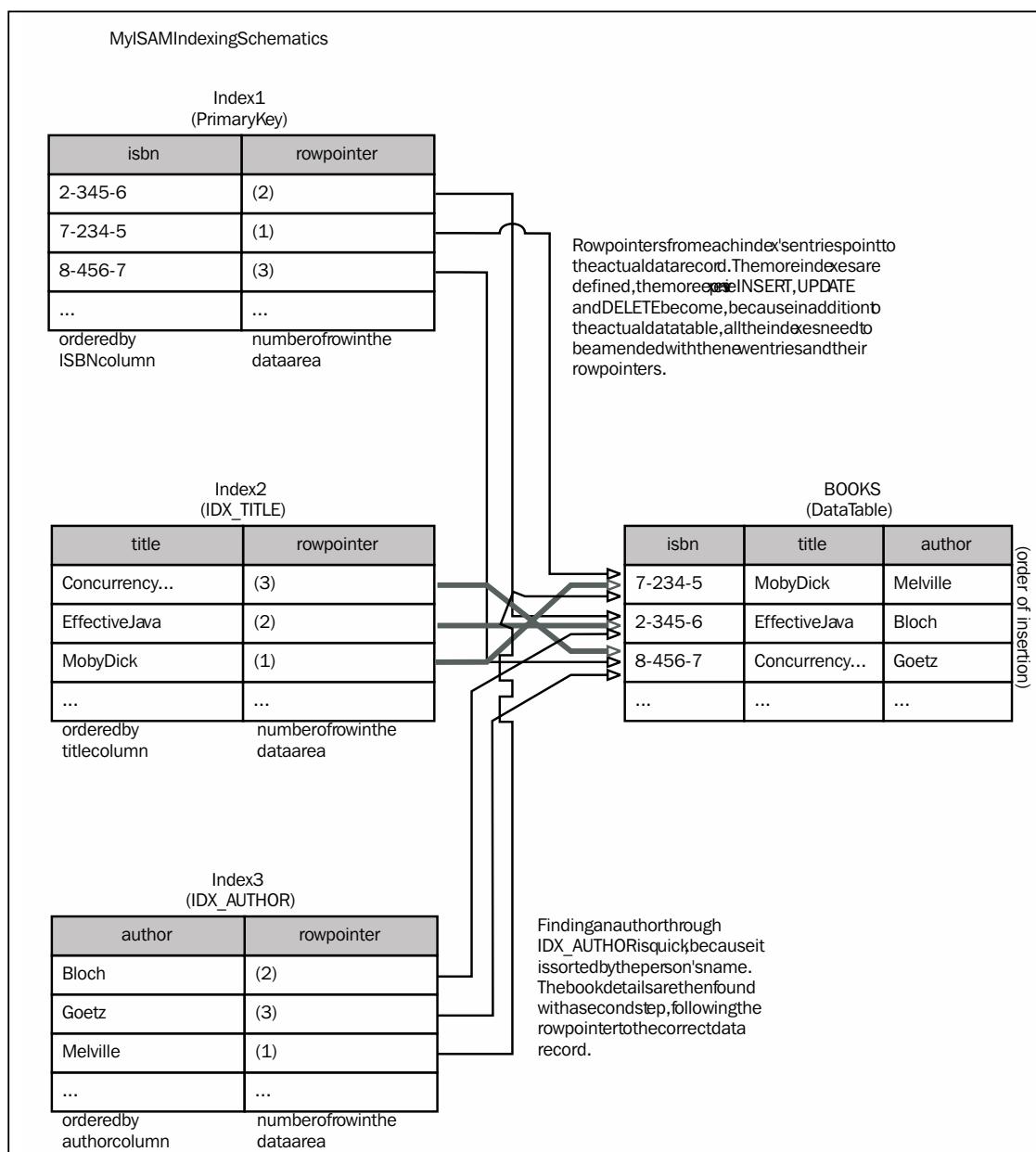
For this reason, keeping the number of indexes as low as possible is paramount, especially for the largest tables where they are most important. In this chapter, you'll find some guidelines that will help you to decide how to define indexes and show you some pitfalls to avoid.

### **Storage engine differences**

We will not go into much detail here about the differences between the MyISAM and InnoDB storage engines offered by MySQL. However, regarding indexes there are some

## MyISAM

In the figure below you can see a simplified schema of how indexes work with the MyISAM storage engine. Their most important property can be summed up as "all indexes are equal". This means that there is no technical difference between the primary and secondary keys.



The diagram shows a single (theoretical) data table called **books**. It has three columns named **isbn**, **title**, and **author**. This is a very simple schema, but it is sufficient for our purposes. The exact definition can be found in the *Adding indexes to tables* recipe chapter. For now, it is not important.

MyISAM tables store information in the order it is inserted. In the example, there are records representing a single book each. The ISBN number is declared as the primary key for this table. As you can see, the records are not ordered in the table itself—the ISBN numbers are inserted in a random order.

## Indexing

Now, have a look at the first index—the **PRIMARY KEY**. The index is sorted by the ISBN number. Associated with each index entry is a row pointer that leads to the actual data record in the **books** table. When looking up a specific ISBN number in the primary key index, the database server follows the row pointer to retrieve the remaining data fields. The same holds true for the other two indexes *IDX\_TITLE* and *IDX\_AUTHOR*, which are sorted by the respective columns and also contain a row pointer each.

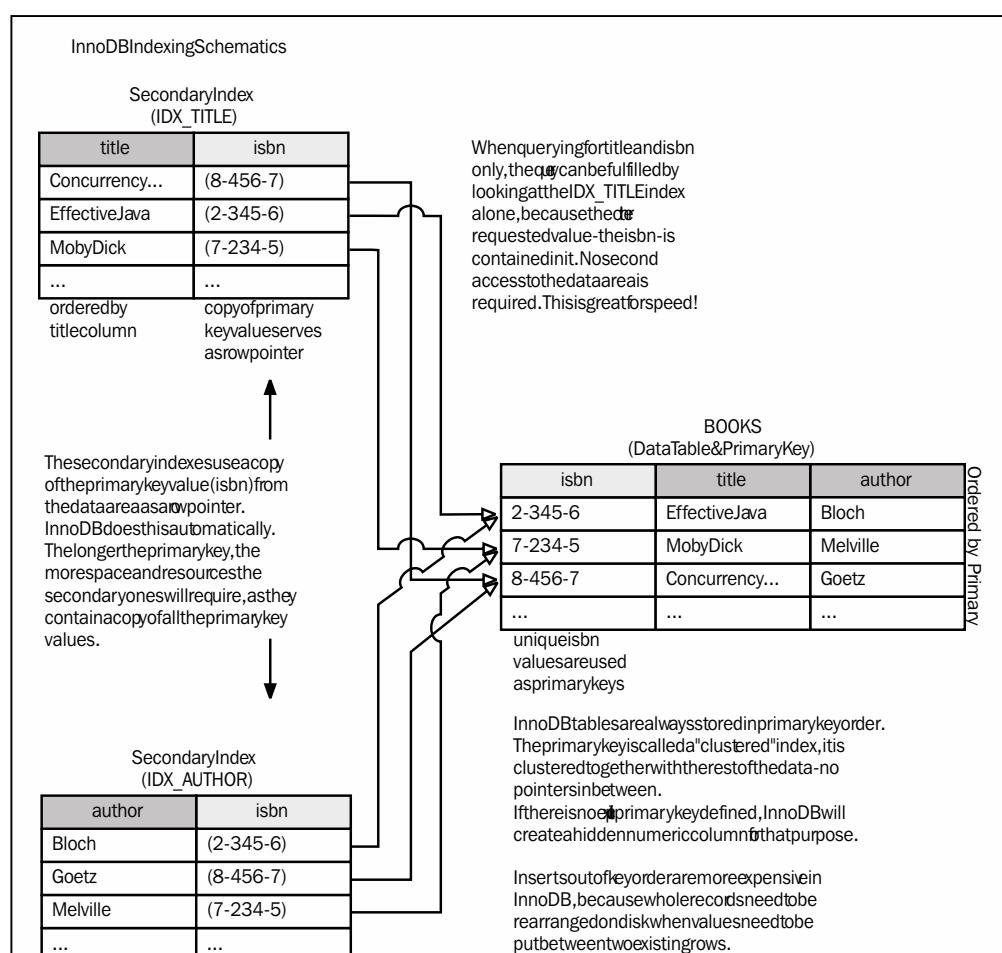
Looking up a book's details by any one of the three possible search criteria is a two-step operation: first, find the index record, and then follow the row pointer to get the rest of the data.

With this technique you can insert data very quickly because the actual data records are simply appended to the table. Only the relatively small index records need to be kept in memory, meaning much less data has to be shuffled around on the disk.

There are drawbacks to this approach as well. Even in cases where you only ever want to look up data by a single search column, there will be two accesses to the storage subsystem—one for the index, another for the data.

## InnoDB

However, InnoDB is different. Its index system is a little more complicated, but it has some advantages:



---

## Primary (clustered) indexes

Whereas in MyISAM all indexes are structured identically, InnoDB makes a distinction between the primary key and additional secondary ones.

The primary index in InnoDB is a *clustered index*. This means that one or more columns of each record make up a unique key that identifies this exact record. In contrast to MyISAM indexes, a clustered index's main property is that it itself is part of the data instead of being stored in a different location. Both data and index are *clustered* together.

An index is only serving its purpose if it is stored in a sorted fashion. As a result, whenever you insert data or modify the key column(s), it needs to be put in the correct location according to the sort order. For a clustered index, the whole record with all its data has to be repositioned.

That is why bulk data insertion into InnoDB tables is best performed in correct primary key order to minimize the amount of disk I/O needed to keep the records in index order. Since the clustered index should be defined so that it is hardly ever changed for existing records, that too would mean relocating full records to different sectors on the disk.

Of course, there are significant advantages to this approach. One of the most important aspects of a clustered key is that it actually *is a part* of the data. This means that when accessing data through a primary key lookup, there is no need for a two-part operation as with MyISAM indexes. The location of the index is at the same time the location of the data itself—there is no need for following a row pointer to get the rest of the column data after an expensive disk access.

Looking up a book by ISBN in our example table simply means locating it quickly, as it is stored in order, and then reading the complete record in one go.

## Secondary indexes

Consider if you were to search for a book by title to find out the ISBN number. An index on the name column is required to prevent the database from scanning through the whole (ISBN-sorted) table. In contrast to MyISAM, the InnoDB storage engine creates secondary indexes differently.

Instead of record pointers, it uses a copy of the whole primary key for each record to connect the connection to the actual data contents.

In the previous figure, have a look at the *IDX\_TITLE* index. Instead of a simple pointer to the corresponding record in the data table, you can see the ISBN number duplicated as well. This is because the **isbn** column is the primary key of the **books** table. The same goes for the other indexes in the figure—they all contain the book ISBN number as well. You do not need to (and should not) specify this yourself when creating and indexing on InnoDB tables, it is done automatically under the covers.

## *Indexing*

Lookups by secondary index are similar to MyISAM index lookups. In the first step, record that matches your search term is located. Then secondly, the remaining data retrieved from the data table by means of another access—this time by primary key.

As you might have figured, the second access is optional, depending on what information request in your query. Consider a query looking for the ISBN numbers of all known editions of Moby Dick:

```
SELECT isbn FROM books WHERE title LIKE 'Moby Dick%';
```

Issued against a presumably large library database, it will most certainly result in a two-step lookup on the *IDX\_TITLE* key. Once the index records are found, there is no need for a second lookup to the actual data pages on disk because the ISBN number is already present in the index. Even though you cannot see the column in the index definition, MySQL will skip the second seek saving valuable I/O operations.

But there is a drawback to this as well. MyISAM's row pointers are comparatively small, while the primary key of an InnoDB table can be much bigger—the longer the key, the more space that is stored redundantly.

In the end, it can often be quite difficult to decide on the optimal balance between the space requirements and maintenance costs on index updates. But do not worry; we will provide help on that in this chapter as well.

## **General requirements for the recipes in this chapter**

All the recipes in this chapter revolve around changing the database schema. In order to add, drop, or change indexes or remove them, you will need access to a user account that has an effective **CREATE INDEX** privilege or the **ALTER** privilege on the tables you are going to modify.

While the **INDEX** privilege allows for use of the **CREATE INDEX** command, **ALTER** is required for the **ALTER TABLE ADD INDEX** syntax. The MySQL manual states that the former is mapped to the latter automatically. However, an important difference exists: **CREATE INDEX** can only be used to add a single index at a time, while **ALTER TABLE ADD INDEX** can be used to add more than one index to a table in a single go.

This is especially relevant for InnoDB tables because up to MySQL version 5.1 every update to the definition of a table internally performs a copy of the whole table. While for small databases this might not be of any concern, it quickly becomes infeasible for larger ones due to the high load copying may put on the server. With more recent versions this might have changed, but make sure to consult your version's manual.

In the recipes throughout this chapter, we will consistently use the **ALTER TABLE ADD INDEX** syntax to modify tables, assuming you have the appropriate privileges. If you do not have to rewrite the statements to use the **CREATE INDEX** syntax.

## Adding indexes to tables

Over time requirements for a software product usually change and affect the underlying database as well. Often the need for new types of queries arises, which makes it necessary to add one or more new indexes to perform these new queries fast enough.

In this recipe, we will add two new indexes to an existing table called `books` in the `library` schema. One will cover the `author` column, the other the `title` column. The schema of the `books` table can be created like this:

```
mysql> CREATE DATABASE library;
mysql> USE library;
mysql> CREATE TABLE books (
    isbn char(13) NOT NULL,
    author varchar(64) default NULL,
    title varchar(64) NOT NULL,
    PRIMARY KEY  (isbn)
) ENGINE=InnoDB;
```

### Getting ready

Connect to the database server with your administrative account.

### How to do it...

1. Change the default database to `library`:

```
USE library;
```

2. Create both indexes in one go using the following command:

```
ALTER TABLE books ADD INDEX IDX_author(author), ADD INDEX
title(title);
```

### How it works...

The `ALTER` table statement shown above is almost self-explanatory. The `books` table is altered to be indexed with individual indexes on the `author` and the `title` columns. Each index is given an easily recognizable name: `IDX_author` and `IDX_title` for the `author` and `title` columns respectively.

Index names are helpful when you later decide to remove an index from a table. Instead of listing all the columns again, you can just refer to the index name.

*Indexing***Index names**

It is very common to name indexes with some sort of prefix like `IDX_` then append the column name(s) the index spans.

This is not strictly necessary and you might want to establish a different naming scheme. Whatever you choose, make sure you follow your schema and assign names consistent with it for all your indexes.

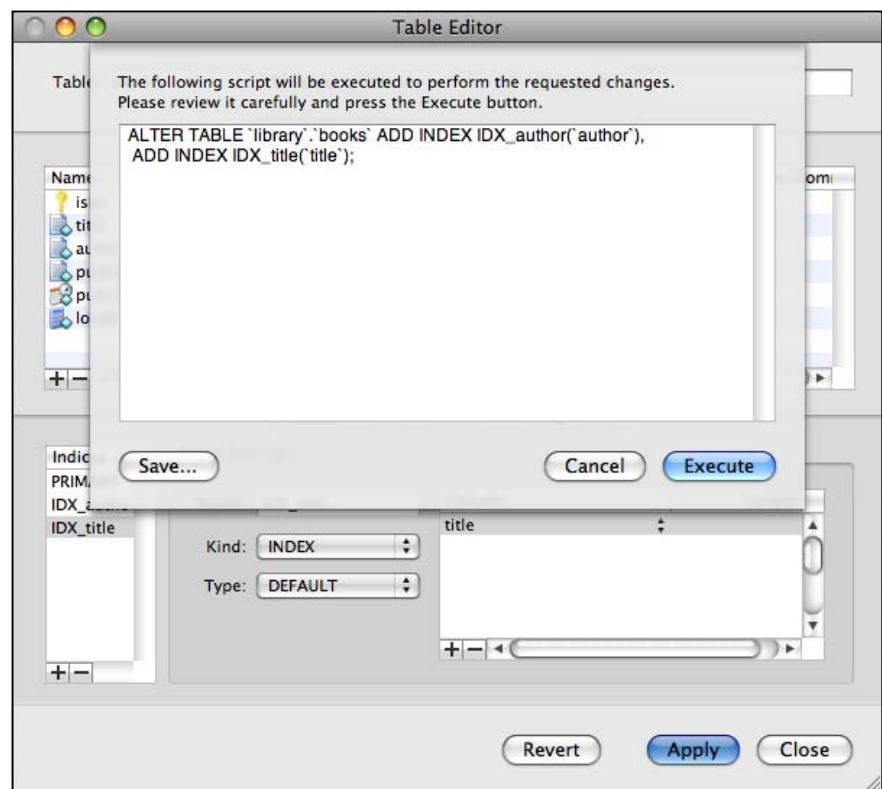
**There's more...**

There are some more details worth knowing about when creating indexes on any given table.

**Using MySQL Query Browser to generate the SQL statements**

Setting up indexes can be done either through a command line as shown earlier or through an arguably more comfortable graphical tool like MySQL Query Browser. Especially when dealing with more complex table setups, the graphical presentation can provide additional context. When applying any changes to your database, the product will display and allow you to copy the full SQL statement(s) that are equivalent to the changes you made in the graphical editor.

This is very convenient because this way you can be sure not to make any mistakes concerning statement syntax, table, or column names. We usually make changes in MySQL Query Browser on a development or testing machine just to grab the SQL statements and put them into SQL update script files for later execution, for example, as a part of a software update routine. The following figure shows what the changes made in this browser look like. Note that the generated statements contain all table and column names in quotes. This is generally not required as long as those identifiers do not collide with keywords—something you should avoid anyway. Also, the statements will be *fully qualified*, which means the database name is put before the table name. This is also not strictly required if you set the default database to the right schema beforehand.



## Prefix indexes

In the example above, we created an index with default settings. This will create an index that covers the entire column. This is usually "just right". You may, however, have special requirements or possess knowledge about the table data that cannot be derived from the schema definition alone, making a custom index a better choice than the default one.

The detail most often changed in an index definition is the length of the index field. MySQL provides support for so-called *prefix indexes*. As the database does not know about the specific contents of the columns that are going to be stored in any particular column apart from the primary key, it has no choice but to take the safe route and consider the full length of the column when creating a sorted index copy.

For long columns in large tables, it can be a waste of space to copy the complete character values to the index, which in turn can have negative impact on performance just because there's more data involved.

You can aid the database to work more efficiently with your domain knowledge. In the example table the title can be up to 64 characters long. However, it is very unlikely that there will be a lot of books whose titles start alike and only differ in the last few characters. Having the index cover the maximum length is probably not necessary for quick lookups. By changing the index creation statement to include a prefix length (say 20 characters) for the column to be indexed, you can tell MySQL to only copy the first 20 characters of the column when creating the index:

## *Indexing*

As a result, the index will use less space—in terms of both disk usage and memory for queries. As long as the book title differs within the first 20 characters, this index is more efficient than one covering the full column.

Even when there is a certain number of titles that are identical within this 20 character limit, the index will still be useful. This is because as long as MySQL can rule out all but a few records, having to look at the actual table data for the final decision as to which row matches query conditions is still significantly faster than having to scan the whole table with a full scan.

Unfortunately, there is no easy-to-use formula to determine the ideal prefix length because it heavily depends on the actual data content. This is why by default the whole column is indexed.

## **Prefix primary keys**

Most documentation on indexing in some way or another covers the topic of prefix indexing for text type columns, using only a portion at the beginning of column contents instead of the whole values for the actual index.

However, often this topic is presented in a way that might suggest this only works for secondary keys; but that is not true. You can also use a *prefix primary key*, as long as the important requirement of a primary key is not violated: the uniqueness of each key value must be guaranteed.

## **See also**

- ▶ *Estimating InnoDB index space requirements*
- ▶ *Removing indexes from tables*

## **Adding a fulltext index**

Indexes are an important means of making sure a database performs well and returns results quickly when queried. However, they can only live up to their full potential when applied to well-structured data. Unfortunately, not all information we would like to query can be easily represented to fit into regular relational database tables and columns.

A prime example of this is free text. It does not follow any well-defined structure and therefore does not lend itself to the principle by which regular indexes work. For example, a fulltext index allows you to query for search terms no matter where in the indexed column they occur and not just at the beginning of the column as would be the case with normal indexes.

Fulltext indexes require you to use a special syntax to express your queries. Querying for words using the **LIKE** keyword will not be accelerated by a fulltext index. In this recipe you will learn how to create a fulltext index on an existing database column. For the purpose of this exercise, we will



InnoDB tables do not support fulltext indexing. This feature is only available for tables using the MyISAM storage engine.

## Getting ready

Connect to the database using your administrative account.

## How to do it...

1. Change the default database to forum:

```
USE forum;
```

2. Create the fulltext index using the following command:

```
ALTER TABLE posts ADD FULLTEXT INDEX IDX_content(content)
```

## How it works...

While regular indexes create ordered copies of the relevant columns to enable quick lookups, fulltext indexes are a more complicated matter.



### Dropping and recreating fulltext indexes for bulk data imports

When (first) inserting bulk data into a table, it is faster to first drop an existing fulltext index and then later recreate it. This will speed up the data insertion significantly because keeping the fulltext index up to date during data insertion is an expensive operation.

## There's more...

Here are some details that are important to know when including fulltext indexing in your applications.



Please be aware that changes to any of the parameters that follow require a rebuild of any fulltext index that was created before the change!

See the MySQL online manual at <http://dev.mysql.com/doc/refman/5.1/en/fulltext-fine-tuning.html> for more details.

## *Indexing*

### **Case sensitivity**

Fulltext index queries are usually run in a case-insensitive way. If you need case-sensitive fulltext search, you will have to change the collation of the affected underlying column to a binary variant.

### **Word length**

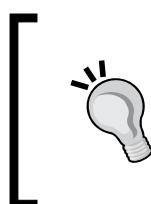
When a fulltext index is created, only words within a configurable range of lengths are considered. This helps to prevent acronyms and abbreviations being included in the index. You can configure the acceptable length range using the `ft_min_word_len` and `ft_max_word_len` variables. The default value for the minimum length is 4 characters.

### **Stopwords**

In every language, there are many words that are usually not wanted in fulltext searches, such as articles and prepositions. These so called stopwords might be "is, a, be, there, because, done" among others. They appear so frequently in most texts that searching for them is hardly useful. To save storage space and processing resources, these stopwords are ignored when building a fulltext index. MySQL uses a built-in default English stopwords list that defines what is to be ignored, which contains a list of about 550 words. You can change this list of stopwords with the `ft_stopword_file` variable. This variable takes a filename with a plain text file containing the stopwords you would like to use. Ignoring all stopwords can be achieved by setting this variable to an empty string.

### **Ignoring frequent words**

Frequent words will be ignored: if a search term is present in more than half of the rows of a table that is being searched, it will be considered a stopword and effectively ignored. This is useful especially in large tables; otherwise you would get half of the table as query hits, which can be considered useful.



When experimenting with fulltext search, make sure you have a reasonable number of rows in your table and a reasonably large dataset to play with. Otherwise you will easily hit the 50 percent mark of the table described above and not get any query results. This can be confusing as the results will look correct, making you think you did something wrong, while in fact everything is perfectly in order.

### **Query modes**

Apart from the default *human query* mode you can use a *boolean query* mode, which allows you to use special search-engine-like operators to be used—for example, the plus and minus signs (+ and -) to include or exclude words in the search.

This would allow you to use query terms such as '+apple -macintosh' to find documents containing the word apple, but not the word macintosh.

For all the possible operators, have a look at the MySQL online manual at  
<http://dev.mysql.com/doc/refman/5.1/en/fulltext-boolean.html>

## Sphinx

MySQL's built-in fulltext search is only available for MyISAM tables. In particular, InnoDB tables do not support fulltext indexing. If you cannot or do not want to use MyISAM, have a look at Sphinx—another open source, free fulltext search engine that was designed to integrate nicely with MySQL. You can find more information at <http://sphinxsearch.com/>.

### See also

- ▶ [Removing indexes from tables](#)

## Creating a normalized text search column

Usually, regular and fulltext indexing as supported by MySQL are sufficient for most applications. There are, however, situations where they are not perfectly usable:

- ▶ InnoDB tables cannot use fulltext indexes. At the time of writing there were no plans of this changing in the foreseeable future.
- ▶ There are different ways to spell the search terms

Especially in non-English speaking countries, a problem often arises that does not occur as frequently in English environments. Words in the English language consist of letters ranging from A to Z without diacritics. From a software development perspective this is a well-known simplification because it allows for simpler implementations.

One problem you are often faced with German, for example, is different ways to spell the same word, making it complicated to formulate suitable search terms.

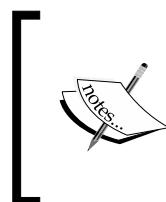
Consider the German words "Dübel" (dowel) and "Mörtel" (mortar). In a merchandise management database you might find several variants of similar products, but each one is spelled in different ways:

productID	name	stock
12352323	DÜBEL GROß 22	76
23982942	"Flacher-Einser" Mörtel	23
29885897	DÜBEL GROSS 4	44
83767742	Duebel Groß 68	31

## *Indexing*

As an end user of the corresponding application searching for those becomes cumbersome because to find exactly what you are looking for you might have to attempt several searches.

In this recipe, we will present an idea that needs some support on the application side. This will allow you to use simple regular indexes to quickly search and find relevant records in situations like the above.



To implement the ideas presented in this recipe, modifications to the application software accessing the database as well as the table definition will be necessary. We advise that this is a process that usually entails a higher complexity and increased testing efforts than simply adding an index.

## **Getting ready**

To implement the ideas presented here, you will need to connect to the database server using your administrative account for the schema modifications. Because apart from the schema modifications application program code changes will be necessary as well, you should be an application developer.

In the example, we are going to assume a table definition as follows:

```
CREATE TABLE products (
    productID int(11) NOT NULL,
    name char(30) default NULL,
    stock int(11) default NULL,
    PRIMARY KEY  (productID)
) ENGINE=InnoDB
```

## **How to do it...**

1. Connect to the database server using your administrative account and make the default schema:

```
use test;
```

2. Add a new column `norm_name` to the `products` table:

```
mysql> ALTER TABLE products ADD COLUMN norm_name CHAR(90)
      name;
```

The column needs to be at least as long as your original column. Depending on the character mapping rules you are going to implement, the projected values may take up more space.

- 
4. Optionally, consider dropping an existing index on the original column. Also modifying other indexes currently containing the original column to include one instead.

Implement the replacement algorithm depending on your language. For German language substitutions, the following substitutions could be used. This is an excerpt from the `Transformers.java` class you can find on the book's website.

```
private static String[] replacements = {
    "ä", "ae",     "null", "0",      ":", "", 
    "ö", "oe",     "eins", "1",      ":", "", 
    "ü", "ue",     "zwei", "2",      ".", "", 
    "ß", "ss",     /* ... */   "-", "", 
    " ", "",       "neun", "9",      ",", "", 
    // ... further replacements...
};
```

5. Modify your application code to use the new mapping function and issue queries against the new `norm_name` column where previously the original `name` column was used. Depending on how you decide to expose the search features to your users, you might want to make searching the new or the old column an option.
6. Modify your application code to update the new column parallel to the original one. Depending on the application scenario, you might decide to only update the normalized search column periodically instead.
7. Before handing out a new version of your software containing the new code, make sure the normalized search column gets populated with the correct values.
8. Optionally, declare the new column **NOT NULL**, after it has been initially filled.

## How it works...

By implementing the mapping algorithm, we make the application think about the different ways to spell things, not the end user. Instead of creating all possible variants, which would become a large set of permutations depending on the length and content of the original input, we project the search terms to a normalized form for both the original data and for queries issued against it. As both use the same mapping functions, only a single supported—query against MySQL is needed. The application of course usually never needs to know these internals. The person in front of the computer will just be pleased to find the right records easily.

The mapping rules from input to search terms depend on the language and application specific needs. For German words, they are rather short—only the umlaut characters need to be transformed to a normalized form. Other languages might require more complex rules.

## *Indexing*

In the example code earlier, we also transform the input to lowercase and remove special characters like dashes and colons, and also the whitespace. For the sample of products we used, this is the result of the transformation:

productID	Name	name_nrm	stock
12352323	DÜBEL GROß 22	duebelgross22	76
23982942	"Flacher-Einser" Mörtel	flacher1ermoertel	23
29885897	DÜBEL GROSS 4	duebelgross4	44
83767742	Duebel Groß 68	duebelgross68	31

Now instead of querying the original data column, we ask the database to search the transformed representation of the search terms in the additional `norm_name` (norm column. For this it can use regular indexes and provide results quickly and efficiently.

Note that the `Transformer.java` code available from the book's website is now of production quality but only serves for demonstration purposes. It does not, for example, have any error checking or exception handling and the mapping algorithm is very simple,

### **There is more...**

If you do not care about international specialties but still want to improve user experience by allowing for less strict searches, you might want to have a look at the `SOUNDEX()` function. It is designed to work for English language words only and allows you to query for words that sound like the search terms.

However, note that the results of using it may not be what you expect—opinions on its use on the Internet range from extreme enthusiasm to complete disappointment. You can find the MySQL documentation at [http://dev.mysql.com/doc/refman/5.1/en/string-functions.html#function\\_soundex](http://dev.mysql.com/doc/refman/5.1/en/string-functions.html#function_soundex).

## **Removing indexes from tables**

Once-useful indexes may become obsolete as requirements change with the evolution of the database. In this chapter, we will show you how to get rid of the `IDX_author` index created in the *Adding indexes to tables* recipe.

### **Getting ready**

Connect to the database server with your administrative account.

## How to do it...

1. Change the default database to library:

```
USE library;
```

2. Drop the `IDX_author` index using the following command:

```
ALTER TABLE books DROP INDEX IDX_author;
```

## How it works...

Using the `ALTER TABLE` statement, we tell the database that we want to remove (`DROP`) index named `IDX_author` from the `books` table.

## There's more...

As with the creation of new indexes, you can drop multiple indexes at once using the `ALTER TABLE` statement by simply adding more `DROP INDEX` clauses, separated by commas. If you were to delete both indexes defined in *Adding indexes to tables*, you could use this:

```
ALTER TABLE books DROP INDEX IDX_author, DROP INDEX IDX_title;
```

## See also

- ▶ *Adding indexes to tables*

## Estimating InnoDB index space requirements

While indexes might very well be the single most important key in database performance tuning, they come at the price of redundancy.

There are two main disadvantages tightly connected to redundant data storage:

- ▶ The danger of inconsistencies between the redundant copies of data that are not at all times identical.
- ▶ Increased storage and memory consumption because the same data is physically duplicated.

Fortunately, the former is a non-issue with indexes. As the database server takes care of keeping data and indexes consistent without human intervention, you cannot get into a situation where two columns that should contain equal data at all times are out of sync.

## *Indexing*

---

In contrast to that, there is no way to prevent the latter disadvantage. We need to store multiple copies if we want different sort orders for quick lookups. What we can do, is to attempt to minimize the negative effect by trying to limit the amount of duplicate information as far as possible.



The employees database is an open source test database available for free. It contains examples for many MySQL features including large tables, foreign key constraints, views, and more. It can be found along with some documentation at <http://dev.mysql.com/doc/employee/en/employee.html>.

In the example below, we assume the existence of the employees test database with employees table defined as follows:

```
CREATE TABLE employees (
    emp_no int(11) NOT NULL,
    birth_date date NOT NULL,
    first_name varchar(14) NOT NULL,
    last_name varchar(16) NOT NULL,
    gender enum('M','F') NOT NULL,
    hire_date date NOT NULL,
    PRIMARY KEY (emp_no)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

We will add an index each to the last\_name and the first\_name columns and predict the necessary space.



Please note that the results will never be exact. The storage requirements especially of text-value table columns (**VARCHAR**, **TEXT**, **CHAR**, and so on)—can be difficult to determine because there are multiple factors that influence the calculation. Apart from differences between storage engines, an important aspect is the character set used. For details refer to the online manual for your server version: <http://dev.mysql.com/doc/refman/5.1/en/storage-requirements.html>.

Moreover, it is not possible to find out the exact size even for existing indexes because MySQL's SHOW TABLE STATUS command only gives approximate results for InnoDB tables.

## Getting ready...

Connect to the database server with your administrative account.

## How to do it...

- Find out the size of one primary key entry.

To do so, look at the primary key definition in the table structure. Add the primary key columns as documented in the MySQL Online Manual. In the table `customer`, the INT column takes 4 bytes. Write this number down.

- Determine the size of each column to be included in the new indexes and add them up per index. In the example, both `first_name` and `last_name` are VARCHAR columns—this means their lengths are not fixed as with the INT type. For simplicity, we will assume completely filled columns, meaning 14 bytes for `first_name` and 6 bytes for the `last_name` column.
- For each index, add the lengths of all relevant columns and the size of the implicit primary key. In our example, this gives the following results:

Index	Column size	Primary Key Size	Index Record Size
IDX_FIRST_NAME	14	4	18
IDX_LAST_NAME	16	4	20

The rightmost column contains the pure data size of a single index record, not including the overhead taken by InnoDB to internally organize the implicit primary key.

- Multiply the size per index record with the number of rows in the table:

Index	Rows	Index record size	Est. index size
IDX_FIRST_NAME	300024	18	5400432
IDX_LAST_NAME	300024	20	6000480

The rightmost column contains the estimated size of the index, based on the number of records, and the overhead taken by InnoDB to internally organize the data.

*Indexing***How it works**

In the previous steps, we simply added up the sizes of all columns that will form a index entry. This includes all columns of the secondary index itself and also those primary key because, as explained in the chapter introduction, InnoDB implicitly adds every index.

Internally, the server of course needs a little more than just the raw column content. Management overhead (such as column widths, information on which columns are indexed, as well as some constant overhead) add to the required space. Calculating these internal requirements is complicated and error-prone because they rely on many parameters and implementation details can change between MySQL versions. This is not required, however, because it is a ballpark number. As table contents often change quickly, exact numbers would not be valid for long.

You can see this in our example—the values are too low. In reality, you will need to multiply these values by a factor of 1.5 to 2.5.

You will find that depending on the lengths of the columns indexed and those that are part of the primary key, the accuracy of the estimates can vary.

**There's more...**

Predicting space requirements is not an exact science. The following items are intended to give some more hints on what you might want to think about.

**Considering actual data lengths in your estimate**

When adding an index to an existing column, you can try to use the average length of the column values:

```
SELECT AVG(LENGTH(first_name)) AS avg_first, AVG(LENGTH(last_name)) AS avg_last FROM employees;
```

For the sample data the results are:

<b>avg_first</b>	<b>avg_last</b>
6.2157	7.1539

Round this up to the next integer (7/8). Note that especially for short columns like first\_name and last\_name, these estimates can be much less reliable because relative to internal database overhead, the declared maximum length of the columns is less significant. This is why in the recipe earlier we went with declared maximum lengths instead.

## Minding character sets

For columns storing text information—such as CHAR and VARCHAR, VARBINARY, and TEXT—the storage requirements depend on the character set used for the text instead. For most English-speaking countries, this is something like the Latin-1 character set, which uses a single byte per character of text. However, in international environments, this is hardly sufficient. To accommodate German text, for example, you need some special characters—not to mention Chinese, Japanese, or other non-Latin languages.

MySQL supports different character sets on a per column basis. However, often you choose a default character set for a database including all its tables and their columns.

When estimating index (and data) sizes for Unicode-aware columns (MySQL supports UTF-8 and UCS2 character sets for this purpose), you need to take into account that they may require more than a single byte per character. The very popular UTF-8 encoding uses between 1 and 4 (even though 4 are only used in very special cases) bytes per character. UCS2 has a constant size of 2 bytes per character. For details on how UTF-8 works see <http://en.wikipedia.org/wiki/UTF-8>.

## Using prefix primary keys

In this example we will add indexes to two tables that are almost identical. The only difference will be the definition of their primary keys. You will see the difference in space consumption for secondary indexes between a regular full column primary key and a prefix primary key. The sample table structure and data are designed to demonstrate the effect very clearly. In real-world scenarios the effect will most certainly be less severe.

### Getting ready...

Connect to the database server with your administrative account.

### How to do it...

1. Download the sample script for this chapter from the book's website and save it to your local disk. In the example below, we will assume it is stored in /tmp/idxsizeestimate\_sample.sql.
2. Create a new database and make it the default database:
 

```
CREATE DATABASE pktests;
USE pktests;
```
3. Import the sample data from the downloaded file. When done, you will be presented with some statistics about the two tables loaded. Note that both tables have the same structure, but one has a full column primary key and the other has a prefix primary key.

*Indexing*

4. Now with the sample tables present, add an index to each of them:

```
ALTER TABLE LongCharKey ADD INDEX IDX_PAY_10(Payload(10))
ALTER TABLE LongCharKey10 ADD INDEX IDX_PAY_10(Payload(10))
```

5. Display the data and index sizes of the tables now:

```
SHOW TABLE STATUS LIKE 'LongCharKey%';
```

6. Add another index to each table to make the difference even more evident:

```
ALTER TABLE LongCharKey ADD INDEX IDX2_PAY_10(Payload(10))
ALTER TABLE LongCharKey10 ADD INDEX IDX2_PAY_10(Payload(10))
```

7. Display the data and index sizes of the tables again and compare with the values:

```
SHOW TABLE STATUS LIKE 'LongCharKey%';
```

Name	Rows	Data Length	Index Length	Index/Data
LongCharKey	50045	30392320	28868608	94.99%
LongCharKey10	50045	29949952	3178496	10.61%

With the second index added, the difference in index length becomes even more evident.

## How it works...

Executing the downloaded script will set up two tables with the following structures:

```
CREATE TABLE `LongCharKey` (
  `LongChar` char(255) NOT NULL,
  `Payload` char(255) DEFAULT NULL,
  PRIMARY KEY (`LongChar`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
CREATE TABLE `LongCharKey10` (
  `LongChar` char(255) NOT NULL,
  `Payload` char(255) DEFAULT NULL,
  PRIMARY KEY (`LongChar`(10))
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

The two tables are almost identical, except for the primary key definition. They are both populated with 50,000 records of sample data.

The tables are populated with exactly the same 50,000 records of pseudo-random data. The Payload column is filled with sequences of 255 random letters each. The LongChar column is filled with a sequential number in the first 10 characters and then filled up to use all 255 characters.

---

```
SELECT LEFT(LongChar,20), LEFT(Payload, 20) from LongCharKey
```

<b>LEFT(LongChar,20)</b>	<b>LEFT(Payload, 20)</b>
<b>0000000000KEAFAYVEJD</b>	RHSKMEJITOVBPOVAGOGM
<b>0000000001WSSGKGMIJR</b>	VARLGOYEONSLEJVTVYRP
<b>0000000002RMNCFBJSTL</b>	OVWGTTSHEQHJHTHMFEXV
<b>0000000003SAQVOQSINQ</b>	AHDYUXTAEWRSHCLJYSMW
<b>0000000004ALHYUDSRBH</b>	DPLPXJVERYHUOYGGUFOS

While the *LongKeyChar* table simply marks the whole *LongChar* column as a primary key over its entire 255 characters length, the *LongCharKey10* table limits the primary key to just the first 10 characters of that column. This is perfectly fine for this table, because the test data was crafted to be unique in this range.

Neither one of the two tables has any secondary indexes defined. Looking at some statistics on the table data shows they are equally big (some columns left out for brevity):

```
SHOW TABLE STATUS LIKE 'LongCharKey%';
```

<b>Name</b>	<b>Rows</b>	<b>Data Length</b>	<b>Index Length</b>
LongCharKey	50045	30392320	0
LongCharKey10	50045	29949952	0

With each index added, the Index Length for the first table will increase significantly, while for the second one its growth is much slower.

In case of the *LongCharKey* table, each secondary index record will carry around with it a complete copy of the *LongChar* column because it is the primary key without limitations. Assuming a single byte character encoding, this means every secondary index record will end up in size by 255 bytes on top of the 10 bytes needed for the actual index entry. **The whole kilobyte is spent just for the primary key reference for every 4 records!**

In contrast to that, the primary key definition of the *LongCharKey10* table only includes the leading 10 characters of the *LongChar* column, making the secondary index entry much shorter and thereby explaining the much slower growth upon adding further indexes.

*Indexing*

## Choosing InnoDB primary key columns

In the chapter introduction we promised to shed some light on how to choose your primary key columns sensibly. Be advised that choosing good primary key columns is an exact science—there are multiple aspects that influence this decision. Depending on your needs and preconditions you will want to prioritize them differently from one to the next. Consider the following as general advice rather than hard rules that must be obeyed unconditionally.

### Getting ready

In order to make reasonable decisions on primary key columns, it is important to have a very clear understanding of what the data looks like in the table at hand. If you already have existing data that is to be stored in an InnoDB table—for example in MyISAM format—it is helpful to compare it with the criteria below.

If you are planning a new schema, you might have to guess about some characteristics of future data. As is often the case, the quality of your choices is directly proportional to how good those guesses are.

This recipe is less strict step-by-step instructions that must be followed from top to bottom, and should be considered a list of properties a good primary key should have, even though you might decide some of them do not apply to your actual environment. As a rule of thumb, however, a column that fulfills all or most of the attributes described below is most likely a sensible choice for a primary key. See the *How it works...* section for details on the individual items.

### How to do it...

- Identify unique attributes:** This is an absolute (technical) requirement for primary keys in general. Any data attribute that is not strictly guaranteed to be free of duplicates cannot be used alone as a primary key.
- Identify immutable attributes:** While not absolutely necessary, a good primary key should never change once it has been assigned. For all intents and purposes, you should avoid columns that have even a small chance of being changed for existing rows.
- Use reasonably short keys:** This is the "softest" criterion of all. In general, longer keys have negative impacts on overall database performance—the longer the key, the more memory it requires. Consider a prefix primary key. See *Using prefix primary keys* earlier in this chapter for more information.
- Preferring single-column keys:** Even though nothing prevents you from choosing a composite primary key (a combination of columns that together form the key), it is generally better to use a single column as the primary key.

- 
5. **Consider the clustered index nature of the primary key** As InnoDB's primary key is also clustered, you should take this special nature into account as well. It can end up read access a lot, if you often have to query for key ranges, because disk access times will be minimized.

## How it works...

In the following sections, we will try to shed some light on what each step of the process is concerned with in a little more detail.

### Uniqueness

An absolute requirement for primary keys is their uniqueness. Every record in your database will have to have a distinct value for primary keys. Otherwise, neither MySQL nor any other database product for that matter could be sure about whether it was operating on the right rows when executing your queries.

Usually, most entities you might want to store in a relational database have some unique characteristics that might be a suitable Primary key. If they do not, you can always assign a so-called surrogate key for each record. Often this is some sort of unique value, either generated by an application working on top of the database or MySQL itself using an **AUTO\_INCREMENT** column.

### Immutability

Primary key columns should generally be (virtually) immutable, that is, under no circumstances should you have to modify their values, once they are inserted into the database.

In our books example, the ISBN number cannot be changed once a book has been published. The same would apply for a car's chassis number.

Technically, of course, they can be changed after their creation. However, this will be very difficult to perform in practice, once the original value has been used to establish relationships between tables. In these cases, you will often have to revert to completely even unsafe methods (risking data inconsistencies) to perform the changes.

Moreover, as the primary key is stored as a clustered key in InnoDB, changing its value will require the whole record—including all columns—to be moved to its new location on disk, causing additional disk I/O.

Note that sometimes columns that may at first seem constant over time really are not. For example, consider a person's social security number. It is designed to be unique and will never change or be reassigned to a different human being. Consequentially, it would not be like a good choice for primary key in a table of people.

## *Indexing*

---

But consider that in most cases data will be entered into the database manually—through forms, text file imports, among others. In some form or another, someone typed it in through a keyboard.

Manual input is by definition an error prone process. So you might end up with a record that has two digits transposed in their primary key social security number without immediately knowing it. Gradually, this wrong value will spread through your database to be used in foreign key relationships, forming complex data structures. When you later learn about the error—for example, because another person who really owns that number is inserted—then you are facing a real problem.

Unless you are absolutely and positively sure a value can never change once it has been assigned to a record, you should consider adding a new column to your table and using a surrogate key, for example, an auto-incrementing number.

## **Key length**

There are several reasons for keys being as short as possible. InnoDB basically operates on a single large heap of memory—the buffer pool—for its caching purposes. It is used for both data and index data, which are stored as memory cached copies of individual pages stored in the tablespace data files. The shorter each key value is, the more of them fit into a single data page (the default size is 16 KB). For an index with 16 bytes per index value, a single data page will contain about a thousand index entries. For an index with only 8 bytes per entry, twice as many values can be cached in the same amount of space. So to utilize the benefits of memory-based caching, smaller indexes are better.

For the data record as a whole there might not be much of a difference between 8 bytes compared with the overall record length. But remember (or refer to the chapter introduction if you don't) that the primary key length is added to each secondary index's length. For example, a secondary index on an 8 byte field will actually be 16 bytes long if the primary key also has 8 bytes per entry. A 16 KB data page would provide space for roughly 1,000 entries in this scenario. If the primary key is 16 bytes long, it would only be sufficient for 680 entries, reducing the effectiveness of cache memory.

## **Single column keys**

Depending on the data you intend to store in an InnoDB table, you might consider using a composite primary key. This means that no single column's value alone uniquely identifies a single record but only the combination of several independent columns allows unique identification. From a technical point of view, this is perfectly feasible and might even be a good idea from a semantic point of view.

---

However, you should very carefully weigh the alternatives because composite keys become a burden. The more secondary tables define foreign key relationships with using a composite primary key, the more complicated your queries will become because whenever you join the two, you have to define a join condition on at least four columns. In addition, as you add more complex queries with multiple joins, this quickly becomes very hard to maintain, therefore, carries a great risk of errors that might be hard to find.

In addition, you also have to consider the increased size of each key entry and that of all the participating columns must be added.

As general advice, you should definitely consider using a surrogate key when you can't find any candidate that fulfills the other criteria just discussed.

## **Clustered Index**

As data is physically stored on disk in the order of the clustered key, similar key values are stored in neighboring locations. This makes clustered indexes very efficient for queries that range over ranges of records by this key. If, for example, the clustered key is a timestamp of some sort, retrieving all records within a contiguous timespan is likely to require relatively little disk I/O because ideally all requested result rows are stored in the same data page, only needing a single read operation (which might even be cached). Even if multiple pages had to be read, this will only require a sequential read operation, which leverages read performance best.

Unfortunately, InnoDB does not allow a non-primary key to be clustered—other DBMSes do, so you have to weigh the alternatives and maybe live with a compromise when deciding on a primary key for your InnoDB tables.

## **Speeding up searches for (sub)domains**

In a column with domain e-mail addresses, searching for all addresses of a given domain is a non-trivial task performance-wise. Given the following table structure, the only way to search for addresses @gmail.com is to use a LIKE query with a wildcard:

```
SELECT * FROM clients WHERE email LIKE '%@gmail.com';
```

Of course, storing the address and domain parts in separate columns would solve this particular problem. But as soon as you were asked for a quick way to find all client e-mail address from a British provider, you would be out of luck again, resorting to:

```
SELECT * FROM clients WHERE maildomain LIKE '%.co.uk';
```

Both queries would cause a full table scan because no index can support the wildcard at the beginning of the search term.

## *Indexing*

In this recipe, you will be given a simple but effective approach to enable the use of indexes for both of the problems just presented. Notice that you will need to make minor adjustments to the queries sent against the database. This might involve some code adjustments in your application.

### **Getting ready**

To implement the ideas presented here, you will need to connect to the database system with your administrative account for the schema modifications. Apart from the database modifications, application program code changes will be necessary as well and you will need to contact an application developer.

### **How to do it...**

1. Identify which column is currently used to store domain-related data. In this example we will be using the `maildomain` column of the `clients` table.
2. Update this column and reverse the contents of the field like this:  

```
UPDATE clients SET maildomain=REVERSE(maildomain);
```
3. If not already set up, add an index to the column:  

```
ALTER TABLE clients ADD INDEX IDXR_MAILDOMAIN(maildomain);
```
4. Change all queries in your application as follows:

Before:

```
SELECT name, maildomain FROM clients WHERE maildomain LIKE '%.co.uk';
```

After:

```
SELECT name, REVERSE(maildomain) AS maildomain FROM clients
WHERE maildomain LIKE REVERSE('%.co.uk');
```

The point here is to reverse the search condition as well as the column in the WHERE clause. SELECT statements using the star placeholder instead of column names can also be rewritten to reverse the `maildomain` column.

### **How it works...**

Indexes are designed to speed up queries by sorting the relevant column contents. This makes finding records with a given search prefix easy.

Searching for all people whose name starts with an "S", for example, is supported by the index. The query does not need to scan the entire table to find the results.

---

Domain names are a different story, however, because those belonging together do not share a common prefix but suffix. There is no immediate way of telling MySQL to create an index supporting this kind of data.

The first idea that comes to mind to work around this would be to use a query along the lines of:

```
SELECT * FROM clients
WHERE REVERSE(maildomain) LIKE 'ku.oc.%';
```

Unfortunately, MySQL—in contrast to other DBMS—can neither use indexes in conjunction with functions like `REVERSE()` nor create an index based on a function in the first place. Instead, it resorts to full-table scans to find the results as soon as it encounters a function call applied to a column in a query's `WHERE` clause. In this case, the `REVERSE()` function is applied to the `maildomain` column.

With a minor adjustment to the way data is stored, this limitation can be alleviated by storing the data backwards in the first place!

When inserting new data into the table, we reverse it first:

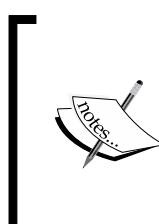
```
INSERT INTO clients (maildomain, ...)
VALUES (REVERSE('example.co.uk'), ...);
```

When retrieving data later, we just need to reapply the same function to get back at the original data:

```
SELECT REVERSE(maildomain) FROM clients
WHERE maildomain LIKE REVERSE('%.co.uk');
```

As now the query condition does not contain a function call on a *column* anymore, we are happy to use an index on the `maildomain` column to speed up the search.

It might seem odd at first that now even with two calls to the `REVERSE()` function, MySQL can in fact use an index.



The key point is that MySQL does not have to apply the function on any *column data* but only on the *constant* condition (the '`%.co.uk`' string). This means that MySQL can skip the first `REVERSE()` function call entirely and later—when the rows have already been fetched—on the already retrieved data, apply the second `REVERSE()` function call to the `maildomain` column. Both of these are not a problem for MySQL's optimizer and allow for index use.

The query is really executed in two phases. In the first phase, MySQL will have a look at the `WHERE` condition and check if it can replace any function call with constants. So, when we run the following query:

```
SELECT REVERSE(maildomain) FROM clients
```

## *Indexing*

after the first phase, internally the query looks like this:

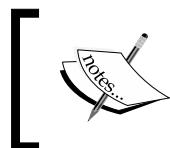
```
SELECT REVERSE(maildomain) FROM clients
WHERE maildomain LIKE 'ku.oc.%';
```

In this query, there is no function call left in the condition. So the index on the `maildomain` column can be used, speeding up the execution as desired.

## **There's more...**

If your application typically issues queries that need to retrieve contiguous ranges of domains—as in the preceding example—you might consider using the reversed domain name as primary (and therefore clustered) key.

The advantage would be that the related records would be stored closely together on the same or adjacent data pages.



However, updating an existing table on its primary key column can be highly time consuming, as all data rows need to be physically rearranged. This is sometimes complicated to do when foreign key constraints are in place.

## **See also**

- ▶ *Choosing InnoDB primary key columns*

## **Finding duplicate indexes**

Over time database schemata are subject to changes such as index additions and deletions. It is not uncommon to end up with multiple indexes that are equivalent in terms of execution but might be defined with different names or even different columns.

This duplication of indexes has negative consequences for your database:

- ▶ **Increased size:** The more the indexes, the bigger the database.
- ▶ **Lower performance:** Each index has to be updated on modifications of the table, wasting precious I/O and CPU resources.
- ▶ **Increased schema complexity:** Schema maintenance and understanding of tables and relationships gets more complicated.

For those reasons, you should be concerned about superfluous indexes.

In this recipe, we will present a way to quickly find out which indexes can be dropped.

## Getting ready

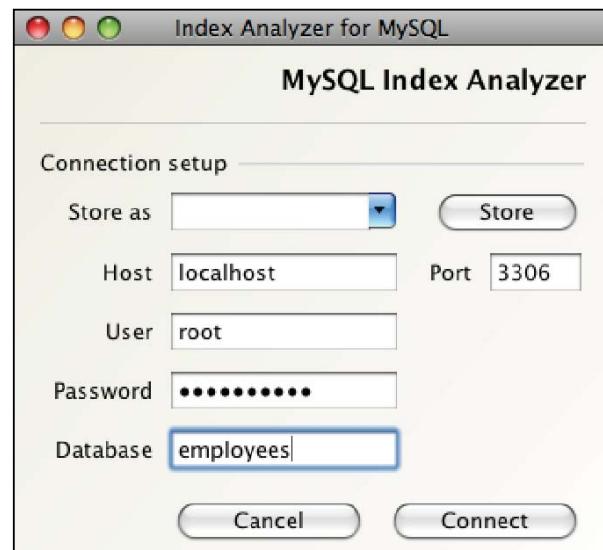
In order to run the program presented here, you will need a Java Runtime Environment (just Java) installed. You can download it for free from <http://www.java.com>.

Download the Index Analyzer for MySQL from the book's website.

You will also need login credentials with administrative privileges for the server and database you want to analyze.

## How to do it...

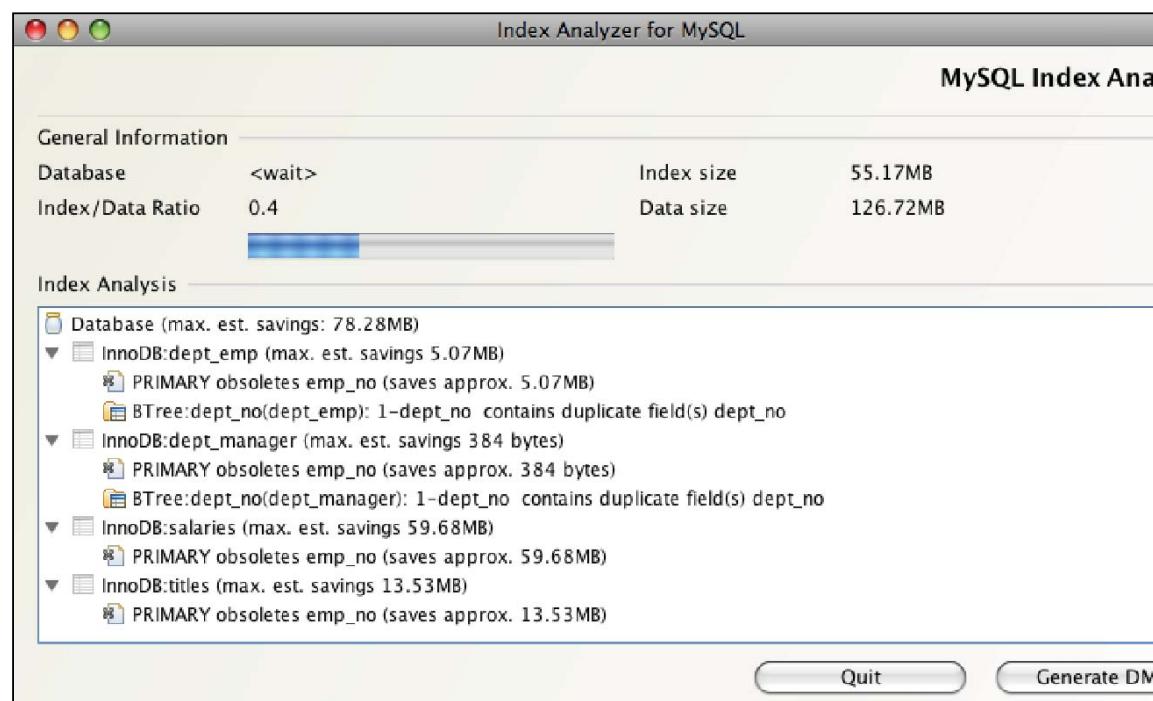
1. Launch the downloaded application by double-clicking its icon. The connection window will appear.



2. Enter the connection data for your MySQL server and specify the database. If you like, you can store these settings for later use.
3. Hit the **Connect** button. The analysis will begin. Stand by—this might take two, depending on the number of tables, columns, and indexes in that database.

## *Indexing*

- When the analysis is complete, review the proposed changes the tool makes from the tree-like display, you can use the Generate SQL button to copy ALL TABLE statements to either the clipboard or a file that will apply the changes suggested to the database.



Make sure you do not just blindly execute the proposed statements against your database!



You must always carefully review anything that an automated tool suggests you do to your data. No program can replace your professional judgment about whether or not an index is obsolete or required for some specific reason beyond the computer's understanding.

## **How it works**

The Index Analyzer for MySQL tool connects to your database and retrieves information about the indexes defined in the database you specified. It then checks for indexes redundant compared with one or more of the others. It will detect the following situations:

- Two indexes are completely identical.
- One index is a prefix of a second longer one. As MySQL can use the second index for the same queries (ignoring the superfluous columns) the shorter index is redundant.

---

An index on an InnoDB table is defined so that it ends with the primary key column. MySQL internally appends the primary key columns, they should be removed from the definition. The tree display's root node is the database you selected, followed by the tables with redundant indexes. For each table, one or more detail nodes describe the analysis results in terms of which index is made obsolete by which other.

Each node also contains a rough estimate on how much space could be saved by removing a redundant index. Note that this is just a ballpark figure to get an idea. In the example, the actual savings are lower according to the statistics MySQL offers via the SHOW STATUS command:

<b>Table / Index</b>	<b>Index Size before</b>	<b>Estimated Savings</b>	<b>Actual Savings</b>
dept_emp / emp_no	10MB	5.5MB	4.5MB
dept_manager / emp_no	32k	384 bytes	16k
Salaries / emp_no	34.6MB	59.7MB	35MB
Titles / emp_no	11MB	13.5MB	11MB

All tables were defined with an extra index on the `emp_no` column, which was made by the primary key. Note that the difference between estimated and actual savings is significant. This is because MySQL estimates are based on multiples of the data page—KB—while the Index Analyzer application uses average column lengths.

## There's more...

Apart from the Index Analyzer for MySQL available from this book's website, there are other tools available for the same purpose as well. If you do not want to, or cannot, install the MySQL Runtime Environment, you might be more content with Maatkit's *mk-duplicate-key*. It is a free command-line tool based on Perl and can be used on a variety of platforms. You can get it from <http://www.maatkit.org> including the full documentation.



# To

In this chapter, we will discuss about:

- ▶ Transferring connection settings between different machines using a network
- ▶ Sorting MySQL GUI Tools' stored connections
- ▶ Automatically creating stored connections
- ▶ Adding custom graphs to MySQL Administrator
- ▶ Displaying query results page by page and with scrolling, using the `mysql` command-line client
- ▶ Extracting information from verbose output using the `mysql` command-line client
- ▶ Specifying a default pager
- ▶ Using a custom prompt to distinguish connections
- ▶ Encrypting a MySQL server connection with SSH
- ▶ Creating an encrypted MySQL console via SSH
- ▶ Using a PuTTY template connection for SSH secured connections

## Introduction

Everyone expects a DBA to keep database servers running smoothly day in and day out, handing out data to maybe thousands of clients or even more simultaneously, quickly and reliably.

Apart from a solid knowledge about the inner workings of the server(s) you manage, just as important are the tools at your disposal. They enable you to inspect and tune the server's running parameters, configuration options, and so on.

## Tools

While there is a plethora of utilities and tools, both free and commercial—each of them with their respective strengths and weaknesses—the official tools available directly and (mostly) from the MySQL website are often underestimated or even overlooked.

While they may not be the most polished and may certainly have their quirks, they nevertheless, not escape your attention because either on their own or sometimes by some other (free) software, they are capable of helping you a great deal with your MySQL administration tasks.

### Tools used in this recipe

In this chapter, we will look at the MySQL command-line client `mysql`, which is available for free with the server, and the MySQL GUI Toolkit parts *MySQL Query Browser* and *MySQL Administrator*.

The command-line client lies at the heart of many scripts and other command-line tools. This is because it is quite flexible and lends itself to automation by its very nature as a text-based tool. It can generally be used on all platforms supported by MySQL. However, the respective underlying operating system and the shell used play an important role in what you can do with `mysql`.

Most Linux distributions and other Unix-like systems—this includes Apple's Mac OS X—will default come with rich support for scripting and automation. On Microsoft Windows, support has become better over the years. However, some funny things that you get for free with the software and the like will just not work. In some cases, such lack of functionality can be alleviated by the installation of some additional packages. In other cases, Windows users, unfortunately, are simply out of luck due to restrictions posed by the operating system's very core.

Nevertheless, wherever possible we will give advice on how to work around such problems should the need arise in any of the following recipes.

MySQL GUI Tools have become a respectable set of tools over the past couple of years. Although sometimes they still have issues with stability, they have reached a level where they can be recommended for everyday use without hesitation. In this chapter, you will find some recipes that revolve around MySQL Administrator and MySQL Query Browser to make using them an even more pleasant experience. You can get them from the MySQL website page at <http://dev.mysql.com/downloads/gui-tools>.

This chapter is not a manual to these tools in general. In fact, some experience will be recommended to take full advantage of the recipes presented here. To follow along, you will need to have the GUI Tools installed on your machine.

## Platform differences

MySQL GUI Tools are available for major operating system platforms. The recipes in them presented in this book should work equally well on Windows, Linux, and Mac OS X. However, there are some differences between these platforms, for instance, where preferences files are stored or what the user interface looks like. Whenever necessary, aspects that need different handling depending on the underlying platform will be discussed separately.

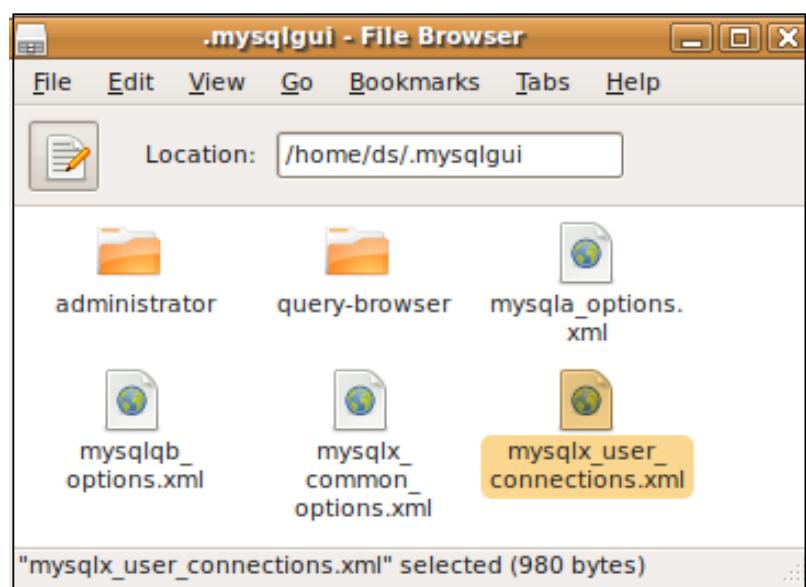
## MySQL GUI Tools config file locations

One major difference between operating systems is the location of the MySQL GUI configuration files. Some recipes manipulate those directly; so instead of describing again where to find them on each operating system, please have a look at the following whenever you need to locate one of them.

In Mac OS X, the preferences files are stored in the Library/Application Support of the user's home directory shown as follows:

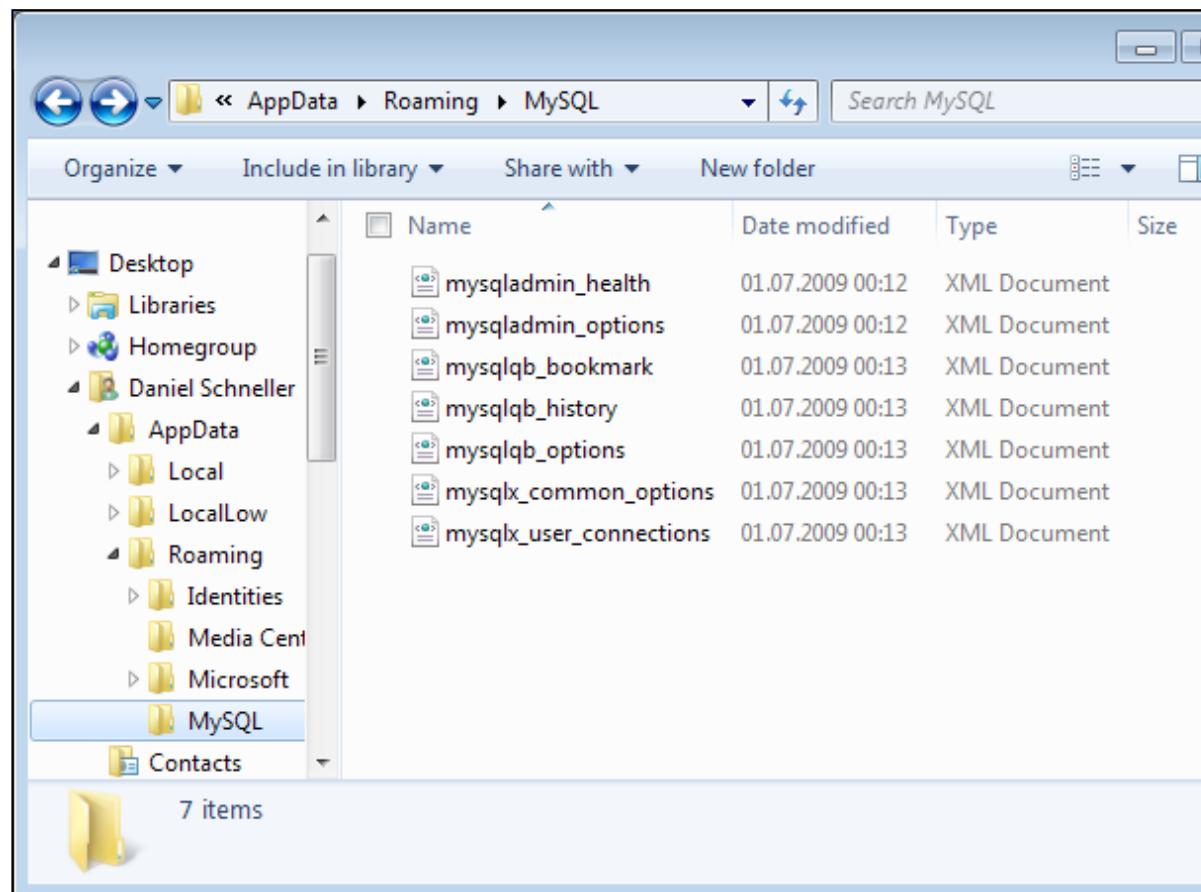


In Linux, the settings are stored in a hidden directory `.mysqlgui` inside the user's home directory.



## Tools

In Windows, the settings are stored in the %APPDATA% /MySQL folder located in the user profile. The next screenshot shows Windows 7, but with XP it is just the same.



## Transferring connection settings between different machines using a network share

MySQL Administrator and MySQL Query Browser allow storing connection profiles for databases that you regularly use. These are stored locally as part of the user profile you are logged in with. Unless you only ever work from a single machine with a single user account, you will usually have to recreate and maintain each and every machine's list of stored connection manually, which is neither a fun nor a productive task.

This recipe will show you how to store connection profile settings in a way that allows them to be used from multiple user accounts and even multiple machines.

The next steps will demonstrate how to share connection settings between Ubuntu and a Windows 7 machine using a network share.

You can apply this to any combination of machines with any supported operating system.

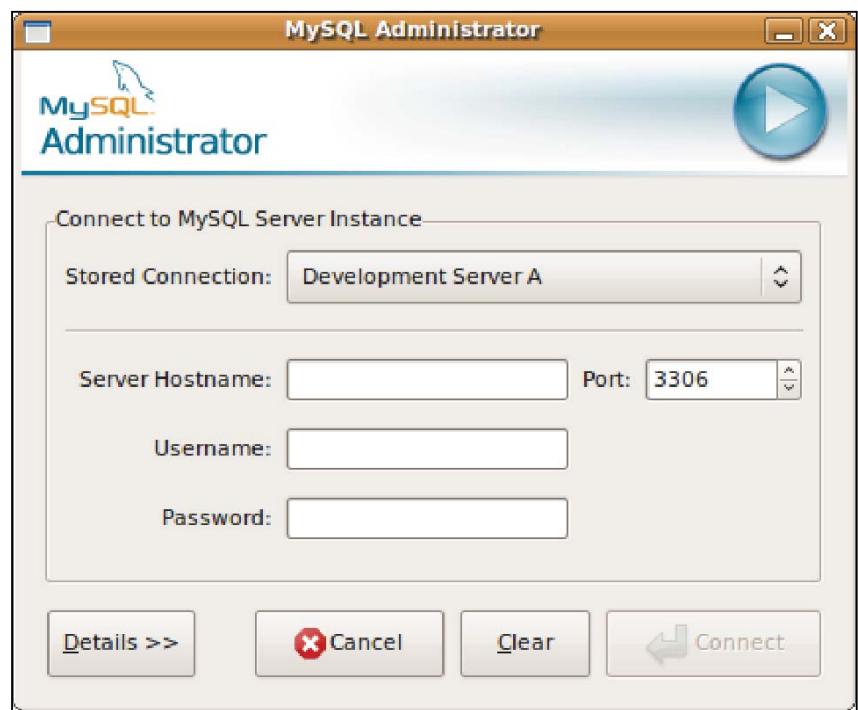
## Getting ready

Make sure you have some sort of a shared medium ready with write permissions. This can be a directory on the local machine accessible by several user accounts, a USB pen drive you carry around with you, a web server, or a network share accessible by anyone you share the connections with.

See the chapter introduction for a guide on where to find the stored connection files for each platform.

## How to do it...

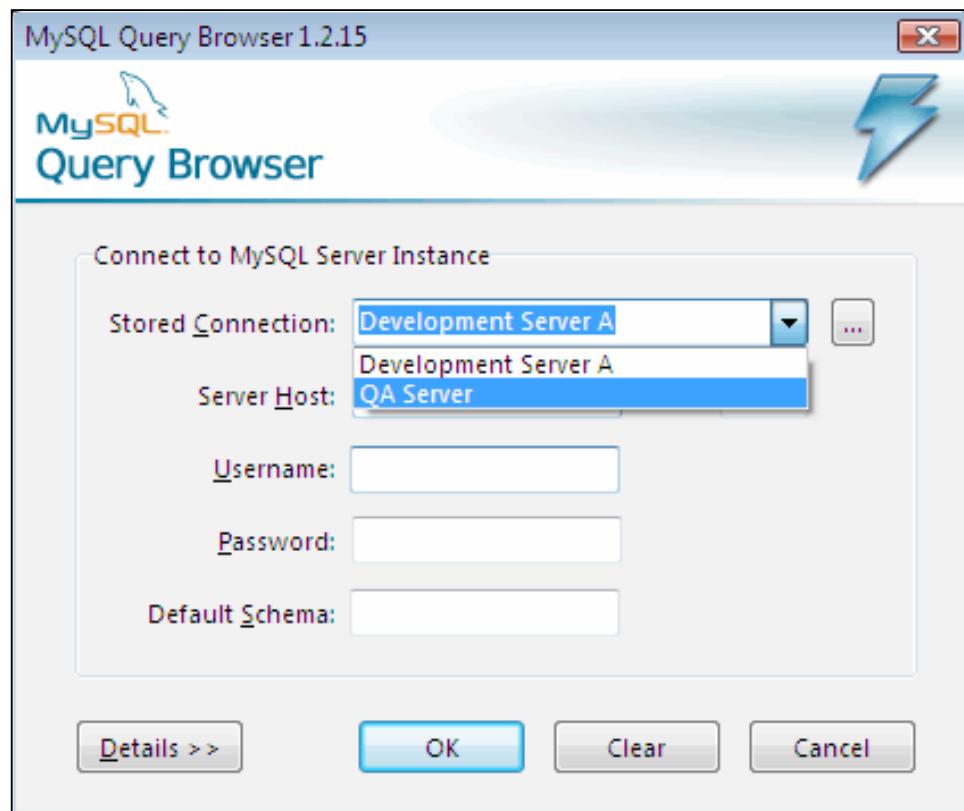
1. Set up one or more connections in MySQL Administrator on the Linux machine. The screenshots in this example were taken on a machine with two connections, which are called **Development Server A** and **QA Server**.



2. Close MySQL Administrator and also MySQL Query Browser (if running).
3. Navigate to `~/.mysqlgui` and copy the file called `mysqlx_user_connections.xml` to the network share.
4. On Windows 7 open an explorer window and enter `%appdata%\MySQL` in the address bar. Click *Enter* to open the folder.
5. Open another Explorer window and point it to the network share.
6. Drag `mysqlx_user_connections.xml` from the network to the local drive, making a copy.

**Tools**

- Start MySQL Administrator or Query Browser and find the connections defined on your Linux machine also present in Windows 7.



## How it works...

MySQL GUI Tools store information about the connections you enter and store using a graphical user interface in an XML file. As there are no other places to consider (such as registry or binary files), it is rather easy to copy them around and use them in different environments.

All the settings—including host names, user account names, and even the notes you add to a stored connection—are saved to this XML file. It is used by both MySQL Administrator and MySQL Query Browser, so you need not do anything to share a common set of connections between the two on the same machine. Passwords can optionally be stored and are saved to this file too, if you choose so. However, storing passwords is generally not recommended because they are not securely encrypted.

Ideally, the tools would optionally allow you to specify which file to use so you could store them all at the common location like the network drive employed in the previous section. While this would work great in theory, there would be all kinds of unwanted consequences. For example, what would you do if two people on different machines tried to modify the same stored connection?

---

Because of this, it is necessary to provide each machine (or user account to be precise) with its own copy of the file. This is done by creating a local copy of the file in the %appdata% portion of the path named in the previous steps takes you to (a subdirectory of the user's profile) with its own copy of the file to work with. This is what MySQL GUI Tools do by default. They create a local copy of the file on each machine. To MySQL GUI Tools it makes no difference whether they created the file or got spoonfed by a different instance, even on a different piece of hardware.

## There's more...

For simple setups and with a limited number of people sharing connection settings, this basic approach works quite well. However, maybe you'd like some more pointers on how to go from here.

### Dealing with changes

As we already discussed, each workstation machine gets its own copy of the connection file. While this is a technical necessity, it brings along the problem of concurrent changes to the same file. If two people try to edit the same connection profile on different machines, one person's edits will be lost because the second one would overwrite their colleague's edits.

As soon as at least two people modify different local copies of the file, they will have to agree on who is to copy his or her new version to the network drive. Only one of them can do this because the second one would overwrite their colleague's edits.

For small teams, the simplest approach could be to agree on a dedicated PC on which all the changes are made. All other machines only update their local copy of the file from the network, but never push them back.

Alternatively, you can name a single user who makes all of the changes and puts them on a shared medium.

For larger teams, you may consider using a version control system like Subversion. This can be especially practical for larger organizations already having a repository in place. The local XML file could be a working copy checked out from the repository. Whenever someone changes it, he or she would check it into the repository for others to fetch. In case of conflicting changes, you would be notified by the version control software and would have to merge both changes into a new combined version.

If you do this, you might consider using some kind of XML formatting tool because MySQL GUI Tools tend to write out the file with platform-specific line endings, making it a little cumbersome compared to the files originating from Windows and Unix-like platforms.

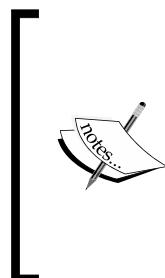
For Windows you might want to check out a program called *firstobject XML Editor* that is available at <http://www.firstobject.com>, which has a formatting feature.

*Tools*

## Sorting MySQL GUI Tools' stored connections

MySQL Administrator and MySQL Query Browser both include a connection editor that allows you to manage your stored connection profiles. While this offers a comfortable way to create new or edit existing profiles including a lot of settings and even comments, this editor has no way to sort your profiles. Especially, if you work with a lot of stored settings, you will have trouble maintaining a certain tidiness to quickly find specific profiles from the drop-down list in the **Connections** dialog.

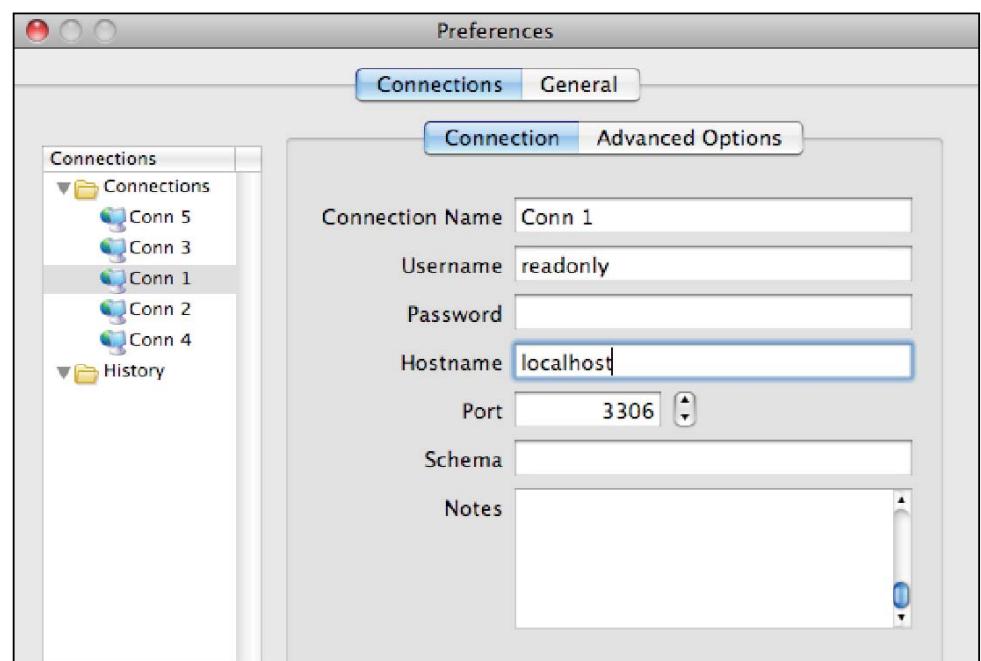
This recipe will show you how to sort connection profile settings by **Connection Name**, assuming Mac OS X as the platform.



For Linux and Windows, the instructions are mostly identical. However, on Mac OS X you will have to replace the path names with the appropriate values for your system. Instead of the terminal application, Windows users use the **Start | Run | cmd.exe** to launch a command interpreter. Also, note that the `cp` command is called `copy` on Windows, so make sure you change that too!

### Getting ready

To try this you will need several stored connection profiles. You can create those in either MySQL Administrator or MySQL Query Browser as they both share the same list of connections. In this example, there are five connections that appear unsorted in the connection editor:



You will need an XSLT processor installed. Most Linux distributions and Mac OS X come preinstalled with a command called `xsltproc`. Windows users can download a beta version of this tool for their platform from <http://www.xmlsoft.org/XSLT>.

## How to do it...

- Save the following code to a file called `sortconnections.xsl` and place it in your home directory. You can download this file from the book's website ([www.packtpub.com](http://www.packtpub.com)) too.

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output omit-xml-declaration="yes" indent="yes"/>
  <xsl:strip-space elements="*"/>
  <xsl:template match="node() | @* ">
    <xsl:copy>
      <xsl:apply-templates select="node() | @* "/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="user_connections">
    <xsl:copy>
      <xsl:apply-templates select="@* "/>
      <xsl:apply-templates select="* [name() != 'user_connections']"/>
      <xsl:apply-templates select="user_connection">
        <xsl:sort select="connection_name"/>
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

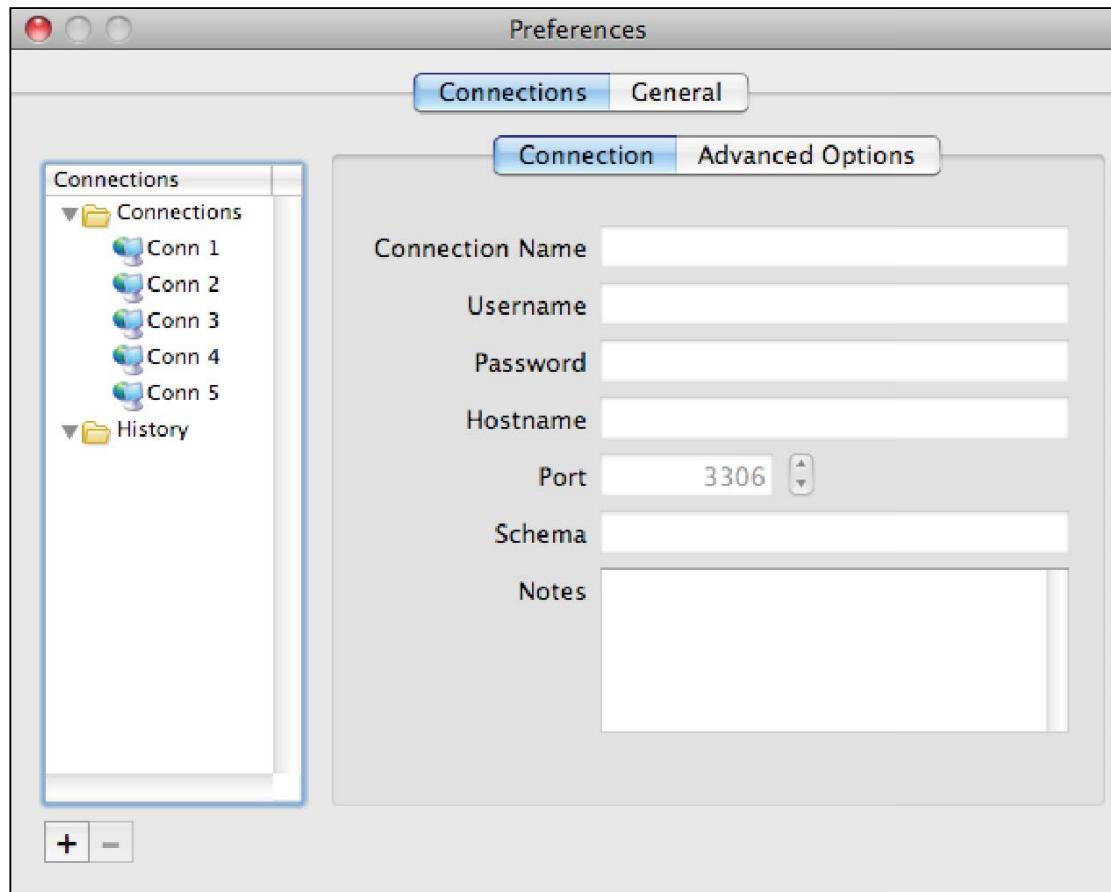
- Close MySQL Administrator and also MySQL Query Browser (if running).
- Open Terminal.app.
- Enter the following commands. The first `cd` command will take you to your directory in case you have configured a different default.

```
$ cd ~/Library/Application\ Support/MySQL/
$ cp mysqlx_user_connections.xml mysqlx_user_connections.unsorted
$ xsltproc ~/sortconnections.xsl mysqlx_user_connections.unsorted > mysqlx_user_connections.xml
```

- Close the terminal.

**Tools**

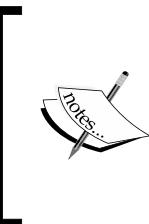
6. Start MySQL Administrator or MySQL Query Browser and find the connection in the connection editor.



## How it works...

MySQL GUI Tools store information about the connections that you enter using the user interface in an XML file. Each connection is represented by an XML element <user\_connection>. The order of these elements in the file determines in which GUI will display them.

As there is no way to rearrange the connection profiles using the user interface we underlying data file.



Because the data file is locked by the GUI Tools while they are running, it is important to quit them before beginning the procedure. Otherwise your changes might get lost, because next time you close the GUI Tools, the file gets rewritten in the order that the running instance knew!

---

Using a simple XSL stylesheet and a suitable processing tool (`xsltproc`), the XML file is read and transformed into a new version with the `<user_connection>` elements sorted by their `<connection_name>` sub-element. To make sure nothing goes wrong, we make a backup copy of the original file and save it as `mysqlx_user_connections.xml` in an unsorted state.

Finally, the `xsltproc` command applies the `sortconnections.xsl` stylesheet to the resulting output and stores it back into `mysqlx_user_connections.xml` where it will be used the next time a MySQL GUI Tool starts.

## There's more...

Sorting by name is just one way of organizing your connection templates and probably not the one most often used. However, XSL stylesheets are a very versatile and powerful means to manipulate XML files such as the connection profiles store. There are lots of resources available online and in printed form that can teach you how to filter, group, and rearrange XML data. For a tutorial on XSL transformations, for example, go to <http://www.w3schools.com/xsl/>.

Here are some more pointers as to what you might do to make stored connections even more useful:

- ▶ Instead of sorting by connection name, you might sort by hostname. This is particularly useful if you manage a large number of machines following a naming convention that allows for sensible sorting.
- ▶ A combination of sort criteria might prove useful if you store several profiles for a single server. You might sort by host first and then by database or username.

Even in cases where the predefined fields are insufficient, there is a way to define arbitrary sort criteria by leveraging the otherwise seldom-used **Notes** field in the query. For example, in a scenario where you have to manage hosts in different locations (countries, subsidiaries, and so on) you could store the city name in the **Notes** field and use this as a sorting criterion.

## Automatically creating stored connections

Although managing a handful of connections is no problem with the built-in connection managers in MySQL GUI Tools, creating a larger series of them tends to become tedious. For example, when managing a shared database server there might be several hundred databases—one or even more for each customer. In a different scenario you may have numerous servers, for each of which you might want a separate stored connection to allow easy access.

## Tools

This recipe presents a way to automatically generate new connection entries based on a simple CSV (comma-separated-value) file using a short shell script. This very simple approach allows for a wide range of data sources. For example, you might already have a spreadsheet with all of the important information or extract it from some sort of database.

Refer to the *MySQL GUI Tools config file locations* section in the chapter introducing MySQL GUI Tools where you can find the connection storage XML file for each operating system.

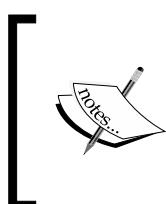
## Getting ready

To try this you should know your way around with a command line. There is no need to be experienced with advanced shell scripting, but you should be familiar with navigating around the file system and some basic commands. Moreover, you will need a file containing the basic information about each connection profile you would like to include in your MySQL GUI Tools stored connections. The steps outlined in this recipe require a bash shell, which is default in many Linux distributions and Mac OS X 10.3 and later.

In Windows, you can either install the Cygwin toolkit from <http://www.cygwin.com/>, which provides a bash shell and sed tool, or modify the script to match the batch file syntax. The sed command can be downloaded as a native Windows executable as part of the Unxutils package from <http://unxutils.sourceforge.net/>. We definitely recommend you have a look at Cygwin, unless you are really experienced with Windows batch scripting.

Connections will be generated from a CSV file, which should contain one line per connection. Each line will contain the connection information in a fixed order separated by commas like this:

```
Title,Username,Hostname,Port,Schema,Notes,Password
```



All fields are optional. However, the number of commas per line must be obeyed, even when nothing more follows in the same line!

Everything that you don't include in this file must either be entered manually each time you choose this connection profile, or be a fixed value that all connections have in common. The following file contains several valid sample lines to demonstrate the idea. The example assumes that you start with an empty list of connection profiles and add these samples directly from this file called `connections.csv`:

```
LocalDev,root,localhost,3310,dev_db,Version 1.5,rootpw
Web01,smith,web01.example.com,3306,cmsweb,,
Austria.guest.10.22.109.12,3306,mz011_Staging_DB,
```

In order to create connections with a password taken from the CSV file, you must make sure that the template connection you are going to create in the following section is configured to use **Plain Text** password storage setting. If you do not want to store passwords in plain text connection profiles for security reasons, you can set this option to any of the other password settings, but make sure that you do not provide any passwords in the CSV file because they would not work.

The example in the next section assumes plain text password storage for demonstration purposes.

## How to do it...

1. Open MySQL Administrator or MySQL Query Browser, and go to the **Connections Editor**.
2. Create a new connection with this data:

Field	Content
Connection Name	p_0
Username	p_1
Password	p_6
Hostname	p_2
Schema	p_4
Notes	p_5

3. Close MySQL Administrator and MySQL Query Browser (if running).
4. Open the connection profiles XML file with a text editor and search for **p\_0**. You most likely find it close to the end of the file.
5. Replace the port number **3306** with **p\_3**. This is necessary because you cannot enter non-numeric characters in the GUI.
6. Save the `<user_connection>` element surrounding your template profile. Copy the `</user_connection>` line at the end to a new file called `oneconnection.xml`. This file should look like this:

```
<user_connection>
  <connection_name>p_0</connection_name>
  <username>p_1</username>
  <hostname>p_2</hostname>
  <port>p_3</port>
  <schema>p_4</schema>
  <advanced_options>
```

*Tools*

```

<notes>p_5</notes>
<connection_type>0</connection_type>
<storage_type>1</storage_type>
<password_storage_type>6</password_storage_type>
<password>p_6</password>
</user_connection>

```

7. Create a script file named `mkuserconn.sh` with the following contents:

```

#!/bin/bash
IFS=
TMPFILE=$(mktemp ~/mkuserconn.XXXXXXXXXX) || exit 1
while read p[0] p[1] p[2] p[3] p[4] p[5] p[6] ; do
    s=""
    for ((i=0; i<${#p[*]}; i++)); do
        s=$s${p[i]}/${p[$i]}/g;""
    done
    newentry=$(sed -e "$s" oneconnection.template)
    echo "$newentry" >> ${TMPFILE}
done < ${2}
echo "</user_connections>" >> ${TMPFILE}
sed -e "/<\user_connections>/ {
    r ${TMPFILE}
    d
}" ${1}

```

8. Make the file executable with:

```
$ chmod u+x mkuserconn.sh
```

9. Invoke the script like this:

```
$ ./mkuserconn.sh mysqli_user_connections.xml connections
```

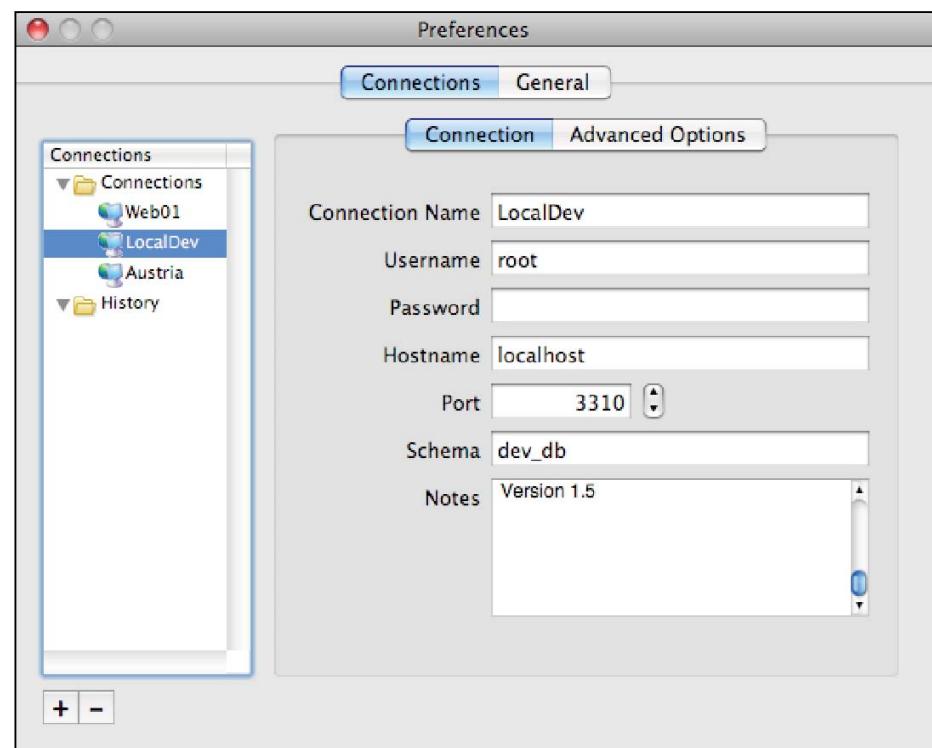
10. The output will be sent to `stdout` to give you a chance to verify that everything is as planned. Once you are content, redirect it to a new file with the redirection operator like this:

```
$ ./mkuserconn.sh mysqli_user_connections.xml connections
mysqli_user_connections_new.xml
```

11. Replace the original `mysqli_user_connections.xml` file with the new `mysqli_user_connections_new.xml` once you are content, saving a copy of the previous version:

```
$ cp mysqli_user_connections.xml mysqli_user_connections.
$ cp mysqli_user_connections_new.xml mysqli_user_connecti
```

- 
12. Open MySQL Administrator or MySQL Query Browser and find the newly created connections ready to go:



## How it works...

MySQL GUI Tools store information about the connections you enter and store using graphical user interface in an XML file. Each connection is represented by an XML element called `<user_connection>`.

The script presented takes one such entry as a template to create new ones, replacing `p_0`, `p_1`, `p_2`, and other similar placeholders with values from the connections.

First the script creates a file in your home directory (~) where the new connection is stored temporarily. It then reads the fields from a file passed in as the second parameter, stores them in an array called `p`. The order of the fields in the file corresponds to the array. In the loop body (the loop reads the CSV file one line at a time), the sed command is assembled, consisting of replace commands that fill in the information from the array at the appropriate places in `oneconnection.template`.

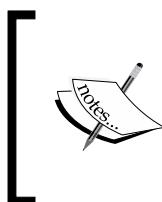
When the command has been built, it is executed, and the output is stored in the temporary file created earlier.

This is repeated until all lines of the CSV file have been read, effectively creating an `<user_connection>...</user_connection>` element for each line in your input file.

Finally, when the temporary file contains all new connection profiles, it is merged into the original XML file.

## Tools

When you open one of the GUI tools again, you will find the newly created connection on the template you set up before. You can, of course, remove the `p_0` connection at any time, once you copied it to the `oneconnection.template` file.



Be sure to try this on a copy of your connections XML file first to make sure everything is correct! A single typo in the script file might corrupt the file beyond repair. This is why we designed the script to output to the console by default to prevent accidental overwrites.

## There's more...

The script presented in this recipe is very basic to make it easier to understand, and it uses basic tools that are easy to get and install almost anywhere.

Of course, you could rewrite it in Perl or any other programming language, adding more features, some help texts, and so on.

Some ideas you may want to pursue could be inserting new connections in some specific order, modifying existing connections with the same name instead of creating duplicate entries, or editing the connection storage XML file in place.

## Adding custom graphs to MySQL Administrator

MySQL Administrator is one of the graphical tools that MySQL provides to manage database servers. Apart from other things like server daemon control and a log file viewer, this tool includes visual controls to display the current load and other "vital signs" of a database server.

Even though the out-of-the-box configuration already contains some useful diagrams, it becomes even more useful with some custom-designed graphs. It might not be suitable to replace a fully featured monitoring solution, but it is definitely helpful to gain a quick impression about what stress a server is currently under, how many users are connected, and so on.

MySQL Administrator provides a graphical editor to introduce new *pages*, *groups*, and *graphs*. Even though I would rather use the term tab (which is what it comes down to in the end), I will stick to *page* here because that's what MySQL calls it.

When you open up MySQL Administrator and head to the **Health** section, you will see the default pages: **Status Variables**, **Connection Health**, **Memory Health**, and **Server Status**.

---

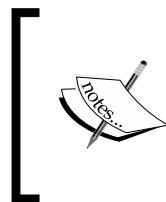
In this recipe, we are going to add a new page called Stats and set it up to display about the number of requests the server has to handle currently, split up by SELECT, UPDATE, and DELETE statements as well as a bar diagram that shows the ratio of read/write operations.

MySQL GUI Tools are not always available for all platforms in the latest version. Us Windows is updated more frequently. This sometimes results in minor differences changes in menu item names and the like, depending on which operating system For example, at the time of writing the most current Windows version was 5.0r17 Mac and Linux it was 5.0r12. While on Windows, the context menu mentioned in t is labeled **Add Page...**, the Mac version calls the same command **New Page**. How should not have any problems moving along.

## Getting ready

To try this you need MySQL Administrator installed. Furthermore, you will need log credentials to a MySQL server instance, preferably a busy one to watch some real course, a locally installed instance will do just as well. Please note that you need s privileges to issue SHOW STATUS commands.

One word of advice before we begin: even though it has gotten much better at this modifying the graphs MySQL Administrator will sometimes just crash and take eve did so far with it. It only saves your modifications to disk when you leave the program.



It is highly recommended to quit the application and restart every on in a while when you have got something working—the way you would to keep it, to prevent losing your freshly made customizations!

In case you experience repeated crashes, you might want to consider editing the g definition manually. Refer to the *How it works...* section for further information.

## How to do it...

1. Start MySQL Administrator and connect to the server.
2. Activate the **Health** section and go to the **Connection Health** page.
3. Right-click on the page and choose **New Page** from the context menu.
4. Enter **Stats** in the newly opened dialog. If asked for it, specify a descriptive like **Statistical breakdown of different query types**.
5. Make sure the new **Stats** page is displayed and then right-click on it. Choos

*Tools*

6. Enter **DML Statements** in the naming box that will appear.
7. Right-click the (empty) **DML Statements** group. Choose **New Graph** from the context menu.
8. Fill up the settings as shown in the following table. Fields not listed should be left empty:

<b>Field</b>	<b>Content</b>
Display Title	Not Checked
Graph Type	Line Graph
Value Formula	$\wedge [com\_select]$
Value Unit	Count
Value Caption	SELECT
Min Value	0
Max Value	100
Auto Extend	Checked

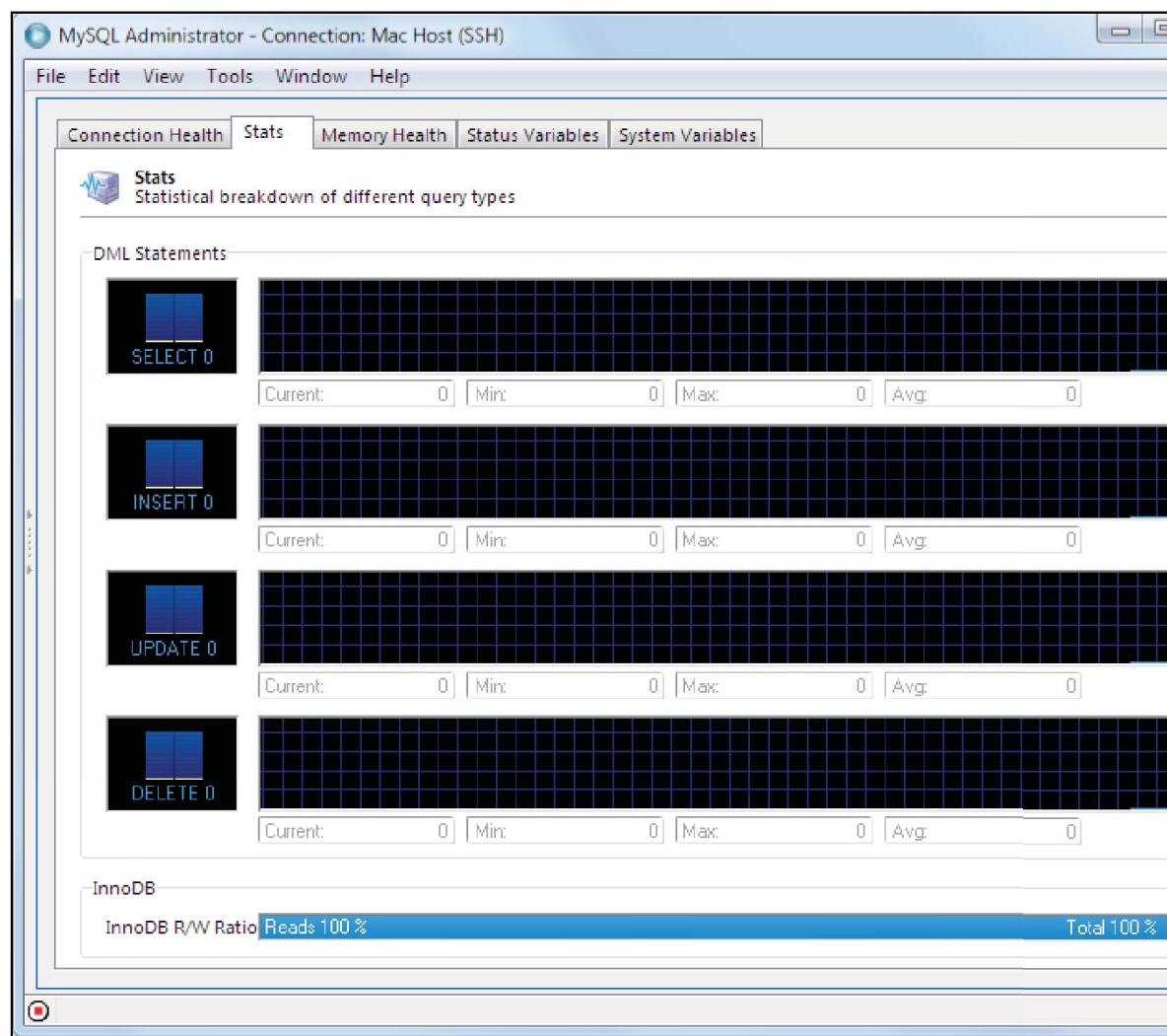
9. Click **OK (Mac) / Apply (Windows)**.
10. Repeat this step for the value formulas  $\wedge [com\_insert]$ ,  $\wedge [com\_update]$ ,  $\wedge [com\_delete]$ , and their appropriate value captions.
11. Right-click on the page and create another group called **InnoDB**.
12. Inside this group add a new graph with these settings:

<b>Field</b>	<b>Content</b>
Title	InnoDB R/W Ratio
Display Title	Checked
Graph Type	Bar Graph
Value Formula	$( [innodb\_pages\_read] / ( [innodb\_pages\_read] + [innodb\_pages\_written] ) ) *100$
Value Unit	Percentage
Value Caption	Reads
Min Value	0
Max Value	100
Max Caption	Total



Do not forget to restart MySQL Administrator to make your changes permanent.

After filling up each field appropriately, as discussed, the final output should look like the following screenshot:



## How it works...

All graphs are based on status variables provided by the MySQL server. The MySQL manual contains an extensive list of these at <http://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html>, documenting the meaning of each variable. The value formulas allow mathematical operations using these values that will then be plotted in bar graphs or line graphs. Which type of diagram you choose depends on the type of information you would like to visualize.

Bear in mind that depending on which version of MySQL you use, the exact set of variables available may differ.

## Tools

---

The examples in this recipe provide a quick overview of what the server is currently doing based on the command counters for the individual data manipulation statements. More sophisticated stats can be built using some of the lower-level status variables like those starting with `Handler_`. They provide information that may allow you to identify performance problems or bad indexing.

MySQL Administrator stores your custom diagrams in an XML file. The name and location of the file are operating system dependent. In Windows and Linux, it is called `mysqladmin_custom_health.xml`. In Windows, it is located in the same folder as `mysqlx_user_connections.xml`. In Linux and Mac OS X, there is a subfolder named `administrator`. Moreover, the MySQL Administrator GUI uses a different filename: `mysqladmin_custom_health.xml`.

Fortunately, the contents are identical, so copying a file from one system to the other requires you to adapt its name and place appropriately.

In certain cases you might want to edit the graph definitions manually. For example, there is no way to change the graph type from **Line** to **Bar** or vice versa once you have created a diagram in the MySQL Administrator GUI. By directly editing the XML definition you gain a lot more flexibility.



### Caution:

Make sure to keep a backup copy of your edits before you start MySQL Administrator and try your changes. If you make a mistake that prevents MySQL Administrator from reading and applying the file, it will overwrite the original with defaults on exit!

## There's more...

While generally most recipes are agnostic to the exact version of MySQL you are running, this one has a caveat: MySQL has a known issue that leads to different behavior in different versions.

From version 5.0 on, the effect is that you cannot query the counters, for example, the number of temporary tables, without modifying it as you go. In versions prior to 5.0, the STATUS commands (the foundation for many of the values MySQL Administrator displays) could be executed without modifying them. Their results were sent to the client immediately. Starting with 5.0, a temporary table with the results is created automatically and its rows are sent to the client. Unfortunately, this temporary table itself is counted in the statistics.

Even though this new technique allows accessing this statistical data in stored procedures, it increases noise in measuring. As a simple workaround, some formulas can be modified to compensate, for example by subtracting the value that an idle server would display. Of course, this is not viable for all types of calculations. Also, consider that there might be multiple simultaneous connections that execute the same queries. You cannot reliably compare the results of these queries.

---

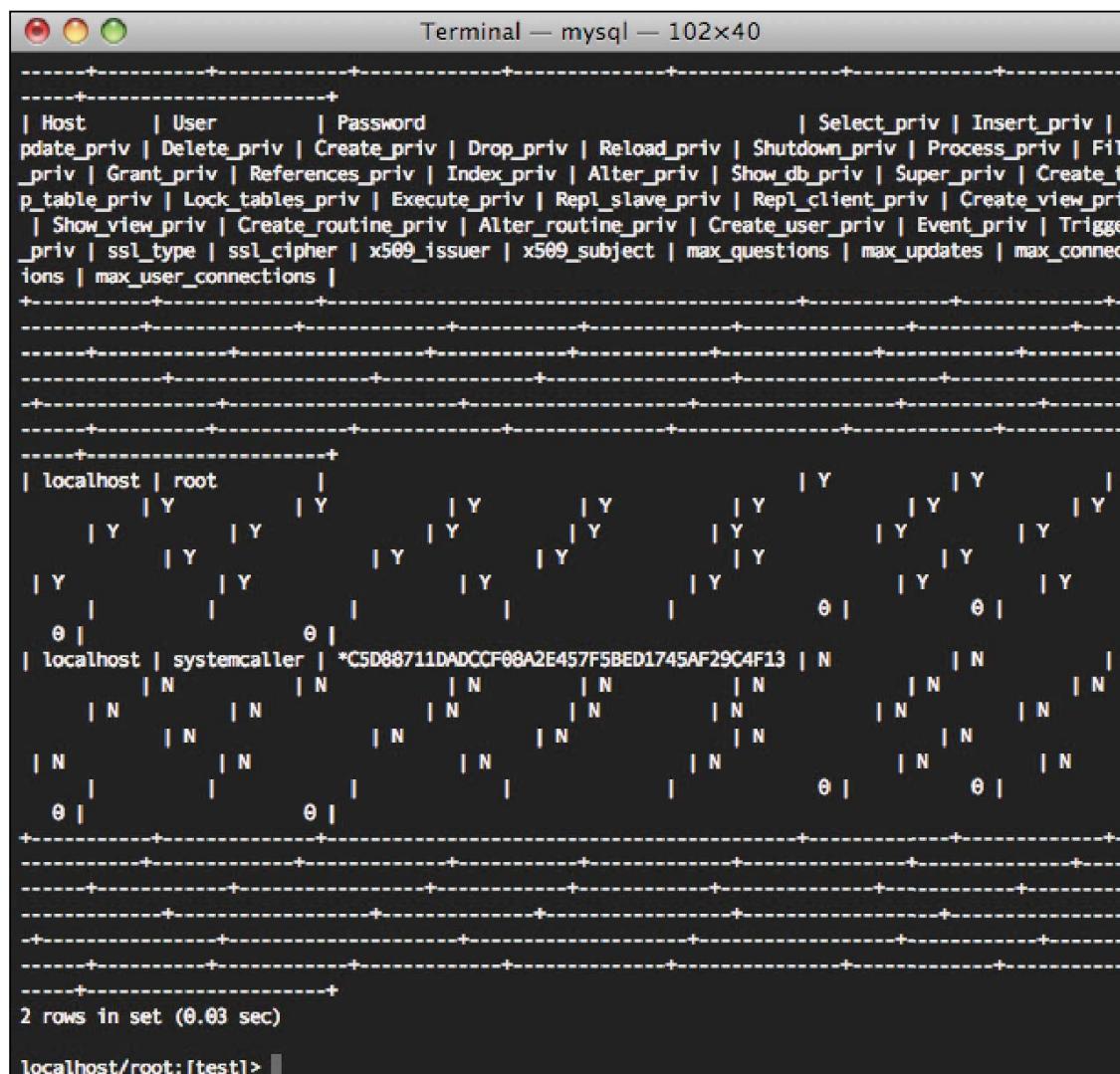
In the end you will have to keep this in mind when interpreting the stats you see.

For more details, we suggest you read the discussion that is part of the bug report <http://bugs.mysql.com/bug.php?id=10210>.

## Displaying query results page by page and with scrolling using the MySQL command-line client

The `mysql` program is part of every MySQL installation. It is a powerful tool, even though some of its functionality is only available on non-Windows platforms. It may not be the most pleasing, but can be used as a versatile client to the database server both interactively and in scripted scenarios.

When using the MySQL command-line client `mysql`, you are certainly familiar with something like the following screenshot:



A terminal window titled "Terminal — mysql — 102x40" displaying MySQL query results. The output shows the privileges for the "root" user on "localhost" and the "systemcaller" user. The results are displayed in a grid format with column headers: Host, User, Password, Select\_priv, Insert\_priv, etc. The "root" user has all privileges granted (Y), while the "systemcaller" user has many privileges denied (N). The command "SELECT \* FROM mysql.user;" was run to retrieve this information.

Host	User	Password	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv	Drop_priv	Reload_priv	Shutdown_priv	Process_priv	File_priv	Grant_priv	References_priv	Index_priv	Alter_priv	Show_db_priv	Super_priv	Create_table_priv	Lock_tables_priv	Execute_priv	Repl_slave_priv	Repl_client_priv	Create_view_priv	Show_view_priv	Create_routine_priv	Alter_routine_priv	Create_user_priv	Event_priv	Trigger_priv	ssl_type	ssl_cipher	x509_issuer	x509_subject	max_questions	max_updates	max_connections	max_user_connections											
localhost	root		Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y						
	systemcaller	*C5D88711DADCCF08A2E457F5BED1745AF29C4F13	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

2 rows in set (0.03 sec)

localhost/root:[test]>

## Tools

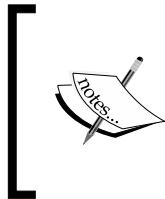
It only shows two rows of data, but you have almost no chance of getting the information wanted. When you have larger result sets, it gets even worse because then vertical scroll bars might even prevent you from reading the column titles (depending on how large your browser's buffer is). Fortunately, there is a way to process the output of any MySQL command without sending it to the screen.

This recipe will show you how to view a result that is too large to fit on one screen in a more readable way. It allows both vertical and horizontal navigation as well as some other operations.

The next example assumes that you have an instance of the MySQL server running on your local machine. Of course, you can connect to any other host as well.

### Getting ready

To try this you will need the `mysql` command-line client on Mac OS X, Linux, or any other Unix-like platform.



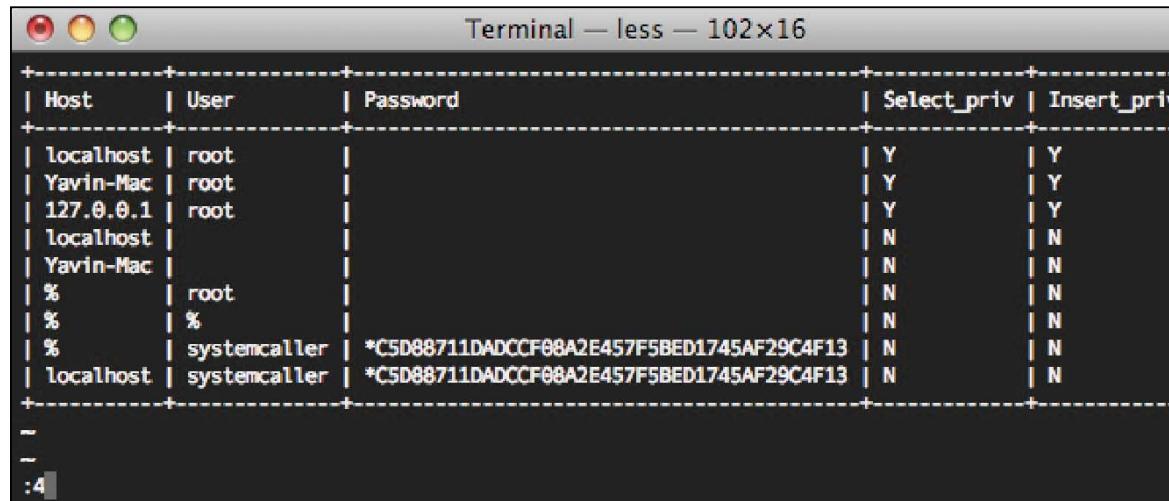
Unfortunately, for Windows users there is no way to apply this recipe because the client cannot provide the necessary functionality in Windows as it is based on some underlying functions that are not available in Microsoft's operating system!

You will also need the `less` pager utility. This is available on Mac OS X and virtually every Linux distribution, so you should not have any problems here. For simplicity, the example assumes that you have privileges to `SELECT` from the `mysql` system database. This is not required, however, as any `SELECT` will do.

### How to do it...

- Run the `mysql` command-line client:  
`$ mysql`
- Enter the following commands at the `mysql` prompt:  
`mysql> pager less -SFX`  
`mysql> SELECT * FROM mysql.user;`

- 
3. You will get an output similar to this:



```
Terminal — less — 102x16
+-----+-----+
| Host | User | Password | Select_priv | Insert_priv |
+-----+-----+
| localhost | root |          | Y          | Y
| Yavin-Mac | root |          | Y          | Y
| 127.0.0.1 | root |          | Y          | Y
| localhost |      |          | N          | N
| Yavin-Mac |      |          | N          | N
| %       | root |          | N          | N
| %       | %    |          | N          | N
| %       | systemcaller | *C5D88711DADCCF68A2E457F5BED1745AF29C4F13 | N          | N
| localhost | systemcaller | *C5D88711DADCCF68A2E457F5BED1745AF29C4F13 | N          | N
+-----+-----+
-
-
:4
```

In this view you can scroll horizontally and vertically using your cursor keys. The `q` key will bring you back to the command prompt.

4. Optionally, to unset the pager type this:

```
mysql> nopager
```

## How it works...

The `pager` command allows you to specify an arbitrary program, which is executed after each query. This is useful if you would normally see the result of a subsequent command. This includes results from `SELECT` statements as well as any other output, for example, from `SHOW ENGINE INNODB STATUS`.

The program receives exactly the same output that would otherwise be printed to the screen.

In this example, the query result is handed over to the `less` utility, a common tool for viewing large files. It is useful for displaying information that would usually be too large to fit on a single screen. `less` allows for navigating through the output using the cursor keys, which is what you just tried. It also knows a command `q` that we used earlier and that exits the program, effectively taking you back to the command prompt.

## Tools

The `-SFX` parameters passed to `less` in this example specify some options that are recommended for use with `mysql`.

Parameter	Description
<code>-S</code>	Causes lines longer than the screen width to be chopped rather than wrapped around. This keeps the table format of the <code>SELECT</code> output intact and allows to navigate horizontally using the cursor keys.
<code>-F</code>	In case the result of a command fits on screen without the need for any scrolling, <code>less</code> automatically terminates, effectively saving you from typing "q" to get ready for the next command.
<code>-X</code>	Instructs <code>less</code> to skip some initialization that would otherwise potentially lead to undesirable screen layout problems.

`less` offers a large number of other parameters you may pass to further customize its behavior. For a complete list, refer to the manual page available from a command-line terminal:

```
$ man less
```

## Extracting information from verbose output using the MySQL command-line client

Output from MySQL commands can be rather verbose. For example, the `SHOW ENGINE INNODB STATUS` command usually produces enough text to cause your typical terminal window to scroll.

Often you are not interested in everything the output contains, but only want to know a particular detail that easily gets lost in the overall amount of text.

This recipe will show you how to easily extract information from such output using the `grep` command. In particular, we will extract the **BUFFER POOL AND MEMORY** section from the `INNODB STATUS` output.

### Getting ready

To try this, you will need the `mysql` command-line client on Mac OS X, Linux, or any Unix-like platform.



Unfortunately, for Windows users there is no way to apply this recipe because the client cannot provide the necessary functionality in Windows as it is based on some underlying functions that are not available in Microsoft's operating system!

Furthermore, you will need the `grep` text search utility. This is available on Mac OS X and virtually every Linux distribution by default, so you should not have any problems here.

## How to do it...

1. Open a connection to the MySQL server using the `mysql` command-line client:
2. Set the pager to the following command:  

```
mysql> pager grep -A 12 -e "BUFFER POOL AND MEMORY"
```
3. Request the InnoDB status using the following command:  

```
mysql> SHOW ENGINE INNODB STATUS\G
```
4. Instead of the complete InnoDB status output, you will only see the 12 lines under the **BUFFER POOL AND MEMORY** heading:

```
Terminal — mysql — 70x20
localhost/root:[test]> pager grep -A 12 -e "BUFFER POOL AND MEMORY"
PAGER set to 'grep -A 12 -e "BUFFER POOL AND MEMORY"'
localhost/root:[test]> SHOW ENGINE INNODB STATUS\G
BUFFER POOL AND MEMORY
-----
Total memory allocated 17334612; in additional pool allocated 871296
Dictionary memory allocated 26616
Buffer pool size      512
Free buffers          476
Database pages        36
Modified db pages     0
Pending reads          0
Pending writes: LRU 0, flush list 0, single page 0
Pages read 36, created 0, written 0
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
1 row in set (0.00 sec)

localhost/root:[test]>
```

---

Tools

## How it works...

The `pager` command allows you to specify an arbitrary program, which is executed when you would normally see the result of a subsequent command. This includes results from `SELECT` statements as well as any other output, for example from the `SHOW ENGINE STATUS` command.

The program is handed the exact same output that would otherwise be printed to standard output. In this example, the query result is handed to the `grep` utility—a common tool to search for specific keywords or phrases in text content. It offers a wide range of parameters and configuration options. In this example, the `-A` (number of lines to be displayed after the one that contains the search term) and `-e` (specifies the search term) options are used to first look for the line that contains the text **BUFFER POOL AND MEMORY** and outputs this and the 12 lines following it.

The exact number of lines to be used depends on the output you are filtering. In this example, the length of the relevant section is 12 (you will have to adapt this to a value suitable for your particular use case). Keeping a note at hand with the correct number of lines for your commonly used commands is recommended.

## There's more...

`grep` is not limited to searching for fixed text fragments. It supports case-sensitive and case-insensitive modes, matching lines that do or specifically *don't* contain the search term, and much more. Apart from simple search terms, you can also specify to search using regular expressions, which makes it extremely flexible and powerful. You should have a look at the `grep` manual pages for more detailed information.

Over time you will find yourself using the same filters repeatedly. Consider putting small individual shell scripts that can also be used as a `pager`, saving you the trouble of typing potentially long and complex commands over and over again.

## Specifying a default pager

In the *Displaying query results page by page and with scrolling using the mysql command-line client* and *Extracting information from verbose output using the mysql command-line client* recipes, we presented a way to have the `mysql` command-line client send its output to an external program to have it formatted before displaying.

Depending on what kind of processing you need, the client's `pager` command provides a useful way of specifying the external command to use. However, this choice is lost when you exit the client. Next time you want to use it, you are back with the standard output (which is to simply write the data to your console).

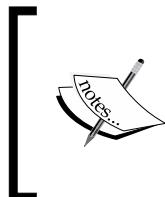
---

This recipe will show you how to configure `mysql` to use an external command of your choice as the default on startup and whenever you issue the `pager` command without parameters, and revert to the default setting.

The next example assumes you have an instance of the MySQL server running on your local machine. You can also connect to any other host.

## Getting ready

To try this you will need the `mysql` command-line client on Mac OS X, Linux, or any other Unix-like platform.



Unfortunately, for Windows users there is no way to apply this recipe because the client cannot provide the necessary functionality in Windows as it is based on some underlying functions that are not available in Microsoft's operating system!

In this example, we are going to use the well-known `less` pager utility that comes pre-installed in Mac OS X and virtually every Linux distribution, and set it up as the default pager, effectively making all `mysql` output navigable if it is larger than your terminal window.

## How to do it...

1. From your home directory, open the `.my.cnf` file in a text editor. Note that this is a hidden file (because of the leading dot (.) in the filename). If it does not yet exist, create an empty file with that name. Make sure you include the leading dot in the filename.
2. Look for a section called `[mysql]`. If it is not there, add it by inserting the following line at the end of the file:  
`[mysql]`
3. Below that line (and before any other section) insert this:  
`pager=less -SFX`
4. Save the file.

Each new instance of the command-line client you start from now on will default to the `less -SFX` pager. You can still use the `pager` command to specify a different command to process output inside `mysql`.

**Tools**

## How it works...

When the mysql client starts, it looks for options in a file called `.my.cnf` in your home directory. Inside this file it reads the `[client]` and `[mysql]` sections. By placing a setting in there we configured a pager different from the default.



Make sure you put the pager setting in the `[mysql]` section, and not in the `[client]` section!



While the `mysql` command-line client would work with the new setting, other client tools (such as `mysqladmin`) would also try to read it and abort because they do not know what to do with it. `[mysql]`, however, is solely read by the command-line client.

## There's more...

The example uses the `.my.cnf` option file in your home directory to specify the default pager. In a multi-user environment this makes perfect sense, since everyone can configure their own client with individual settings.

But there are several locations `mysql` looks in for its configuration that might come in handy if you want to share parts of your configuration among several user accounts. These are explained in detail in section 4.2.3.3. *Using Option Files* in the MySQL online manual at <http://dev.mysql.com/doc/refman/5.1/en/option-files.html>.

## Using a custom prompt to distinguish connections

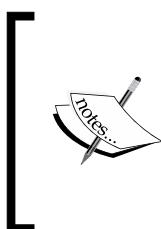
In many situations, the `mysql` command-line client program is the first choice when connecting to MySQL servers. With its default `mysql>` prompt, it sets itself apart from the local command shell's prompt so you know where you are at a glance.

But when you need to connect to more than one server at a time (or maybe just keep multiple connections to the same one open), this simple prompt is not enough to quickly tell the sessions apart.

In this recipe, we will change the default prompt to include useful session-related information. It will be configured to show the host you are connected to, the username used to connect, and the current default database.

## Getting ready

To try this you will need the `mysql` command-line client and login credentials to at least one MySQL server. Ideally, you can access more than one machine to see the effect of the settings presented.



In Windows, there is no `.my.cnf` but a `my.ini` file. Moreover, it is not usually located in your user profile—or home—directory, but in the Windows folder or the folder your MySQL installation resides in. If you are unsure, use the `my.ini` in the Windows directory or create one there if it does not exist yet!

## How to do it...

1. Open the `.my.cnf` file in a text editor. It is located in your home directory. If it does not yet exist, create an empty file with that name. Make sure you include the dot in the filename.
2. Look for a section called `[mysql]`. If it is not yet there, add it by inserting the following line:  
`[mysql]`
3. Below that line (and before any other section) insert this:  
`prompt=\h/\u: [\d]>\_`
4. Save the file.
5. Open at least two connections with different credentials and/or to different servers. In each connection you can now see the host you are connected to, followed by the current username and the name of the current default database.

```

Terminal — mysql
argument.                                         Pending writes: LRU 0, flush list 0, single page 0
status   (\s) Get status information fr Pages read 36, created 0, written 0
system   (\!) Execute a system shell cd 0.00 reads/s, 0.00 creates/s, 0.00 writes/s
tee      (\T) Set outfile [to_outfile]. No buffer pool page gets since the last printout
en outfile.
use      (\u) Use another database. Take ROW OPERATIONS
t.
charset  (\C) Switch to another charset 0 queries inside InnoDB, 0 queries in queue
ssing binlog with multi-byte charsets. 1 read views open inside InnoDB
warnings (\W) Show warnings after every Main thread id 2957045760, state: waiting for server activity
nowarning (\w) Don't show warnings after Number of rows inserted 0, updated 0, deleted 0, read 0
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
For server side help, type 'help content 1 row in set, 1 warning (0.00 sec)

slave_A1/replamin:[accounting]> mysql-51-test/root:[productdb]>

Terminal — mysql — 102x12
+-----+
| CATALOG_NAME | SCHEMA_NAME          | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME | SQL_PATH |
+-----+
| NULL          | information_schema          | utf8                      | utf8_general_ci        | NULL      |
| NULL          | accounting                  | latin1                    | latin1_swedish_ci     | NULL      |
| NULL          | mysql                       | latin1                    | latin1_swedish_ci     | NULL      |
| NULL          | productdb                  | latin1                    | latin1_swedish_ci     | NULL      |
+-----+

```

Tools

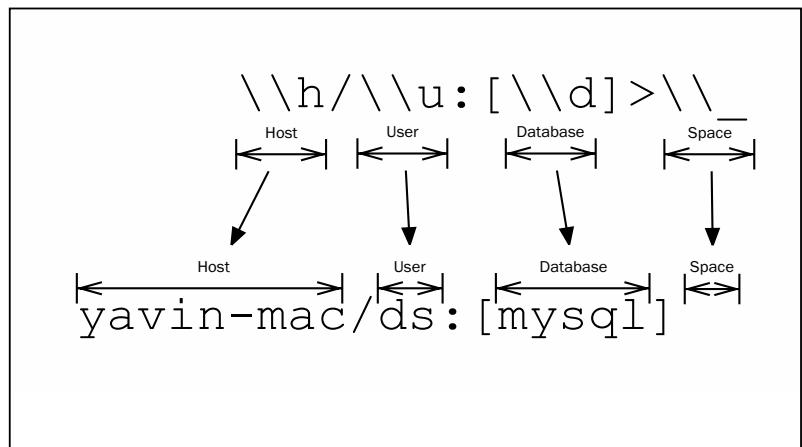
## How it works...

Basically the way this recipe works is the same as described in the *Specifying a default pager* recipe. The difference is just the parameter you set (`prompt`). Please read the description there to get a deeper insight into the preference file mechanism.

The value we set for `prompt` consists of several tokens, each of which gets replaced by a value specific to the current connection:

Token	Replacement
<code>\h</code>	The name of the host you are connected to
<code>\u</code>	Your username
<code>\d</code>	The name of the current default database
<code>\_</code>	A space character

For the machine used to write this book, the prompt in this example is expanded as follows:



## There's more...

See the *There's more...* section of the *Specifying a default pager* recipe for more information on the parameters file.

Apart from the tokens used in this example, there are several more that allow you to tailor the `prompt` to your needs. You can find a comprehensive list for your MySQL server in the server's online manual section 4.5.1.2. *mysql Commands* at <http://dev.mysql.com/doc/refman/5.1/en/mysql-commands.html>.

Configuring a prompt that really suits all your needs is often a matter of trial and error. It can take some time before you are really satisfied. To speed up this process use the `prompt` command inside `mysql` to change the prompt for only the current session interactively.

---

## See also

- ▶ *Specifying a default pager*

# Encrypting a MySQL server connection with SSH

When connecting to MySQL server over the Internet, you should be aware that any communications between your client and the host machine are transferred without encryption. This applies to both the login credentials and the actual database contents you send to or receive from the server. While the MySQL authentication schema provides some measure of protection for your password (it is not sent in the clear, but using a challenge/response type mechanism), it is not as secure as if you were using a real cryptographic encryption.

Though in a controlled environment like a corporate local area network this may not be an issue, sending database contents through a public network is a different thing.

Theoretically, MySQL allows SSL secured connections between the server and its clients, both are built to support it. Unfortunately, the default MySQL packages available for download from the `mysql.com` website are not encryption enabled. To get that, you would have to compile the server from the source code and include the necessary encryption options. This is tedious and requires a rather substantial amount of knowledge and the appropriate tool chain.

This recipe will show you how to establish an encrypted connection to a standard MySQL server with any client program such as the command-line client or MySQL GUI Tool. We will start on. In the next example, we will encrypt a connection from MySQL Administrator to a standard MySQL server.

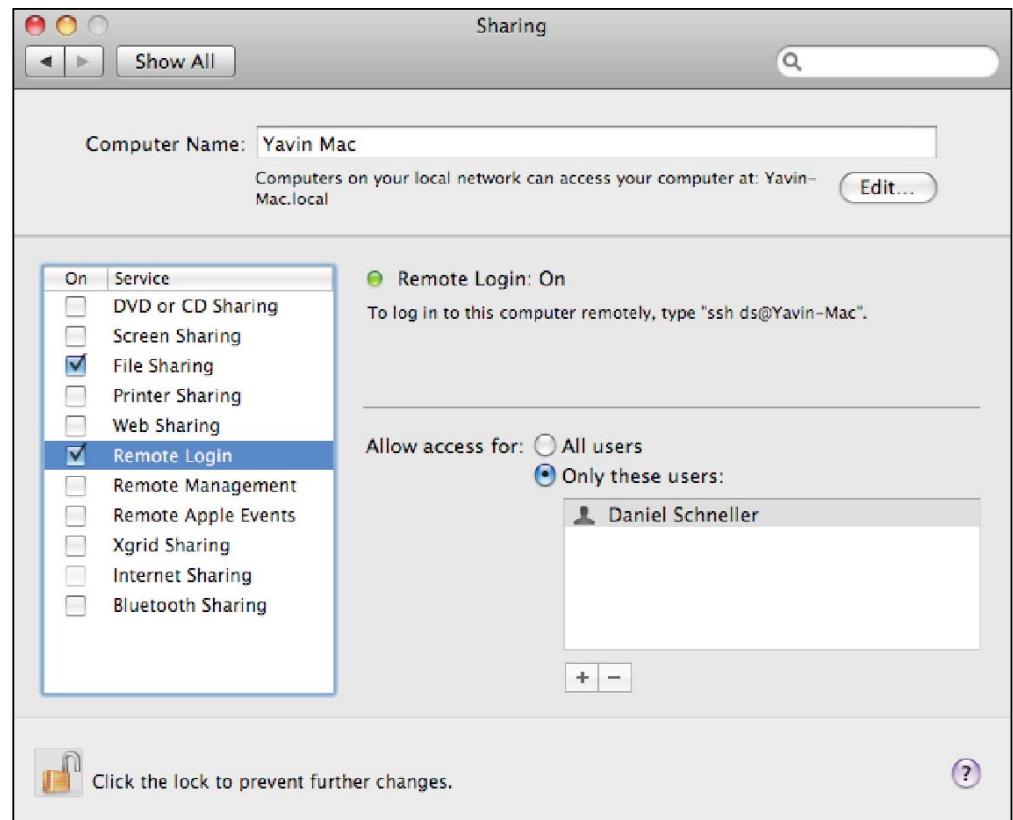
## Getting ready

For this recipe to work, the MySQL server's operating system has to support a secure (SSH) connection. Most Linux distributions include out-of-the-box support for this. Mac OS X 10.5 "Leopard" also supports encrypted SSH connections without third-party products. If you are using Windows, MySQL server configurations for these versions must be amended with appropriate software packages.

Windows users also need to install the SSH server themselves. There are several open source offerings, but for the purpose of this recipe, the SSH server available with the Cygwin package is completely sufficient.

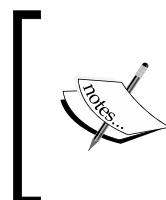
## Tools

Make sure you have set up SSH correctly before you proceed. In Mac OS X, go to the **Preferences** and open the **Sharing** pane (as seen in the following screenshot). Enable **Remote Login** and make sure you select the appropriate user accounts:



In Linux, you usually do not have to do anything.

In Windows, follow the setup instructions for the SSH server of your choice (for example <http://www.cygwin.com> or <http://www.freesshd.com/>).



Please note that you will need a set of username and password for the MySQL server as well as for the operating system it runs on. This is because the SSH will check before allowing you to try to connect to the database server!

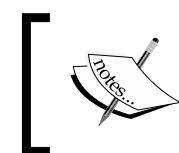
While in Linux and Mac OS X all necessary client tools are again bundled with the system, Windows users need to acquire an SSH client program and install it on the where they want to run their MySQL client. The Cygwin package includes an SSH client, but there is a better alternative: The excellent free *PutTY* SSH client. It is available on <http://www.chiark.greenend.org.uk/~sgtatham/putty> (alternatively search for 'putty' into a search engine of your choice). We will use it in this and the following examples.

In the next example, we are going to log in to a MySQL server on Mac OS X through

## How to do it...

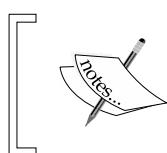
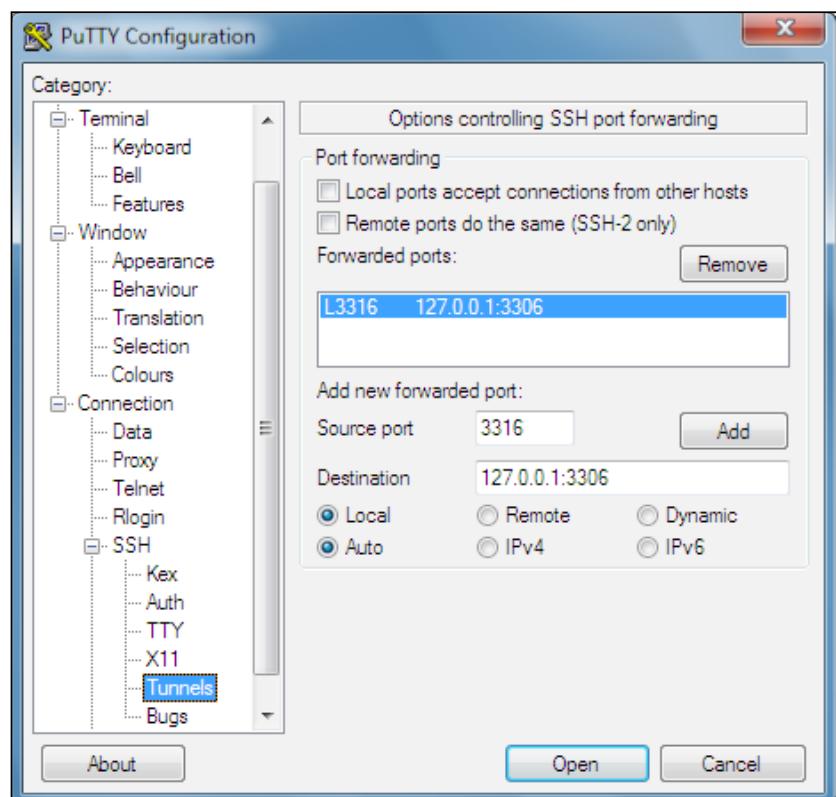
1. In Linux and Mac OS X, open a command-line shell. In Windows, launch Putty.
2. In Linux and Mac OS X, enter the following command. Replace *OSUSER* with your operating system account name. Replace *HOST* with the SSH server host name.

```
$ ssh -L3316:127.0.0.1:3306 OSUSER@HOST
```



Notice that there is no mention of a MySQL username or password yet! This is purely to log on to the SSH server.

In Windows, set up a connection with PuTTY making sure you have the following settings on the **Connection/SSH/Tunnels** page:

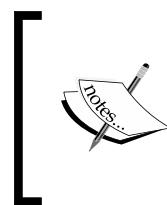
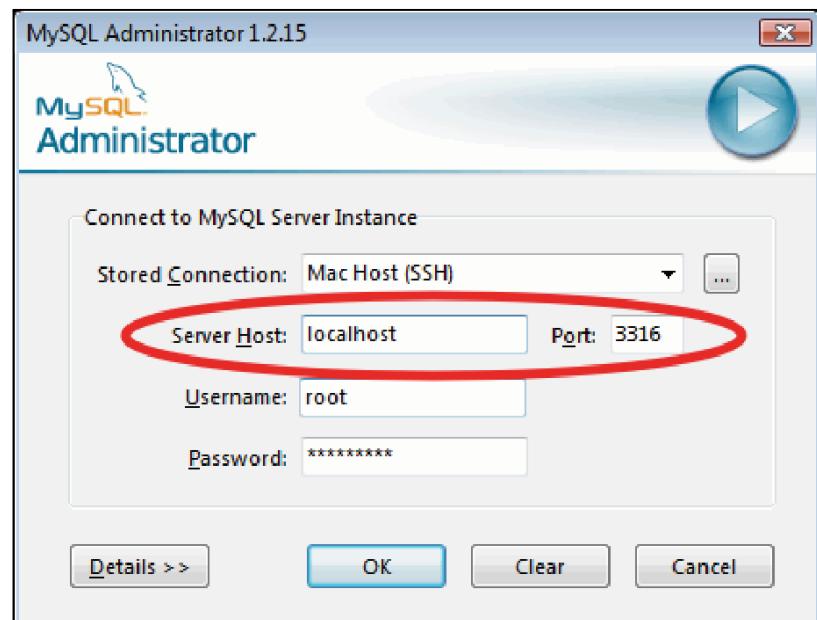


Make sure you hit the **Add** button once you have entered the **Source port** and **Destination**; otherwise the tunnel settings will not be activated!

3. For Linux and Mac OS X, hit the *Enter* key; for Windows click the **Open** button.
4. Log in to the server using your operating system password.

**Tools**

6. Once the connection has been established, launch MySQL Administrator.
7. Set up a connection like this:



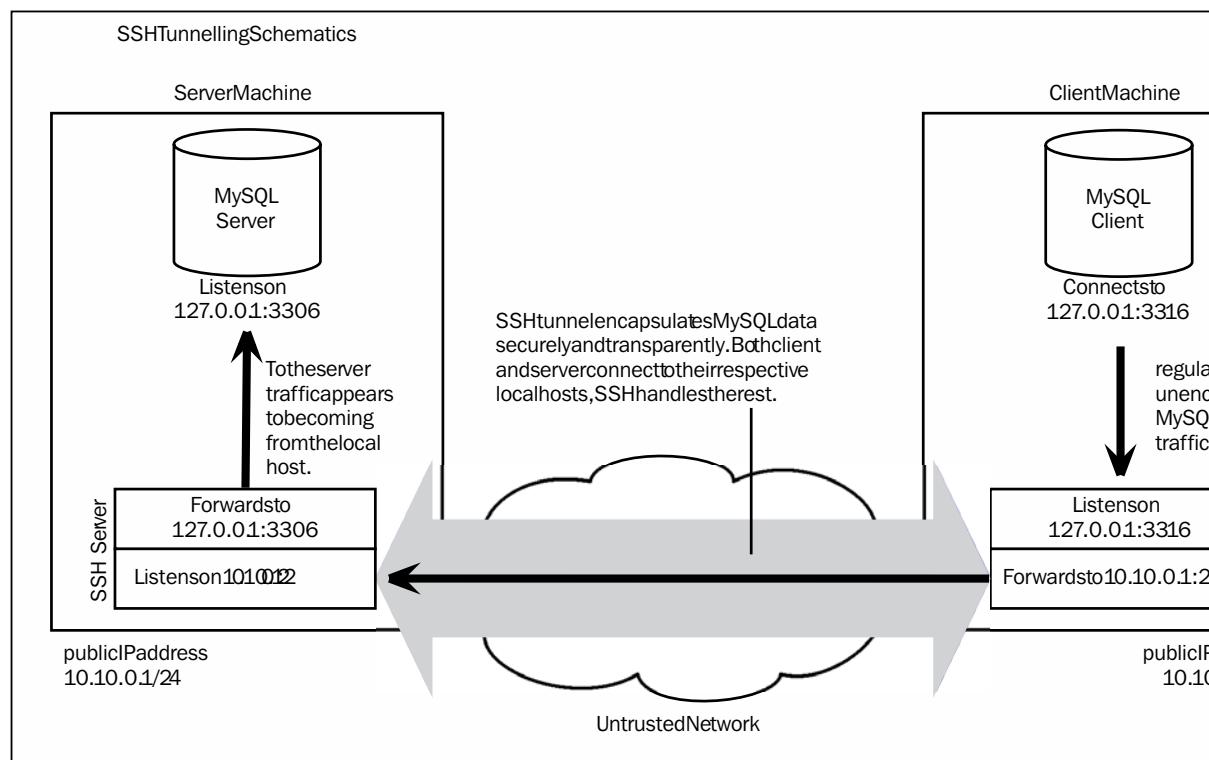
Please note that the server host is `localhost`, even though you are about to connect remotely. Also, make sure that the port number is set to 3316. The username and password are those for the MySQL server this time.

8. Click on **OK** to connect.
9. To disconnect later, first close MySQL Administrator then the SSH client p

## How it works...

The SSH server (or daemon as it is often called) and client toolset provide a versatile and powerful way of *tunneling* network connections through the secure link they establish one another. This feature enables encrypted data transfers for applications that are not able to do so themselves, like in our case the default MySQL server and client built into Mac OS X.

To make the encryption process transparent to the tunneled application (MySQL), the client accepts incoming connections on a configurable TCP port on its behalf. Any connection that is established with this port gets its data encrypted and sent to the SSH daemon, which then decrypts the data and relays the original information to the original MySQL server).



Let's have a look at the command line we used on the client machine (this is on the preceding image). The parameters in PuTTY are identical:

```
$ ssh -L3316:127.0.0.1:3306 OSUSER@HOST
```

ssh is the name of the client program. The `-L3316` parameter tells it to open port 3316 on the local machine (hence the `-L`).

Following that, the address and port of the target MySQL server are set. This is a little bit counterintuitive at first because 127.0.0.1 is the standard address of the localhost. The key to understand this is that the part after the colon is an address from the perspective of the SSH remote server machine. In the example, the MySQL server is running on the local host 127.0.0.1 on port 3306 (on the left side in the earlier image).

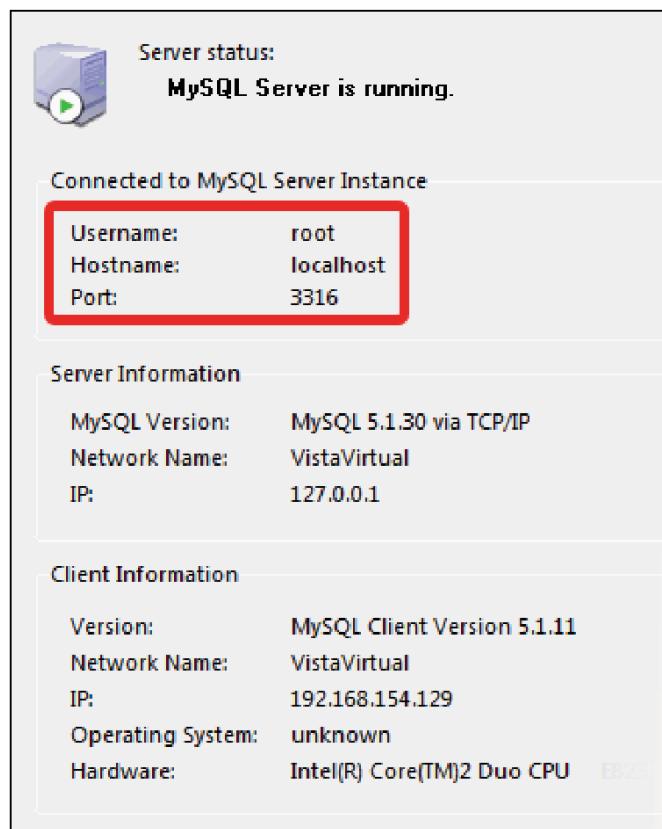
Finally, `OSUSER@HOST` tells the client which host to connect to and which user account to log in with.

Once the secure channel is established MySQL Administrator can connect to port 3316 on its local (Windows) machine. ssh transparently forwards the connection to the MySQL server, which in turn connects to the database on port 3306. In the illustration (refer to the figure), this is the dashed line inside the tunnel depicted in gray.

You can also tell ssh to listen to connections on the regular port 3306, as long as there is no other process, such as a local MySQL server, using it already.

## Tools

In the next screenshot, notice the information in the uppermost box **Connected to MySQL Server Instance**. As far as MySQL Administrator is concerned, it does not know about the HOST machine; from its point of view it is connected to localhost, port 3316. This behavior in this scenario.



## There's more...

By now you must have already understood from the description earlier that you need not necessarily have the SSH daemon installed on the same machine as the target MySQL server. In fact in many production scenarios, you will have a single SSH gateway server from which you can reach your MySQL servers. You can then connect to this gateway server with one or more SSH clients and have it relay MySQL communications to the respective database servers.

Often hosting providers will not allow you to log in to your MySQL server directly for security reasons. They do, however, usually equip you with a set of SSH user credentials for the server. Usually, you get to a command shell when you log in using PuTTY or a similar program. With this recipe you will no longer be confined to using the `mysql` command-line client for your database management needs, but simply run your tool of choice (for example MySQL Query Browser) locally and just tunnel it through SSH.

If you want to attach to different MySQL hosts from the same client, be sure to specify an individual local port number (for example: 3316, 3317, 3318) for each connection.

# Creating an encrypted MySQL console via SSH

Often you connect to a remote machine using SSH just to launch the `mysql` command-line client as soon as you are logged in. Especially, on dedicated database machines it is easier to get the `mysql` prompt right after you established the SSH connection.

In this recipe, we will set up a dedicated user account on the server machine that automatically launches the `mysql` command-line client and connects you to the MySQL server once you log in using SSH. When you leave the `mysql` client, you will automatically be disconnected from SSH as well.



**Important:** The procedure presented in this recipe may pose a security risk. *Effectively, users have got shell access with this!*

*Therefore, apply this recipe only in tightly controlled environments where you trust the users!*

**Never use it to allow arbitrary access to your servers!**

`mysql` provides the `system` command on Unix-like operating systems. This command allows you to run operating system level commands in the context of the currently logged-in user. Even though in the example we utilized a restricted user account that does not have any special rights, it can still access world-readable files (like `/etc/passwd`) and run programs as if it were logged onto the server machine locally!

## Getting ready

For this recipe to work, the MySQL server's operating system has to support an SSH connection. Most Linux distributions include out-of-the-box support for this. Mac OS X and Leopard also supports encrypted SSH connections without third-party products.

While in Mac OS X and Linux, the `ssh` client is provided out of the box, Windows users need the free PuTTY SSH client from the Web or the Cygwin bundled `ssh` client.

You will need the necessary privileges to create a new operating system user on the server machine. This need not necessarily be the same machine as the database server. Moreover, you will need a valid user account on the MySQL server.

In the next example, we will use an Ubuntu Linux machine, which is both the SSH client and the database server. We assume root access on that machine. To connect to the server from Windows, use PuTTY on Windows.

**Tools**

Setting up this scenario using a Windows SSH server depends on the SSH server you want to install. While it would be technically possible to achieve this with the Cygwin SSH server, it would be rather cumbersome to do so, due to the way Cygwin integrates with the Windows user account system. Because of this, we will concentrate on Unix-like servers in this chapter and only use Windows as a client.

## How to do it...

1. Open a command shell. Create a new user group on the Linux machine using the following command:

```
sudo addgroup mysqlshellusers
```

2. Create a new directory and set the permissions using these commands:

```
sudo mkdir /usr/local/bin/mysqlshells
```

```
sudo chmod u=rwx,g=rx,o= /usr/local/bin/mysqlshells
```

```
chgrp mysqlshellusers /usr/local/bin/mysqlshells
```

3. Create a new file on the SSH server with the following content. Save it as /usr/local/bin/mysqlshells/mysqladmin.sh:

```
#!/bin/bash
```

```
echo -----
```

```
echo Connecting to MySQL...
```

```
echo -----
```

```
/usr/bin/mysql -uroot -p
```

4. Set the permissions on the new file as follows:

```
sudo chmod u=rx,g=rx,o= /usr/local/bin/mysqlshells/mysqladmin.sh
```

5. Create a new user account using this command:

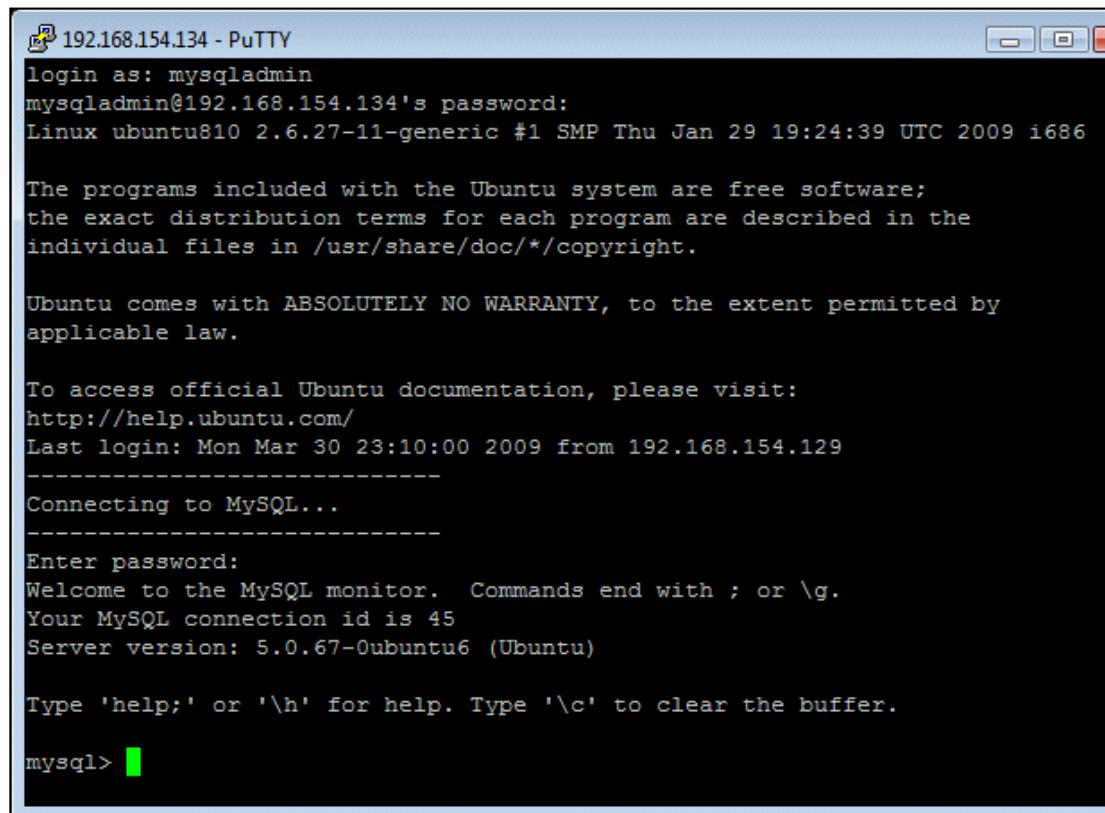
```
sudo useradd -d /tmp -g mysqlshellusers -s /usr/local/bin/mysqlshells/mysqladmin.sh mysqladmin
```

6. Set a password for the new user using this command:

```
sudo passwd mysqladmin
```

7. On the client machine launch PuTTY. Connect to the SSH server. Use mysqladmin as the username and supply the password that you set in step 5.

8. Once you have logged into the host machine, enter the MySQL password for the database root account:



The screenshot shows a PuTTY terminal window titled "192.168.154.134 - PuTTY". The session is a root login for the user "mysqladmin" on a Linux Ubuntu 8.10 system. The terminal displays the standard Ubuntu welcome message, including copyright information, warranty disclaimers, and documentation links. It then proceeds to connect to the MySQL monitor, prompting for a password. The MySQL prompt "mysql>" is visible at the bottom.

```

192.168.154.134 - PuTTY
login as: mysqladmin
mysqladmin@192.168.154.134's password:
Linux ubuntu810 2.6.27-11-generic #1 SMP Thu Jan 29 19:24:39 UTC 2009 i686

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To access official Ubuntu documentation, please visit:
http://help.ubuntu.com/
Last login: Mon Mar 30 23:10:00 2009 from 192.168.154.129
-----
Connecting to MySQL...
-----
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 45
Server version: 5.0.67-0ubuntu6 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> 

```

## How it works...

When you log in to an SSH server, the program that it will look up is set as the user's *shell*. Usually, this is a command-line shell like bash or zsh that lets you interact with the server's operating system, launch programs, and so on. It need not be a general purpose command-line interpreter, however, it is perfectly fine to specify any executable command or script. On some Linux systems, for example Mandriva, you need to add the path of your shell in /etc/shells.

When we created the new `mysqladmin` user account, we used the `useradd` command's `shell` parameter to tell it that we wanted `/usr/local/bin/mysqlshells/mysqladmin` as our login shell, which we had just created and placed there.

Upon login, the SSH server opened and ran this script, effectively starting our `mysqladmin` command-line client with the `-u root` and `-p` parameters. This is why you were asked for the second password. For security reasons we did not put the password into the shell script, even though that would have been possible and only required us to enter one instead of two passwords to log in. In a tightly controlled environment this might be a viable solution, but it is definitely not recommended.

The rest of the commands discussed earlier create a new user group called `mysqlshellusers` and restrict access to the `/usr/local/bin/mysqlshells` directory to the members of this group to prevent unauthorized users from even viewing the scripts.

*Tools*

## **There's more...**

Of course, you are not limited to a single user account or just one script. You can create as many specialized scripts as you like and assign them to different user accounts. In the previous example, the script logged on to the MySQL server as root, which is necessary for some administrative tasks. You should, however, generally use an account with restricted privileges.

To prevent you from having to enter two passwords each time you log in, you might look into SSH public/private key pairs that can be used to log in to SSH instead of password authentication without compromising security.

## **Using a PuTTY template connection for secured connections**

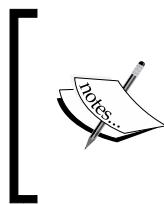
In the previous recipe, we used SSH to tunnel MySQL connections through insecure networks like the Internet and prevented the data and login information from traveling in clear text. While for a single server such connections can quite easily be set up manually, it can become tedious and inflexible once you have more than just a few servers to regularly connect to. On Mac OS X and Linux, it is very easy to create a little shell script that uses variables to store the relevant options you need to pass to the `ssh` client tool.

PuTTY users in Windows might find themselves in a situation where they would like to do the same, but unfortunately PuTTY does not provide command-line options for everything that can be configured through the GUI. For example, you might want to specify your favorite terminal font or have production systems use a different terminal background color than internal test machines.

You might be tempted to create individual connection profiles in PuTTY (if you prefer), but for every MySQL server you have to manage or multiply the corresponding registry entries where PuTTY stores its connection profiles.

Both these approaches work fine up to the point where you need to change a setting that is common to all of your profiles. In that case, you would need to load, modify, and save each and every connection the change is applied to.

This recipe will show you how to establish encrypted connections to MySQL servers using SSH tunnels that are set up using a connection template. As an example, this template will include a red signal color background for the terminal window to easily visually distinguish production systems from test machines.



**Important:** The procedure presented in this recipe may pose a security risk. Please make sure you understand the implications by reading the information box in the *Creating an encrypted MySQL console via SSH* recipe!

## Getting ready

For this recipe to work, the MySQL server's operating system has to support an SSH connection. Most Linux distributions include out-of-the-box support for this. Mac OS X also supports encrypted SSH connections without third-party products.

Windows users need to install the SSH server themselves. There are several commercial offerings, but for the purpose of this recipe the SSH server available with the Cygwin package is completely sufficient.

While in Mac OS X and Linux, the `ssh` client is provided out of the box, Windows users need the free PuTTY SSH client from the Web.

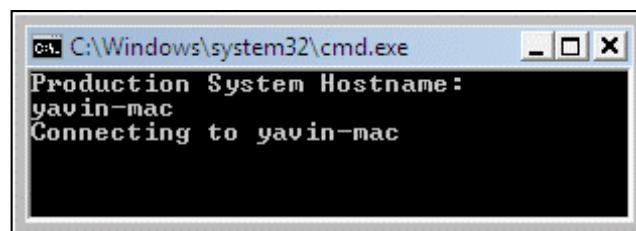
## How to do it...

1. Open Notepad and save the following code as `MySQLTunnel_PROD.cmd`

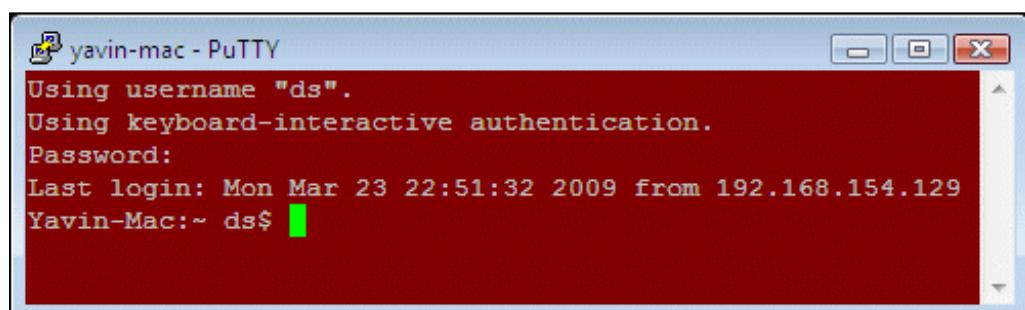
```
@echo off
echo Production System Hostname:
set /p hst=
echo Connecting to %hst%
putty.exe -L 3316:127.0.0.1:3306 -ssh -load "TMPL_PROD" %
```
2. Launch PuTTY.
3. Choose the **Window/Appearance** panel from the left-hand tree.
4. Select **Default Background** from the list and set the RGB value to 120, 0, 0.
5. Choose the **Connection/Data** panel from the left-hand tree.
6. Enter the remote operating system username in the **Auto-login username** field; for example this is `ds`.
7. Save the connection from the **Session** panel under the name `TMPL_PROD`.
8. Close PuTTY.

**Tools**

- Double-click on the MySQLTunnel1\_PROD.cmd file. You will be prompted in a window where you can enter the SSH server's hostname. In this example I entered yavin-mac:



- PuTTY will be automatically launched and will connect to the host you entered, applying both settings from the batch file (the tunnel setup) as well as from the TMPL\_PROD session (background color, username):



- Launch your MySQL client program (for example: MySQL Administrator) and connect to localhost port 3316 to connect through the tunnel.

## How it works...

Instead of creating a single connection for every server you need to connect to, we can create a template that contains all settings common to a group of connections (for example, one template for production systems and a different one for internal test machines).

Using only one set of placeholder settings allows for easy changes later on. If you had to change the initial settings for every new host, you would have to edit each copy separately.

A batch file prompts to enter the hostname to connect to, and stores it in a variable named `hst`. The batch then launches the PuTTY executable passing both the name of the template session and some additional command-line parameters (the tunnel setup, and the content of the `hst` variable), effectively merging both template and manually specified settings.

If you now need to change a configuration setting (for example, you'd like to enable logging to a text file), you just need to edit the corresponding template session in Putty and have the new settings applied to every subsequent connection that is established.

---

## There's more...

In the example, only a limited number of settings are stored in the template session. You can customize this further, for example, by using a key pair for authentication without the user having to enter a password upon each connection.

You can also introduce more variables in the batch file and hand them to one or more of PuTTY's other command-line options. Refer to the online documentation to find out what options can be set directly from the command line.

If your servers are set up using a naming scheme, you might reduce the amount of configuration even further. Just have the user enter only the variable portion of the hostname and supply the complete hostname in the batch file like so:

```
@echo off
echo Subsidiary no:
set /p subsno=
set hst=SUBS_%subsno%.example.com
echo Connecting to %hst%
putty.exe -ssh -load "TMPL_PROD" %hst%
```

Here the hostname is built from a prefix SUBS\_ which is then appended by a fictitious subsidiary number and complete with a domain suffix .example.com. Adapt this environment appropriately.

By either duplicating the batch file or adding some sort of menu to it, you can also switch between different template sessions depending on which set of preferences you would like to use for a given connection.



# Backing Up and Restoring MySQL Data

This chapter will cover the basic tasks necessary to back up your MySQL data efficiently and the steps to restore this data if necessary. We will discuss the following recipes:

- ▶ Using MySQL Administrator GUI Tool as a frontend for backups
- ▶ Copying all data files to a backup location
- ▶ Creating a SQL dump of all databases
- ▶ Creating a SQL dump of specific databases
- ▶ Creating a SQL dump of specific tables
- ▶ Compressing SQL dumps on-the-fly
- ▶ Rotating and purging binary logs
- ▶ Using replication to perform backups without hurting a production system's performance
- ▶ Restoring data from a dump to a previously backed-up state
- ▶ Performing a point-in-time recovery using the binary logs

## Introduction

Although MySQL has a reputation for robustness and data loss is a problem you will never encounter, it is best to be prepared for when your data gets corrupted or lost. Experience shows that it will eventually happen and probably when you least expect it.

The first thing you should make sure of is that you have a backup at hand. In this chapter we will show you different ways of saving your data elsewhere. But having a backup is not enough, as even the most complete backup is basically useless if you are not able to restore your data from it. This chapter also covers different ways of restoring the data in the event of losing it using an existing backup.

You should, however, be aware that a backup strategy does not only consist of the backup itself. You should also consider the details on how to back up and how to restore your data. You should also consider the aspects like backup frequency, how many generations have to be kept available, suitable backup media, and constructional conditions. Is one backup per week sufficient? Or is one backup a day a better choice? Will you need a tape drive, or will a USB hard disk do? Is it necessary to store backup media in a separate fire compartment? All these questions will be answered differently depending on your application's criticality, so there are no best practices.

The only thing that we strongly encourage you to do in all cases is to repeatedly and continuously test your restore process! A restore process that was tested some years ago might not work today for different reasons. One common problem with restore processes is that the documentation is outdated or not available to the people that are responsible for restoring the database ("not available" also includes "they do not know where to find it"). Another standard issue is that there are no precise responsibility definitions—the people that you think are responsible for restoring the database may neither know of the responsibility nor how to do it. And the fact that the restore process worked perfectly within the given parameters last year is no guarantee that this is the case today as well (data growth being the reason for such a difference). Restoring an almost empty database took only a few minutes back then, but restoring the current multi-terabyte database takes more than six hours, which is not acceptable if the backup-restore concept states a maximum restore time of six hours.

So, as with a fire alarm, you should try and test it on a regular basis to make sure that the restore process still works as expected, and hope that you will never need it.

While definition of a full backup-recovery strategy is beyond the scope of this chapter, we will provide you with the basic technical means of saving and restoring your database.

# Using MySQL Administrator GUI Tool as a frontend for backups

## Getting ready

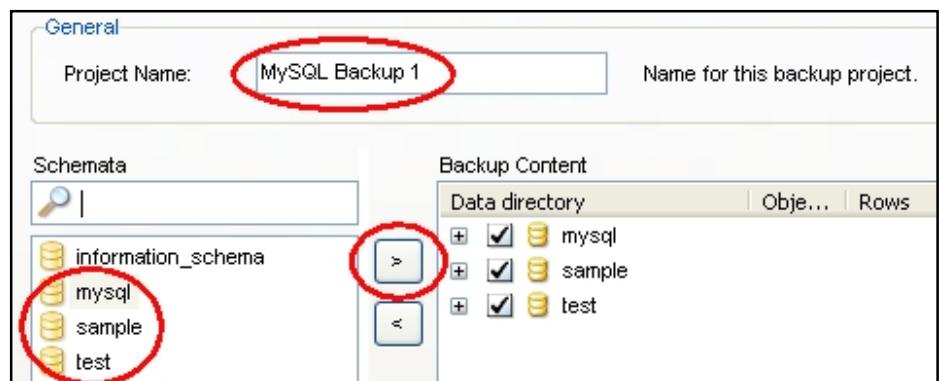
To follow the steps in this recipe, you will need an account that has sufficient permissions to perform a backup (you will need SELECT and LOCK TABLES privileges). We will assume an account named `backup_usr` (refer to Chapter 8, *Defining a specific user for backup*).

You should also make sure that there is no write access to your database. This is to prevent locking issues. For further explanation please refer to the *There's more...* section.

And finally, you will need sufficient space on one of your drives to store the backup files.

## How to do it...

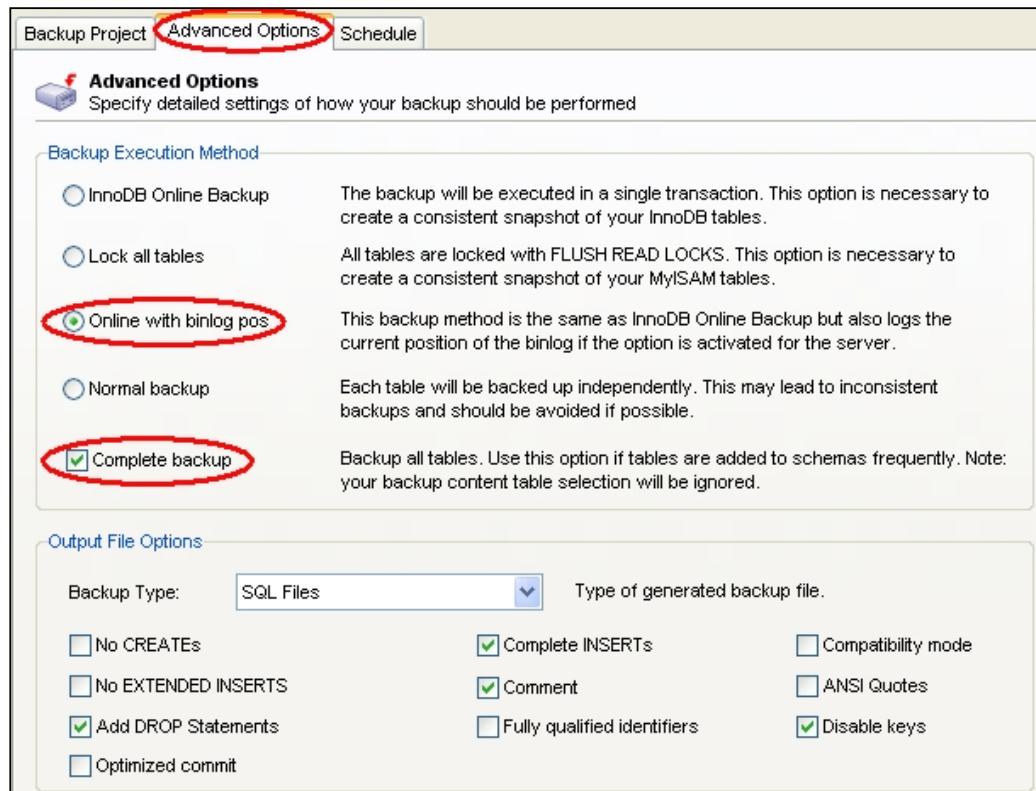
1. Start MySQL Administrator. Connect to your database server using the `backup_usr` account.
2. Select the entry **Backup** either from the list on the left or from the **View** menu.
3. Click on the **New Project** button.
4. Enter the project name **MySQL Backup 1** in the **Project Name** field.
5. Successively select each schema from the **Schemata** list and click on the **>** button to add them to the **Backup Content** list. Exclude the **information\_schema** (you will be warned if you accidentally try to add this to the list).



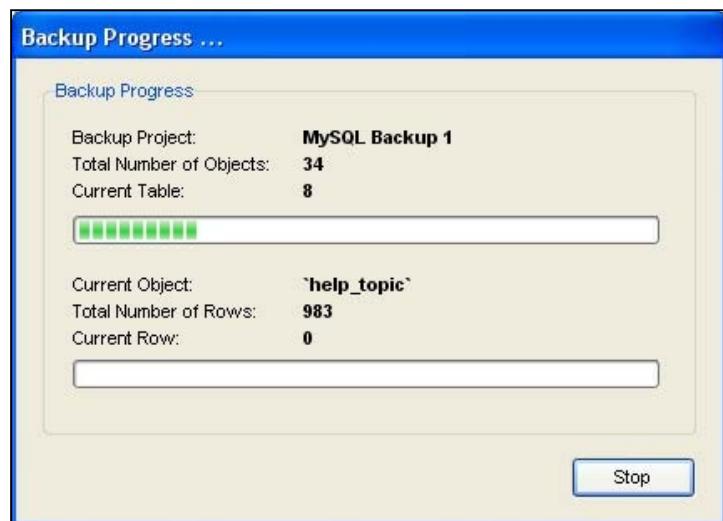
6. Select the **Advanced Options** tab.
7. Choose **Online with binlog pos** as the backup execution method.

## Backing Up and Restoring MySQL Data

8. Enable the **Complete backup** option and click on the **Save Project** button.



9. Click on the **Execute Backup Now** button, and select a location for the backup in the next dialogue.  
 10. The backup will begin and a progress indicator will be displayed, as seen in the next screenshot:



## How it works...

This recipe is a pretty straightforward way of creating a backup. With steps 1 and 2, you have created a backup of your MySQL database. Step 3 is optional and allows you to restore the backup to a different MySQL server or database.

---

In step 5, you select all databases that will be included in your backup. If you wish to back up all databases, but only a selected schema, you can also restrict your selection to specific databases. But please be aware that in case of cross-database dependencies you must select all relevant databases as well, otherwise this might lead to problems later when trying to restore the data. Also, note that all MySQL accounts and their respective privileges are stored in the **mysql** schema. If you choose not to include this in your backup, you will need to restore the accounts and privileges by other means.

Steps 6 through 8 define the exact method for backing up your data. With step 7, you can choose the default method from **InnoDB Online Backup** to **Online with binlog pos**. We advise using this method because its performance is identical to the default method, but it provides additional information about the current binlog position that is extremely useful in case of a restore. The **Complete backup** option selected in step 8 makes sure you can use the backup project later without any changes even if new tables were added since the backup's creation. If you do not plan on using the backup project again, you can simply skip steps 7 and 8.

Step 9 is again pretty straightforward: after selecting the target for the backup file, click the **Start** button to start the backup process.

## There's more...

In addition to the steps described in this recipe, MySQL Administrator features several options regarding the backup. The following sections will discuss some of these options as well as the limitations of this backup approach.

## Scheduling backups

As backups are something you typically need on a regular basis, MySQL Administrator provides you with a scheduling option. Under the **Schedule** tab, you can enable scheduled backups. After entering the target directory and the base name of the backup files, you can choose between daily, weekly, and monthly backup. Saving your project using the **Save Project** button will install a scheduling entry task in your operating system (either a cron job on Unix-based systems or a Scheduled Task on Windows—you will need to provide the respective operating system credentials for this).

## Understanding and handling limitations of using MySQL Administrator for backups

The approach of using MySQL Administrator as a graphical tool makes it very easy to create backups, either manually or scheduled. For many purposes, this is completely sufficient. However, for larger installations, you will probably need some more flexibility, for example when it comes to the scheduling options or the targeted MySQL instances. If you have multiple MySQL processes running on one server, MySQL Administrator provides no out-of-the-box functionality to create backups for all of them, as it mainly targets single instance installations.

## Backing Up and Restoring MySQL Data

---

Due to these limitations, we recommend to write and schedule your own scripts for installations that deal with mission-critical data, and to simultaneously establish a trigger or event for these scripts, so the responsible administrators are informed if the backup is not created properly.

Regardless of whether MySQL Administrator is used or not, use of a desktop computer to backup your data is generally not recommended. Desktop machines are typically not set up to run continuously, so you can't rely on uninterrupted backups. Separate machines are also more prone to security issues, as they are often easily physically accessible by people, which makes data theft easier. In many installations, desktop computers and database servers are also located on different networks. This possibly facilitates attacks at the networking layer, and might even be a legal problem when dealing with personal data.

## Exploring additional backup options

We will discuss a few additional options in the **Advanced Options** tab of MySQL Administrator that might be useful for some situations.

First of all, the **Add DROP Statements** option is enabled by default. This causes MySQL Administrator to include `DROP TABLE IF EXISTS` statements in the dump. This makes sense because it helps removing all existing data from a table before restoring it after restoring the data, the table contains exactly the same data as it did at the time the dump was created. It also makes sure that the restore process will not run into problems if the table's structure has changed between dump creation and restoration. So you will probably want to leave this option enabled. But if you plan to use the dump to import additional data into an existing database, you should disable this option.

The **No CREATEs** option prevents MySQL Administrator from producing `CREATE DATABASE` statements in the dump. According to the documentation, this is intended for situations in which you want to import a dump into a different database. Unfortunately, this will not work as the dump still contains a `USE` statement that will target the restore to the original database name and cause an error if this database does not exist. Hence, this option is basically useless and can be left disabled.

The other options provided in the **Advanced Options** tab of the **Backup** section of MySQL Administrator are not relevant for typical use. For more details, refer to the `mysqldump` manual available at: <http://dev.mysql.com/doc/refman/5.1/en/mysqldump.html>

## See also

- ▶ *Defining a specific user for backup*

# Copying all data files to a backup location

The most straightforward way to back up the data of your database is to simply copy all the data files in which the data is stored to another location. In many cases, this is one of the most reliable ways to perform a backup. This recipe will describe the steps required to successfully implement this approach.

## Getting ready

For the copy-all backup, you have to shut down your database. For this you have to make sure that all connections to your database are closed. Furthermore, you have to identify the directories in which MySQL stores its data files, the InnoDB table space, and the configuration file. In the following steps, we will assume the following directories and files: C:\Program Files\MySQL\MySQL Server 5.1\my.ini for MySQL configuration, C:\MySQL\data as the MySQL data directory (where MyISAM data and the transaction logs are stored), and C:\MySQL\InnoDB\ for the InnoDB table space.

And finally, you need sufficient space on a drive to copy the data files to. In this example, we will assume a directory D:\MySQLBackup as the target directory for the backup.

## How to do it...

1. Shut down your database instance (for example using MySQL Administrator).
2. Create the target directories.
3. Copy the full content of the data directory C:\MySQL\Data\ to the destination directory D:\MySQLBackup\Data.
4. Copy the full content of the InnoDB data directory C:\MySQL\InnoDB\ to the destination directory D:\MySQLBackup\InnoDB\.
5. Copy the MySQL configuration file C:\Program Files\MySQL\MySQL Server 5.1\my.ini to the destination directory D:\MySQLBackup\.
6. Start the MySQL Server instance again.

## How it works...

This recipe basically consists of copying the important MySQL files from one location to another. It is important to understand why step 1 (shutting down the database) is necessary. Without this step you risk an inconsistent or (even worse) unusable backup. If any changes are made to the tables while you copy them, you will have an undefined state in some files at the moment they are ready to be copied to the backup location. While MySQL

## *Backing Up and Restoring MySQL Data*

When MySQL is stopped, it leaves a defined state in all of the files, which will not change during the backup process. This is why a file-based backup will be consistent and suitable to restore the saved state if necessary.

You may have noticed that the binary logs are not saved in this particular recipe. This is because of the fact that the binary logs are not necessary to restore a MySQL instance from scratch. If they are missing when the database is restarted after a restore, a new log file (together with an index file) is created automatically.

It should be noted that if you chose to write your transaction logs to a different location outside of the data directory (using the `innodb_log_group_home_dir` option), you will have to save this directory to the backup location as well, as the log position marker is linked to the InnoDB data files. While InnoDB usually has no problems to recover the database in such a situation after a restore, it is advised to save the transaction logs as well.

The configuration file saved in step 5 is not vital, but it is very helpful when trying to restore data from a backup to have the original configuration at hand. Especially for InnoDB databases, the configuration of the InnoDB table space has to be reproduced identically to prevent non-recoverable errors after restore.

### **There's more...**

In the following sections, we will discuss restrictions of the recipe, an advanced variation of the backup method using LVM snapshots, and a few hints on how to restore data from a backup.

### **Understanding the restrictions of the file-based backup method**

The backup method described in the above recipe typically delivers very good performance and excellent duration predictability (especially on a restore). On the downside, the nature of backing up binary files makes it more vulnerable to data corruption problems (because of the fact that the database server needs to be restarted). If a binary file gets corrupted (for example a table that is not accessed on a regular basis), you will not necessarily notice this kind of backup. By the time the file gets used and the error shows up, all your data generations might already contain a corrupted version of the file, basically leaving you with no usable backup for the data stored within the respective file.

This is why we recommend complementing the file-based backup with a dump-based backup as described in the following recipe. This forces MySQL to fully read every table so that corruption is more likely to be discovered. To make use of this fact, the backup should be monitored, as errors during backup might indicate data corruption and should be checked for further evaluation.

---

This dump might be produced less often, but with a frequency that performs a dump of all the retained backup generations of the file-based backups get overwritten again in the cycle. For example, if you perform a daily file-based backup and keep seven generations of these, we recommend performing an additional dump-based backup once a week.

## Backing up using LVM snapshots

The major disadvantage of the copy-all approach is that a database shutdown is required. In many situations this is not a feasible option. To avoid this restriction, it is sometimes necessary to create a snapshot of the file system (typically an LVM snapshot using the Linux Volume Manager). We advise you to use this approach with caution because a snapshot does not guarantee that the state the files were in when the snapshot was created is a valid state from which MySQL can recover. Even if we assume that in most cases MySQL is able to recover from such a backup state, a backup that is only *likely* to be suitable for a recovery is not sufficient—you have to be absolutely positive about that!

To be sure about the state of the tables at the time the snapshot is done, you have to flush all open changes to the disk. This can be done using a `FLUSH TABLES WITH READ LOCK` command right before the snapshot. Note, however, that in case of long-running transactions this might lead to significant delays for SQL statements that try to write to the database parallel, basically causing similar problems as with a database that was shut down.

After the backup is completed, you have to unlock the tables again by executing an `UNLOCK TABLES` command. Note that if you close the connection that was used for the `FLUSH TABLES WITH READ LOCK` command, the tables are unlocked as well. But if you keep the connection, you do not need the `UNLOCK TABLES` statement, but this also means that the connection has to stay open during the whole backup process!

Furthermore, several tests proved that the write performance of LVM snapshots is significantly lower than that of a standard file-based backup (see for example <http://www.nikhef.nl/~dennisvd/lvmcrash.html> or <http://www.mysqlperformanceblog.com/2009/02/05/disaster-lvm-performance-in-snapshot-mode/>). If there is relevant traffic on the database while the LVM snapshot gets copied to the backup destination, your database will probably suffer from significant performance degradation. A possible remedy to this might be to perform the backup on a specific backup slave as described in one of the following recipes.

## Restoring data from a file-based backup

The restore process for a file-based backup is pretty straightforward: stop the database, copy the directories back to their original location. After starting the database again, it is reset to the state it was in when the backup was performed. Easy as that!

## Backing Up and Restoring MySQL Data

---

### See also

- ▶ *Using replication to perform backups without hurting a production system performance.*
- ▶ *Defining a specific user for backup*

## Creating a SQL dump of all databases

In the previous recipe the file-based backup method was presented. As was already mentioned there, this approach requires some caveats. Thus we recommend to perform additional dump-based backups, which store the content of your database as SQL files that can easily be read and are less subject to (unrecoverable) data corruption. This recipe shows you how to create such a dump-based backup.

### Getting ready

To follow the steps in this recipe, you will need a user that has sufficient permissions to perform a backup (most importantly the `SELECT` and `LOCK TABLES` privileges). We will assume a user named `backup_usr` (see *Defining a specific user for backup* in Chapter 1).

As before, you need an additional destination directory with sufficient free space to store the backup. We will use the directory `D:\MySQLBackup\` as the target directory.

### How to do it...

Execute the following command from the command line:

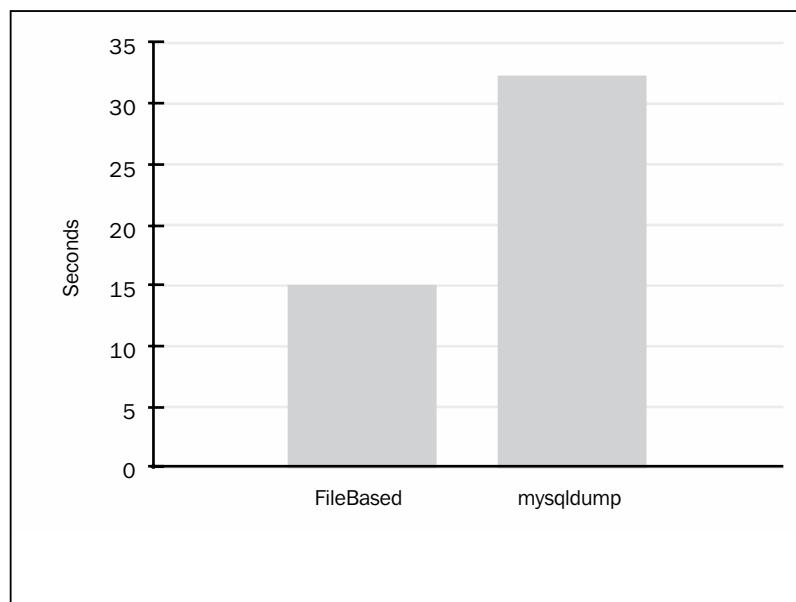
```
C:\>mysqldump -u backup_usr -p"B4ckM3Up!" --all-databases > "D:\MySQLBackup\MySQLDumpAllDatabases.sql"
```

### How it works...

While it might seem somewhat awkward to call a single command a recipe, the decision involved in this command justify this decision. The command-line statement consists of two basic parts: the `mysqldump` command itself and the output redirection clause (`> D:\...\\MySQLDumpAllDatabases.sql`), which writes the resulting dump to the given file. If you take a closer look at the `mysqldump` command, you will notice the `-u` and `-p` that are used to pass the user credentials to use for performing the backup. The next option is `--all-databases`, which tells `mysqldump` to write the data of all databases (including the `mysql` schema) to the dump.

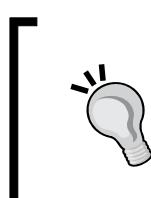
Now what happens on execution of this command? First of all, the default option --all-databases is applied in addition to the given parameters. It is active if no other option is listed except for --compact. This enables several sensible settings, most notably the --lock-tables option that locks all tables (using the `LOCK TABLE ... READ LOCAL` semantics) before they are dumped. This prevents concurrent modification to the tables while data is being dumped (which could result in a backup file with inconsistent data across tables). Afterwards, one database after the other is dumped. For each database, the structure and data of each of the tables (in alphabetical order) is dumped and written to the output file in as SQL `CREATE TABLE` and `INSERT` statements. The resulting dump file is a valid SQL file that can be executed on an empty MySQL installation to restore the database.

It should be noted that the performance of creating a backup using `mysqldump` is not as good as creating a file-based backup (as discussed in the previous recipe). The following chart compares the time needed for backing up the `employees` example database:



## There's more...

The default behavior of `mysqldump` is to lock all tables of each database before the dump starts. This has the goal of preventing inconsistencies inside the dump when concurrent modifications take place during the dump. Unfortunately, this works only within certain limits and might also have a negative impact on the database operations. The following sections will discuss these caveats.



Please note that all additional options to `mysqldump` mentioned in the previous section have to be placed before the `--all-databases` clause (you will encounter errors otherwise).

## Backing Up and Restoring MySQL Data

---

### **Preventing locking issues by using InnoDB storage engine**

The locking mechanism used by `mysqldump` tries to prevent concurrent data modifications during the dump. This obviously can cause significant delays due to table locks when an application tries to concurrently write to the locked tables. Depending on the size of the database and the correlated duration of the dump, this could pose a problem. If that's the case, you should consider using the InnoDB storage engine instead of MyISAM tables (we recommend anyway for most applications), as InnoDB allows dumping a consistent snapshot of the database basically without any potentially long-lasting locks. Another alternative is to use a separate replication slave as the backup source (as described in the *Using replication to perform backups without hurting a production system's performance* recipe), which allows for backups without negative consequences for the master database server.

### **Creating consistent dumps of InnoDB tables**

If you deal mainly with InnoDB tables (or other tables using a different transactional storage engine), you should add the `--single-transaction` option to the previous `mysqldump` command. This option creates a consistent snapshot of all InnoDB tables, which is not written to the dump. But please note that this option ensures integrity only for transactional tables! For MyISAM tables, this option increases the risk of dumping inconsistent data, as `--single-transaction` disables the `--lock-tables` and `--lock-all-tables` option.

One should also be aware that the InnoDB snapshots are not able to cope with commands that alter the table structure (such as `ALTER TABLE`, `DROP TABLE`, `CREATE TABLE` or `RENAME TABLE`). You should make sure that these statements are not executed during a dump.

### **Preventing dump inconsistency across databases**

The default `--lock-tables` option causes `mysqldump` to lock all tables for each database separately. In case of dependencies across schemata, this might lead to inconsistent data again. Think of an entry that gets changed in schema A. The corresponding entry in schema B—which holds a reference to the first entry—is updated only after the dump of the second schema is finished. If `mysqldump` dumps schema B first, then schema A, you will have an orphan entry in schema A.

To prevent this problem, you could pass the `--lock-all-tables` option to `mysqldump`, which results in locks across all databases. As a downside, this approach produces longer lasting locks than the default method, so you should use it only if needed.

---

## Including binary log position in the dump

If your database has binlogging enabled, then you should use an additional option to the mysqldump command: `--master-data=2`. It includes the log position of your database server in your dump file. This is extremely useful when trying to perform a point-in-time recovery using the dump and additional binary log files (as described in the following recipe). This option works only if binlogging is enabled.

Please note that this option implicitly enables `--lock-all-tables` (unless used in conjunction with `--single-transaction`, which disables both table lock options).

## Performing consistent dumps for binary data

When using binary data in columns of type BINARY, VARBINARY, BLOB (in all sizes), you should additionally include the `--hex-blob` option for mysqldump to ensure it is dumped correctly. Otherwise you might encounter problems with messed up data. A common cause for this problem is the conversion of special bytes sequences like line breaks that occur when exporting and importing data. The `--hex-blob` option circumvents this conversion for binary columns.

## Reducing performance impacts by using multiple disks

When creating a dump, your computer has to perform both read and write operations in parallel—the database is read, and the dump is written. If both operations access the same physical disk, it results in a high load on the disk and many head movements, which in turn negatively impacts the performance. If you have multiple physical disks available (a logical drive or partition on the same physical disk will not help), you can increase dump performance considerably by targeting the dump to a different disk than the disk that holds your database data.

### See also

- ▶ *Defining a specific user for backup*
- ▶ *Using replication to perform backups without hurting a production system's performance*

*Backing Up and Restoring MySQL Data*

---

## Creating a SQL dump of specific database

This recipe shows you how to select specific databases for backup. This might be useful for example to separate the backup processes for different databases if these have different (or even worse contradictory) backup policies.

### Getting ready

As this is only a slight adaption of the previous recipe, the preconditions are almost the same again, you have to have a MySQL user account (here: `backup_usr`) with sufficient rights to perform a database dump, and a target directory (here: `D:\MySQLBackup\`).

Additionally, you have to be sure that the database(s) you want to dump have no dependencies on databases that will not be part of the dump. This could result in the dump being not self-contained and might cause data inconsistencies in case of a restore.

Throughout this recipe, we will assume that you want to dump the content of the two databases `employees` and `suppliers`.

### How to do it...

Execute the following command from the command line:

```
C:\>mysqldump -u backup_usr -p"B4ckM3Up!" --databases employees > "D:\MySQLBackup\MySQLDump_Employees_Suppliers.sql"
```

### How it works...

The command is basically identical to the `mysqldump` command presented in the previous recipe. The only difference is that the `--all-databases` clause is replaced by the `--databases` option. This option is followed by a database (or a list of databases separated by blanks) that should be included in the resulting dump.

The additional options mentioned in the previous recipe are applicable for the `--databases` variant as well. You have to make sure, however, that the `--databases` option is the last option to `mysqldump`.

## There's more...

An advantage of dumping databases separately is that a database can be restored without affecting other databases. Although a full restore of all databases takes a little more time, in most situations this is outweighed by the additional flexibility of having separate backups. Another advantage is that the dump files are typically less clumsy and are easier to store because of their reduced size.

### **Considering the side effects of automated backup**

For automated backups, the idea of backing up each database to its own dump file has one side effect that should be considered: a typical simple script that basically contains a single `mysqldump` command for each database has to be adapted every time a database is created or deleted. In a dynamic environment where the schema catalogue is subject to constant changes, you should consider writing a script that reads the available databases (for example using `SHOW DATABASES`) and iterates over the resulting list to dump each database.

### **Increasing performance by dumping in parallel**

To leverage current computers with multi-core processors, one approach to reduce the time needed for backing up a database is to parallelize the work. If you start several processes concurrently, you might be able to achieve faster backups. However, the degree of parallelism largely depends on the available resources. If the data throughputs of your drives are the limiting factor, you will not be able to significantly speed up your backup, but in the case of fast disks (or multiple disks, as noted in the previous recipe) concurrency might help to save some time. Try concurrently backing up each database to separate disks (if applicable) to reach maximum performance.

If you happen to use a Linux server for your MySQL, you might also want to have a look at `mk-parallel-dump` from Maatkit that helps you to dump table sets in parallel (see <http://www.maatkit.org/doc/mk-parallel-dump.html>). Unfortunately, this is not yet available for Windows users. Also, note that the website explicitly states that `mk-parallel-dump` is not a backup program, so you might want to use a different strategy besides `mk-parallel-dump` (just to be sure).

## See also

- ▶ *Defining a specific user for backup*

*Backing Up and Restoring MySQL Data*

---

## Creating a SQL dump of specific tables

This recipe will show you how to dump only a portion of a database by selecting specific tables to include in the dump, which comes in handy if you have specific backup requirements for special tables (for example higher backup frequency) that differ from the backup requirements of the rest of the database.

### Getting ready

Again, this recipe is very similar to the previous one. We will need a suitable MySQL user account (`backup_usr`), and a target directory with sufficient space (`D:\MySQLBackup`).

As we restrict the dump to specific tables, you should additionally check that the tables you want to include in this dump have no dependencies on other tables or databases that will not be included in the resulting dump.

For the following instruction we assume that the table `departments` from the schema `employees` should be written to the dump.

### How to do it...

Execute the following command from the command line:

```
C:\>mysqldump -u backup_usr -p"B4ckM3Up!" employees departments
MySQLBackup\MySQLDump_Departments.sql"
```

### How it works...

The above command works just as explained in the previous recipe, with the only difference being that no specific option (like `--all-databases` or `--databases`) is given to indicate the desired source databases to `mysqldump`. In this case, `mysqldump` reads the first parameter (that is: starting without a dash) as the source database, followed by a list of tables to dump (separated by spaces). In the above example, this list consists of only one table, but others could be passed as well.

### There's more...

While this approach is sometimes very nice to produce backups that are restricted to specific relevant tables only, we recommend you not to use this for automated backups. The reason for this recommendation is that data models are typically subject to a certain extent of evolution. If, for example, a new table is introduced, and this table is referenced by other tables contained in the backup, you should include the new table in the backup as well. Furthermore,

specific structure are typically ignored or overlooked, leaving you with a backup that only a part of the required data. This is why we recommend dumping the schema (schemata) as a whole to prevent problems arising from subsequent data model ev

## See also

- ▶ *Defining a specific user for backup*

# Compressing SQL dumps on-the-fly

When the data stored in your database grows over time, the backups get bigger as often will want to reduce the space required for your backups to reduce the disk (or storage requirements. If you create a backup using `mysqldump`, you can reduce the size of your backups considerably. The following recipe will show you how to achieve this.

## Getting ready

In addition to the prerequisites listed in the previous three recipes (a suitable MySQL account and a target directory), we also need an installed version of **gzip** (see <http://www.gzip.org/#exe>), which is a widely used open source compression program.

In this recipe, we will produce a dump of all databases and compress it before it is written to the disk.

## How to do it...

Execute the following command from the command line:

```
C:\>mysqldump -u backup_usr -p"B4ckM3Up!" --all-databases | gzip >D:\MySQLBackup\MySQLDumpAllDatabases.sql.gz"
```

## How it works...

The only difference in the above command line in comparison to the previous recipe (apart from the slightly changed target file name to reflect the compressed content) lies within the `| gzip --fast` portion. This redirects the output of the `mysqldump` command to the `gzip` program, which compresses data on-the-fly. The compressed data stream is then written to the given file, resulting in a significantly reduced size.

## There's more...

## *Backing Up and Restoring MySQL Data*

---

### **Achieving better compression ratio**

As a ballpark figure for the compression ratio for typical databases, the compressed file will probably be reduced to one-third of its uncompressed size. To achieve better compression ratios at the expense of reduced performance and higher CPU load, you could also leave off the --fast option of **gzip**, resulting in a compressed file that will take less than 25% of the original size. If the size of the backup is an extremely critical issue for you, you might want to try the --best option as a replacement for --fast, but be warned that this might result in a dramatic increase in execution times with mostly minimal improvements in size; therefore, I recommend not using this option.

### **Considering performance factors**

With the additional compression part added to the above command, you would intuitively expect that this necessarily has a negative impact on the dump performance. Surprisingly, this is not always the case, as the reduced size of the resulting file also decreases the amount of data written to your disk. This reduction in disk I/O might make up for the additional time spent on compression work or even cause a performance improvement! You should check whether on-the-fly compression has any performance impact with your configuration.

A rule of thumb: the slower the target disk, and the faster your processors, the better the compression. And if you want to take system load into account as well: If you experience high I/O load on your target disk and low CPU load, it is well possible that the performance gain of on-the-fly compression is not too significant.

### **Considering data robustness and tool availability**

With the on-the-fly compression approach, the resulting dump files cannot be read directly, as they have to be uncompressing them, which is why you need to have the **gzip** tool at hand for the restoration process as well.

Moreover, due to the fact that the compressed files have a compressed binary format, they are much more susceptible to faults caused by data corruption. If data corruption occurs, it is typically more difficult to extract valid data, and the amount of data that is irreversibly lost is typically a good deal bigger.

If a partial data loss occurs for a standard dump (SQL) file, you will still be able to read portions of the data because you are able to read the file, extract the intact data, and even correct the errors.

However, in case of a partial data loss on the compressed file the uncompression process will simply fail, basically leaving you without a usable backup since manual corrections of a corrupted compressed file are almost impossible. You should keep this in mind when deciding what is the best solution to perform a backup.

---

## Achieving better compression with alternative tools

As an alternative to **gzip**, you could also try similar compression utilities like **bzip2** (<http://www.bzip.org/downloads.html>), **p7zip** (for Linux, <http://p7zip.sourceforge.net/>), or **7-Zip** (for Windows, <http://www.7-zip.org/>), which provide a better compression than **gzip**.

### See also

- ▶ *Defining a specific user for backup*

## Rotating and purging binary logs

If you have binary logging enabled, the `binlog` files contain all changes made to your database over time. These are required for replication, but they can also be used for data after a crash. This is why we strongly encourage you to enable binary logging if you do not use replication. If you still have access to the `binlogs` produced between your last backup and the moment the disaster occurred, you can basically recover everything without losing any data at all.

To be able to do so, the binary logs should regularly be saved to a different location (on a different drive or tape media). Moreover, you will need to remove `binlog` files that are no longer needed to prevent the disk from getting full. To be able to copy the relevant files, this recipe will show you how to make sure no concurrent access is active when backing up the data.

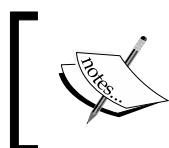
### Getting ready

We assume that you have a regular backup process in place, which produces a database dump based on a MySQL dump. Your application requires that in case of a disaster you will be able to recover all data older than half an hour before the crash. To achieve this, you will need to save the binary logs at least every 30 minutes to a location that will not be affected by a possible crash on your database server.

In addition, let's assume that you will not need any `binlogs` older than 7 days for recovery. This in turn means that any replication slave with a data set older than 7 days will need to re-enter replication again to catch up with the master (but this typically makes no difference anyway). Thus, all `binlogs` older than 7 days can be deleted.

## Backing Up and Restoring MySQL Data

You will again need an appropriate MySQL user account (`backup_usr`) and a place to copy the binlogs to. The recipe assumes a Windows system and a target directory `D:\MySQLBackup\binlogs\`. The `log-bin` parameter of the MySQL configuration is assumed to have the value `C:/MySQL/Binlogs/bluebox`.



The recipe is applicable to Windows systems. Please refer to the *There's more...* section for a Linux/Unix variant.

### How to do it...

Establish a scheduled task to perform the following commands at least every 30 minutes in the context of the directory `C:\MySQL\Binlogs\`:

```
copy bluebox.index idx.tmp
mysql -u backup_usr -p"B4ckM3Up!" -e "FLUSH LOGS;
PURGE BINARY LOGS BEFORE TIMESTAMPADD(DAY, -7, NOW());
FOR /F %i in (idx.tmp) DO xcopy /D %i D:\MySQLBackup\binlogs\;
DEL idx.tmp"
```

### How it works...

Let's take a look at each step of this recipe.

The line `copy bluebox.index idx.tmp` makes a (temporary) copy of the index file. This file contains the index list of all binlog files, which is needed for future reference.

The next line executes the MySQL commands `FLUSH LOGS` and `PURGE BINARY LOGS BEFORE TIMESTAMPADD(DAY, -7, NOW())` on the database. `FLUSH LOGS` causes MySQL to close and reopen all log files and to create a new binlog file with an increased counter. This has the effect that write access to the previous binlog is finished, so we can safely back it up to another location without risking access conflicts.

The `PURGE BINARY LOGS BEFORE TIMESTAMPADD(DAY, -7, NOW())` command causes MySQL to delete all binlog files older than 7 days, which makes sure that the binlog directory does not fill up over time.

---

The loop command FOR /F %i in (idx.tmp) DO uses the old index file that is in the first command to iterate over all binlog files. The xcopy /D %i D:\MySQL\binlogs\ command copies each binlog file to the backup target location. As the file was copied before we executed the PURGE BINARY LOGS command, the list of files does not include the current binlog file, which is still actively accessed and possibly written by MySQL. So we can be pretty sure no write access to the files in this list occurs. The /Y option of the xcopy command is very useful to prevent the same files from being copied over and over again. This option tells xcopy to copy files only if the source file is newer than the existing target file, so unchanged files that are already present at the target directory will not be copied again.

The final command then simply cleans up the temporary file, which we created in the previous step, so we leave the place nice and clean.

## There's more...

In this section, we will discuss some potential risks and a variant of the recipe for Linux systems.

### **Rotating and purging binary logs on Linux systems**

Due to syntax differences in the command-line interface, the above recipe is applicable only to Windows systems only. To perform the same task for Linux systems, use the following approach (we assume /var/mysql\_backup/binlogs as the target directory and /var/log/mysql as the log-bin parameter):

Establish a cron job to perform the following commands at least every 30 minutes:

```
#!/bin/bash
cd /var/log/mysql
cp bluebox.index idx.tmp
mysql -u backup_usr -p"B4ckM3Up!" -e "FLUSH LOGS; PURGE BINARY LOGS BEFORE TIMESTAMPADD(DAY, -7, NOW());"
cat idx.tmp | while read line; do
    cp -u $line backup/
done
rm idx.tmp
```

## Backing Up and Restoring MySQL Data

When comparing the script to the equivalent Windows version, you will notice some differences (we will not discuss the first two lines here, but concentrate on the actual operations). First of all, the Windows copy command is replaced by the Linux `cp` command. The loop command is different as well: `cat idx.tmp | while read line; do cp $line /var/lib/mysql/`` is replaced by the Windows command `FOR /F %i IN (idx.tmp) DO cp %i /var/lib/mysql/``. It also iterates over all files as listed in the old index file that we copied with the first `cp` command. The `/I` option of the Windows' `xcopy` is reflected by the `-u` option of `cp`, which only copies files if they have been modified since the last copy or if the corresponding file at the target directory is older than the file to be copied.

### **Considering risks of data loss**

The whole mechanism of copying the `binlog` files from one place to the other makes little sense if the two directories are located on the same physical disk, because in the event of a disk failure data in both places would be lost. You should at least have two separate disks to store these directories on. Depending on your requirements, you could also consider copying to another host to cover the risk of fire or other disasters, which could damage the internal physical disks of your machine.

### **Ensuring sufficient disk space**

You will have to make sure, of course, that some kind of backup and deletion mechanism is established at the target directory as well to prevent this disk running out of space. In this scenario, an archiving tool might scan this directory once a day, save the files to a tape and delete the archived files.

#### **See also**

- ▶ *Defining a specific user for backup*

## **Using replication to perform backups without hurting a production system's performance**

A backup always produces a significant load on the server the data is read from. In fact, depending on the way the backup is performed and the storage engine used for your tables, locking situations can occur, which might cause major problems for your application. One method to circumvent both of these problems is to back up the data from a replicated client instead of the server. This recipe will show you how to achieve that.

Please note that we do not consider replication itself as a backup technique! Although it is a sensible approach to deal with possible failures and to reduce downtimes, it is not a replacement for creating regular backups.

## Getting ready

To be able to follow this recipe, you will of course need a MySQL instance that acts as a replication client. Without a working replication set up, this recipe is not helpful.

In addition, you need a MySQL account with appropriate privileges (we assume the user used `backup_usr` here). And of course, you will need sufficient space in the target to save the backup to. We will assume the data directory of the slave instance is `C:\MySQLSlave\Data\`, and the backup target directory is `D:\MySQLBackup\`.

In this recipe, we will refer to the previous recipes to back up data either using the `mysqldump` tool or by copying the data files—both approaches work with this technique.

## How to do it...

1. On the replication client, execute the following statement on the command line to stop replication processing:

```
C:\>mysql -u backup_usr -p"B4ckM3Up!" -e"STOP SLAVE SQL_THREAD"
```

2. Perform your backup, either using `mysqldump` or by copying the data files presented in the previous recipes.
3. If the backup in step 2 was performed using `mysqldump`, copy the replicated data to the backup target directory as well by performing the following commands (skip this if you copied the data files in step 2):

```
C:\>copy /Y C:\MySQLSlave\Data\*.info D:\MySQLBackup\C:\>copy /Y C:\MySQLSlave\Data\*relay-bin.* D:\MySQLBackup\
```

4. Execute the following command to start replication processing again: `c:\>mysql -u backup_usr -p"B4ckM3Up!" -e"START SLAVE SQL_THREAD"`

## How it works...

The recipe basically consists of three parts: by performing the operations described in step 1, replication is disabled; in steps 2 and 3, the backup itself is performed; in step 4, replication is enabled again.

Let us have a look at the backup part first. In step 2, the backup of the data stored in the slave database is done as shown in the previous recipes. The addition of step 3 is to also back up the replication state of the slave. In detail, this copies the relay log ([host]-relay-bin.index and [hostname]-relay-bin.00x), the relay-log.info file, and the master.info files. These files are necessary to recover the slave from a backup because without these files, it is very hard to establish a working replication mechanism between the master. If your intention is to perform a backup only for restoring the master, you can skip step 3.

## Backing Up and Restoring MySQL Data

That's about it concerning the backup itself. Now what is the motivation for steps 1 through 3? The deactivation of the slave replication in step 1 is necessary to prevent inconsistencies during the backup. As discussed before, both the copy approach and the `mysqldump` command have restrictions concerning data consistencies for non-transactional tables (like MyISAM) when concurrent updates occur while the data is read. You can tackle this problem by using example using locking options. But for a master-slave constellation, this could (in a worst-case scenario) cause the replication to break! By stopping the SQL slave thread, the binary log is still read from the master, but it is no longer processed. This prevents any concurrent modifications and the data remains in a consistent state throughout the backup process. After the backup is done, step 4 activates replication again, and the slave will start catching up the remaining changes from the master up to the point where master and slave reach a synchronous state.

In this scenario, all backup operations (together with the possible performance degradations they might provoke) are performed on the client. By this, a backup is produced, but the replication master is not affected with respect to performance issues. Note, however, that the client is out of sync with the master throughout the backup, and (depending on the time when the backup is produced) locking issues might occur as well. This is the reason why this method is not intended for use with clients that are actively accessed by applications (for example, for load balancing reasons). You are also not safe when creating the backup on a slave that is used as a hot standby for a fail-over scenario, and which is not actively accessed by your application: if the master crashes while the backup is running on the slave, the system is not ready for fail-over. The backup had to disable replication first, so your hot standby has a data set different from the master. Only if the replication is activated again, and the master is processed, can a fail-over safely take place. Whether this is acceptable has to be evaluated individually, but to completely separate your application from the backup, you should consider establishing a slave dedicated to backup only.

### See also

- ▶ *Defining a specific user for backup*
- ▶ *Copying all data files to a backup location*
- ▶ *Creating a SQL dump of all databases*
- ▶ *Creating a SQL dump of specific databases*
- ▶ *Creating a SQL dump of specific tables*

# Restoring data from a dump to a previous backed-up state

In the previous recipes, we dealt with creating a backup of the existing data and code. The sole reason for this is to be able to restore the data again if required. This recipe shows you how to restore the data created using a dump. The recipe is suitable both for full dumps and for dumps that contain only the data of specific databases.

## Getting ready

To follow the steps in this recipe, you will need a running MySQL instance and a MySQL account that has the privileges necessary to restore all data. In a default MySQL configuration, the root user can be used, but throughout this recipe we use the admin4mysql account with password As, ysp4M (see *Defining a specific user for administrative tasks* in Chapter 1). MySQL installation should have sufficient space available to store the data from the dump and no application connections should be active throughout the restore. And (of course) you will need the dump to restore the data from. In this recipe, we assume the dump is stored at D:\MySQLBackup\MySQLDumpAllDatabases.sql.

## How to do it...

1. Connect the mysql command-line client to your MySQL instance using the admin4mysql account:

```
C:\>mysql -u admin4mysql -p"As, ysp4M"
```

2. Restore the data from the dump by executing the following SQL command:

```
mysql>source D:\MySQLBackup\MySQLDumpAllDatabases.sql
```

## How it works...

Again, this is a pretty straightforward recipe: connect to the database (step 1) and restore the data from the dump (step 2).

The use of the mysql command-line client is not absolutely necessary, but please note that you will not be able to issue the source command from any other SQL client. This is because it is not a regular SQL statement, but a key word recognized by the mysql command-line client itself. You could of course use another SQL client to read in the whole dump file and execute its commands sequentially (for example using the **Open Script...** menu entry in MySQL Query Browser), but the source command is specific to mysql.

## Backing Up and Restoring MySQL Data

If the dump you want to restore contains only specific tables (as described in *Creating a dump of specific tables*), use the following statement to select the target database before restoring the tables into:

```
mysql>use employees;
```

You need to do so because otherwise the tables get inserted to the current default database. Note that the dump itself contains no information about the database the dump stems from.

Please note that after performing the above steps, the content of your database is not necessarily absolutely identical to the state of the database the backup was created from. All databases that were already present before the dump are restored, but any that were not covered by the content of the dump are left unchanged. If your dump for example contains data for the `foo` database only and the database you read the dump into has a database `bar` (which was, for example, created after the dump was produced), the database `bar` will remain unchanged. To be sure you recover to an identical state, you have to:

1. Perform a full backup (using the `--all-databases` option of `mysqldump`)
2. Make sure that the target database is completely empty when restoring.

### There's more...

In the following sections, we will discuss some advanced aspects of restoring data, such as working with compressed dumps and avoiding typical performance problems.

## Restoring compressed dumps

When restoring a database from a compressed dump (for example, as created according to the *Compressing SQL dumps on-the-fly* recipe), the dump needs to be decompressed before restoring it to the database. You could either decompress the file on the disk and subsequently use the preceding recipe without any changes, or decompress on-the-fly while restoring the data in one step:

```
C:\> gzip --decompress --stdout D:\MysqlBackup\DumpAllDBs.sql.gz | mysql -u admin4mysql -p"As,ysp4M"
```

The `--decompress` option tells **gzip** to revert the compression (obviously), while the `--stdout` option makes sure that the compressed file itself is left unchanged, but the decompressed content is written to the `stdout` device. The pipe symbol `|` redirects the output from `stdout` to the `mysql` command, which then receives the decompressed content of the dump and executes the statements contained therein.



If you used a different compression tool like **bzip2** or **7zip** when creating the dump, you have to adapt the above command.

---

## **Temporarily disabling binlogs to save time and space**

If binlogging is enabled on the MySQL instance you restore, all statements from the log are not only executed against your database, but also written to the binlogs. This is an advantage that replication slaves perform the restore as well, but sometimes this is not necessary (or wanted). For example, if you want to keep the clients unchanged for testing purposes, or if you want to save disk space in case the restore fails, or if you do not use binlogging for replication but only for backup purposes. With MySQL 5.1, there are several restore options, you might want to temporarily disable binlogging when processing a backup file. This might save you a significant amount of disk space (depending on the size of the log files) and is also better for performance because the write access to the binlogs is omitted. To temporarily disable binlogging, you have to prepend the following command in step 2:

```
mysql> SET sql_log_bin=0;
mysql> source D:\...
```

This command disables binlogging for the current connection only. So as soon as the connection is terminated, the binlogs will again contain all changes made to your database.

Please note that for this statement your user will require SUPER privileges. This privilege is usually required to perform a restore, but as restoring data is typically a task for the system administrator, it is safe to assume that an account with this privilege is available to the person in charge.

## **Increasing recovery performance by using parallel restoration**

As parallelization is an approach to increase backup speed, it can be used for better performance for restoration as well. With multiple concurrent restore processes, your system might be reinstalled faster. However, whether this is successful or not again depends on the resources available. If the speed of your restore is limited mainly by the drive throughput, the benefit will be basically non-existent. If you have fast (or multiple) disks, however, you will see major advantages. Concurrently restoring each database separately could therefore be the way to maximize performance.

For Linux servers, `mk-parallel-restore` from Maatkit is available, which supports restoring data in parallel (see <http://www.maatkit.org/doc/mk-parallel.html>). However, a Windows version is not available.

## **Restoring tables excluding potentially very large tables**

When dealing with large data sets, you sometimes encounter performance problems that require manual intervention. One real-world example of performance problems is a table of massive tables, which might cause severe performance degradations.

## *Backing Up and Restoring MySQL Data*

---

Imagine a table with half a billion records (for example containing log data). While such a table, you will notice a dramatic decrease of insert performance throughout the process. The first few percent of the data will be written to the table rather quickly, the restore continues, the write rate will gradually decrease (mostly due to internal maintenance).

In the case of a disaster recovery, you have the choice between restoring all data (the price of extended downtime until the large tables are restored completely as well) or a partial restore, deliberately skipping the restore of the data from the large table, by advantage of a faster restore. The decision has to be made on an application level. In the example of log data, it is often feasible to simply exclude the log data from the restore on the sake of a faster recovery.

Let us assume we have a very large dump file that contains the data you actually want to recover (D:\MySQLBackup\MySQLDump.sql) along with a very large table (here op\_detail). To be able to cope with the data contained in this potentially mammoth table, you will need access to the grep command (for Windows users: see <http://unx-sourceforge.net> for an implementation of the common GNU tools).

The following command will restore the data from the dump excluding the data from the op\_detail table:

```
C:\>grep --invert-match "^INSERT INTO .op_detail. VALUES .*"  
D:\MySQLBackup\MySQLDump.sql | mysql -u admin4mysql -p"As,ysp4M"
```

This command uses grep to filter all lines including an `INSERT INTO 'op_detail'` at the start of the line, leaving all other lines untouched. All other lines are piped to the command-line client that imports the remaining dump.

Please note that if data has references to entries in the excluded table, the import dump will not fail. In this case, you would end up with data inconsistencies that might errors much later, so make sure to exclude only tables that are not referenced by foreign keys. Otherwise, you have to search for possible foreign key constraint violations manually. Ideas on how to achieve this are presented for example at <http://dev.mysql.com/doc/refman/5.1/en/innodb-foreign-key-constraints.html>.

### **See also**

- ▶ *Defining a specific user for administrative tasks*
- ▶ *Creating a SQL dump of all databases*
- ▶ *Creating a SQL dump of specific databases*

# Performing a point-in-time recovery using the binary logs

The previous recipes dealt with how to recover data from a backup. After recovery, the database should be in the same condition as it was when the backup was created. Unfortunately, any data that was changed after backup creation is not restored.

To restore the data to the point in time before the recovery, you can use the binary logs. If binary logs were created between backup and recovery. As mentioned in the *Rotating and purging binary logs* recipe, we suggest enabling binary logging even if you do not have a replication setup, as this gives you extended options in backup.

In this recipe, we will discuss how to use the binary logs to restore data up to the last possible point in time before the data was lost.

## Getting ready

To be able to restore the remaining data using the binary logging information, we obviously need the binary log files. If all data from your server is lost (for example in case of a hardware failure), the binary logs are hopefully available from a tape or any other media that is not affected by the data loss. If only your database data is corrupt, you often have the binary logs still stored on your server's disk.

If the binary logs are still present, you should copy the binary log files to a different location *before you start recovering from a dump!* This way you have all binary log files available exactly the state from the point in time before the recovery.

We assume the binary logs (either as a copy of the binlog files from your server or recovered from backup media) to be stored under `C:\tmp\binlogs\`.

To read the changes stored in the binary logs into the database, you need an appropriate MySQL user again (here: `admin4mysql` with password `As, ysp4M`).

And most importantly, you need the position at which recovery from the binary logs should start. The dump your database was restored from should have been created with the `--master-data` option (see the *There's more...* section of the *Creating an SQL dump of all databases* recipe), otherwise this information is not easily available. You could try to manually identify the correct starting position (for example by checking the timestamp of creation or last change or inspecting the files using the `mysqlbinlog` tool). But if you rely on using the binary logs for recovery, you should definitely use the `--master-data` option to create a dump.

## *Backing Up and Restoring MySQL Data*

Find out about the log position by using the `more` command to display the first few dump. Look for a line like the following:

```
CHANGE MASTER TO MASTER_LOG_FILE='myhost.000005', MASTER_LOG_POS=
```

In this line, note the values noted after `MASTER_LOG_FILE` and `MASTER_LOG_POS` reference. We, furthermore, assume that you have binary log files through `myhost.000005` available for recovery. Throughout the following recipe, we will use the above noted. You have to replace them with your respective values accordingly.

### **How to do it...**

1. Recover the data from the dump according to the previous recipes.
2. Execute the following command from the command line in the context of `root` on `C:\tmp\binlogs\` (change directory accordingly):

```
C:\>mysqlbinlog --start-position=201 myhost.000005 myhost.000007 myhost.000008 | mysql -u admin4mysql -p"As,
```

### **How it works...**

The command executed in step 2 reads the given binary logs, translates the changes stored therein into SQL commands and passes these to the MySQL database. The `--start-position` option tells `mysqlbinlog` to begin reading the first given file (`myhost.000005`) at position 201, which starts the recovery process at exactly the point where the dump was created. After this, the next files (000006 through 000008) are applied completely into the database.

### **There's more...**

The following sections address partial restores by introducing means to extend the restore either to restore specific databases, or to restore data only up to a certain point in time.

### **Restoring only a specific database**

If you have to recover only the data of a specific database but want to leave all other databases unchanged, you must not import the full content of the binary log files because these might contain changes of other databases as well. In this case, add the option `--database=dbname` to `mysqlbinlog`. This restricts the SQL statements passed to the MySQL instance to the given database.

---

## Determining the exact location of a failure and restoring that point

You sometimes have the problem where data loss is not necessarily caused by a system error, but by malicious SQL statements. If, for example, an accidental `DROP` statement deleted all of your data, then you will have to recover your database as well. In this situation, directly reprocessing the binary logs is a bad idea because this would also execute the `DROP` statement again, leaving you with a broken database again. In these cases, you will have to manually inspect the binary logs to find out about the specific statements that caused the problem.

To inspect the binary log files, you can create a SQL file from them with a simple `mysqlbinlog myhost.00000x > binlog.sql` command. You can then open the resulting `binlog.sql` file in an editor and scan through the commands. If your binary log files are too large to be opened with an editor, you could make use of the `split` tool (consider using the `--line-bytes` option; see <http://unxutils.sourceforge.net/> for a Windows implementation) to break these files into smaller chunks.

As soon as you have found the first command that you do **not** want to include in the recovery process any more, note the number of the line before it, for example # 1174. This denotes the end position at which processing should stop. To exclude any commands henceforth from the binary log recovery, add a `--stop-position=1174` option to the `mysqlbinlog` command. This will apply to the last given file (in the above example `myhost.000008`) and suppresses processing of all commands following the given position. This should prevent repeating the same errors again, which lead to the problem in the first place.

### See also

- ▶ *Defining a specific user for backup*



# Managing Data

This chapter presents some proven approaches to managing your data beyond the basic operations like `INSERT` or `DELETE` (which we assume the reader to be familiar with). A few recipes will discuss ways of exporting data from and importing data into the database using different file formats. This covers the following topics:

- ▶ Exporting data to a simple CSV file
- ▶ Exporting data to a custom file format
- ▶ Using stored procedures to export repeatedly
- ▶ Importing data from a simple CSV file
- ▶ Importing data from custom file formats
- ▶ Inserting new data and updating data if it already exists
- ▶ Inserting data based on existing database content
- ▶ Deleting all data from large tables
- ▶ Deleting all but a fragment of a large table's data
- ▶ Deleting data incrementally from large tables

## Introduction

The basic set of SQL data manipulation commands (`SELECT`, `INSERT`, `UPDATE`, and `DELETE`) is a well understood means to handle the data stored in your database, and its common basic usage are not difficult to learn. There are situations, however, that require more knowledge about these commands' subtleties. This holds especially true when large amounts of data need to be taken care of—you have to know what you are doing, otherwise you may run into serious performance problems.

## Managing Data

Also, the task of exporting specifically formatted data from or importing external data into a database is a common challenge that can be quite tricky to address.

We are going to address these issues in the following recipes and we will also introduce ways of manipulating data in the database while taking the present contents of the database into account.

We will show solutions to real-world challenges when confronted with data manipulation tasks. It actually took us a while to figure out how to tackle these real-world problems in the most efficient way when first faced with them. Because these strategies are not to be shared openly, this collection of tested methods for these topics might be useful.

## Exporting data to a simple CSV file

While databases are a great tool to store and manage your data, you sometimes need to extract some of the data from your database to use it in another tool (a spreadsheet application being the most prominent example for this). In this recipe, we will show how to utilize the respective MySQL commands for exporting data from a given table in a format that can easily be imported by other programs.

### Getting ready

To step through this recipe, you will need a running MySQL database server and a working installation of a SQL client (like MySQL Query Browser or the `mysql` command line tool). You will also need to identify a suitable export target, which has to meet the following requirements:

- ▶ The MySQL server process must have write access to the target file
- ▶ The target file must not exist



The export target file is located on the machine that runs your MySQL server, not on the client side!



If you do not have file access to the MySQL server, you could instead use the export functions of MySQL clients like *MySQL Query Browser*.

In addition, a user with `FILE` privilege is needed (we will use an account named `sample_install` for the following steps; see also Chapter 8 *Creating an installation account*).

Finally, we need some data to export. Throughout this recipe, we will assume that the export is stored in a table named `table1` inside the database `sample`. As export will use the file `C:/target.csv` (MySQL accepts slashes instead of backslashes in path expressions). This is a file on the machine that runs the MySQL server instance; for example MySQL is assumed to be running on a Windows machine. To access the file from the client, you have to have access to the file (for example, using a file share or executing MySQL client on the same machine as the server).

## How to do it...

1. Connect to the database using the `sample_install` account.
2. Issue the following SQL command:

```
mysql> SELECT * FROM sample.table1 INTO OUTFILE 'C:/target.csv'
FIELDS ENCLOSED BY '"' TERMINATED BY ';' ESCAPED BY '\"'
TERMINATED BY '\r\n';
```

Please note that when using a backslash instead of a slash in the target file's path, you have to use `C:\\target.csv` (double backslash for escaping) instead.



If you do not give a path, but only a file name, the target file will be placed in the data directory of the currently selected schema of your MySQL server.

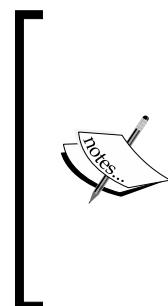
## How it works...

In the previous SQL statement, a file `C:/target.csv` was created, which contains the content of the table `sample.table1`. The file contains a separate line for each row of the table, and each line is terminated by a sequence of a carriage return and a line feed character. This line ending was defined by the `LINES TERMINATED BY '\r\n'` part of the command.

Each line contains the values of each column of the row. The values are separated by semicolons, as stated in the `TERMINATED BY ';'`  clause. Every value is enclosed in double quotation mark ("), which results from the `FIELDS ENCLOSED BY '\"'` part of the command.

When writing the data to the target file, no character conversion takes place; the data is exported using the *binary* character set. This should be kept in mind especially when importing tables with different character sets for some of its values.

## Managing Data



You might wonder why we chose the semicolon instead of a comma as the field separator. This is simply because of a greatly improved Microsoft Excel compatibility (you can simply open the resulting files), without the need to import external data from the files. But you can, however, open these files in a different spreadsheet program (like *OpenOffice.org Calc*) as well. If you do, the usage of semicolons is in contradiction to the notion of a CSV file, thus calling it as a Character Separated File.

The use of double quotes to enclose single values prevents problems when field values contain semicolons (or generally the field separator character). These are not recognized as field separators if they are enclosed in double quotes.

### There's more...

While the previous `SELECT ... INTO OUTFILE` statement will work well in most cases, there are some circumstances in which you still might encounter problems. The following sections show you how to handle some of those.

### Handling errors if the target file already exists

If you try to execute the `SELECT ... INTO OUTFILE` statement twice, an error `File 'target.csv' already exists` occurs. This is due to a security feature in MySQL that makes sure that you cannot overwrite existing files using the `SELECT ... INTO OUTFILE` statement. This makes perfect sense if you think about the consequences. If this were the case, you could overwrite the MySQL data files using a simple `SELECT` because MySQL needs write access to its data directories. As a result, you have to choose different names for each export (or remove old files in advance).

Unfortunately, it is not possible to use a non-constant file name (like a variable) in a `SELECT ... INTO OUTFILE` export statement. If you wish to use different file names, for example, a time stamp as part of the file name, you have to construct the statement inside a variable before executing it:

```
mysql> SET @selInoutfileCmd := concat("SELECT * FROM sample.table
    OUTFILE 'C:/target-", DATE_FORMAT(now(),'%Y-%m-%d_%H%i%s'), ".csv'
    ENCLOSED BY '\"' TERMINATED BY ';' ESCAPED BY '\"' LINES TERMINATED BY '\r\n';");
mysql> PREPARE statement FROM @selInoutfileCmd;
mysql> EXECUTE statement;
```

The first `SET` statement constructs a string, which contains a `SELECT` statement. As you are not allowed to use variables for statements directly, you can construct a string that contains a statement and use variables for this. With the next two lines, you prepare a statement and

## Handling NULL values

Without further handling, NULL values in the data you export using the previous statement would show up as "N" in the resulting file. This combination is not recognized, for example, by Microsoft Excel, which breaks the file (for typical usage). To prevent this, you need to replace NULL entries by appropriate values. Assuming that the table sample.table1 contains a numeric column a and a character column b, you should use the following statement:

```
mysql> SELECT IFNULL(a, 0), IFNULL(b, "NULL") FROM sample.table1
OUTFILE 'C:/target.csv' FIELDS ENCLOSED BY '"' TERMINATED BY ','
BY '"' LINES TERMINATED BY '\r\n';
```

The downside to this approach is that you have to list all fields in which a NULL value might occur.

## Handling line breaks

If you try to export values that contain the same character combination used for line termination in the `SELECT ... INTO OUTFILE` statement, MySQL will try to escape the character combination with the characters defined by the `ESCAPED BY` clause. However, this will not always work the way it is intended. You will typically define `\r\n` as the line separators. With this constellation, values that contain a simple line break `\n` will cause problems, as they are exported without any conversion and can be imported to Microsoft Excel, for example, as they are. If your values happen to contain a combination of carriage return and line feed (`\r\n`) characters, these will be prepended with an escape character ("`\r\n`"), but still the line breaks cannot be imported correctly. Therefore, you need to convert the full line breaks to simple line breaks:

```
mysql> SELECT a, REPLACE(b, '\r\n', '\n') FROM sample.table1
INTO 'C:/target.csv' FIELDS ENCLOSED BY '"' TERMINATED BY ',' ESCAPED BY '"'
LINES TERMINATED BY '\r\n';
```

With this statement, you will export only line breaks `\n`, which are typically accepted by other programs.

## Including headers

For better understanding, you might want to include headers in your target file. You can do this by using a UNION construct:

```
mysql> (SELECT 'Column a', 'Column b') UNION ALL (SELECT * FROM
sample.table1 INTO OUTFILE 'C:/target.csv' FIELDS ENCLOSED BY '"' TERMINATED
BY ',' ESCAPED BY '"' LINES TERMINATED BY '\r\n');
```

The resulting file will contain an additional first line with the given headers from the first `SELECT` clause.

## *Managing Data*

# Exporting data to a custom file format

You sometimes have the task to export data in a special format in order to fulfill the requirements of the recipient of the data. In this recipe, we will show you one way to export data in a format that is beyond the possibilities of the `SELECT ... INTO OUTFILE` format options.

In the following recipe, we will show you how to create an export file in a hypothetical format. This includes the name of the file, a time stamp, a short description of the file's contents, the number of data rows contained in the file in the first four lines. The data portion starts with a header line with names for all columns followed by the actual data rows. Every row should start with a prefix consisting of the hash character (#), the line number, a colon (:), a space. This prefix is followed by the data items separated by pipe (|) characters. The row should end with a dollar sign (\$) (and the line break, of course).

This format is used as an example, but the steps involved can be adapted to more formats if necessary.

# Getting ready

As in the previous recipe, we will need an account with appropriate permissions (E), a SQL client, and a file name for the target file. Again, we will assume an account named sample\_install and we will export data from table sample.table2 (which contains three columns c1, c2, and c3) to a file C:/target.txt in the format mentioned above. We also propose to create a file customExport.sql for the SQL commands using which we will store the SQL commands.

## **How to do it...**

1. Create a new script named `customExport.sql` and add the following statement to it:

```
SET @filename := 'Filename: C:/target.txt';
SET @description := 'Description: This is a test export f
sample.table2 with columns c1, c2, and c3';
SELECT NOW() INTO @timestamp;
SELECT COUNT(*) FROM sample.table2 INTO @rowcount;
SET @rows := CONCAT('Row count: ', @rowcount);
SET @header := '#Row Nr: Column c1 | Column c2 | Column c
SET @counter := 0;
SELECT @filename
    UNION SELECT @description
```

---

```

        UNION SELECT CONCAT('#',
        @counter := @counter + 1,
        ': ',
        CONCAT_WS(' | ', c1, c2, c3),
        '$')
        FROM sample.table2
        INTO OUTFILE 'C:/target.txt';
    
```

2. Connect to the database using the `sample_install` account.
3. Execute the SQL statements from `customExport.sql` (as an alternative to your SQL client, you could also execute the statements using mysql's source command)
4. The target file will look as follows:

```

Filename: C:/target.txt
Description: This is a test export from sample.table2 with
c1, c2, and c3
2009-06-14 13:25:05
Row count: 3
#Row Nr: Column c1 | Column c2 | Column c3 $
#1: 209 | Some text in my test data | Some more text $
#2: 308 | Next test text for testing | Text to test $
#3: 406 | The quick brown fox jumps | Really? $
    
```

## How it works...

Although this solution takes some commands, they are divided into a preparation (the first seven commands that define the user variables) and the actual export command. Of course, you could minimize the number of statements by omitting the user variable definitions, but the resulting statement would be somewhat bulky.

The preparation part simply defines some user variables for later reference. The first command consists of a UNION construct, which basically concatenates the rows required for the file header. The actual data from the table is prepared by the following SELECT clause:

```

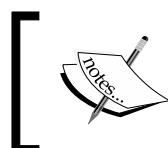
SELECT CONCAT('#', @counter := @counter + 1, ': ', CONCAT_WS
c1, c2, c3), '$') FROM sample.table2
    
```

The CONCAT statement concatenates its parameters; so let us have a look at the statement parts. The clause '#', @counter := @counter + 1, ': ' forms the required line number portion of the rows. The variable `@counter` gets incremented for every row, producing the intended line number. The following `CONCAT_WS` will also concatenate the values, but the first parameter is used as a separator character (`_ws` stands for *with separator*). For every row, this will result in a string with the values of columns (`c1`, `c2` and `c3`) separated by a space.

## Managing Data

---

While this approach allows for the creation of rather complex file formats, it is not appropriate for every situation. For advanced requirements, we encourage the use of other programming techniques beyond the SQL commands (for example, reading and processing the output of a scripting language). This holds especially true when the target file has to be in XML.



For advanced formatting capabilities, consider exporting your data in XML format (using `mysqldump --xml`) and processing the resulting file using an XSLT processor!

### There's more...

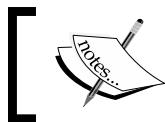
Please note that using parentheses on the `UNION` to clarify the separation of the multiple `SELECT` statements might lead to unexpected problems: the `INTO OUTFILE` clause can only be attached to the last `SELECT` statement of a `UNION` construct. A statement like `(...) UNION (SELECT ...) INTO OUTFILE ...` will **not** work, while `SELECT (...) (SELECT ... INTO OUTFILE)` does. While this might not seem too intuitive, it is a well-documented behavior and not a bug.

## Using stored procedures to export repeatedly

In some situations, you will need data exports on a regular basis, for example, to provide an external system with data for daily reports. One possibility is to define a scheduled task (on Windows) or a *cron* job (for Unix/Linux systems) to execute an appropriate export SQL script every day. The drawback of this is that if you need to change the internal data structure of your database, it is not sufficient to change only the database content, but you also have to adapt the export SQL script definition of your export job simultaneously.

To resolve this problem, you could define a stored procedure. This procedure defines an interface by which the external job can trigger the export. The definition of the actual code necessary to perform the export is encapsulated by the stored procedure that is implemented in the database. Thus, all the necessary changes in case of structural changes are restricted to the database itself.

In this recipe, we will show you an example of how this works by defining a stored procedure that performs the same export task as in the previous recipe.



Stored procedures are only available since MySQL version 5.0. Earlier versions do not support this feature.

## Getting ready

To step through this recipe, you basically need the same prerequisites as in the previous recipe *Exporting data to a custom file format*. In addition, you need a user account with the `CREATE ROUTINE` privilege to define a stored procedure. Finally, you have to make sure that the account used for the external job has the `EXECUTE` privilege. We will use the `sample_install` and `sample_guest` accounts here, assuming that they have the appropriate privileges.

## How to do it...

1. Create a stored procedure by connecting to MySQL (using `mysql` and the `sample_install` account) and entering the following statements:

```
mysql> delimiter //
mysql> CREATE PROCEDURE sample.export_table2() READS SQL DATA
        BEGIN
        SET @filename := 'Filename: C:/target.txt';
        SET @description := 'Description: This is a test
                           from sample.table2 with columns c1, c2, and c3';
        SELECT NOW() INTO @timestamp;
        SELECT COUNT(*) FROM sample.table2 INTO @rowcount;
        SET @rows := CONCAT('Row count: ', @rowcount);
        SET @header := '#Row Nr: Column c1 | Column c2 |
                        c3 $';
        SET @counter := 0;
        SELECT @filename
        UNION SELECT @description
        UNION SELECT @timestamp
        UNION SELECT @rows
        UNION SELECT @header
        UNION SELECT CONCAT('#',
                            @counter := @counter + 1,
                            '|');
        END;
```

## Managing Data

---

```

->          '$')
->      FROM sample.table2
->      INTO OUTFILE 'C:/target.txt';
-> END //
```

**Query OK, 0 rows affected (0.00 sec)**

```

mysql> delimiter ;
mysql>
```

2. Call the stored procedure by executing the following command using the guest account:

```

mysql> CALL sample.export_table2();
Query OK, 0 rows affected (0.01 sec)
```

## How it works...

The steps discussed in the aforementioned section are pretty straightforward. In step 1 we create a stored procedure that consists of the very same commands as presented in the previous recipe *Exporting data to a custom file format*. In step 2, this newly created procedure is called from a different account.

The first and last command of step 1 is the `delimiter //` statement, which is needed to define a stored procedure with more than one statement. It orders the `mysql` client not to interpret the semicolon as the end of a command, but to consider `//` as the end of a statement. With this we can use semicolons inside the procedure body, which would otherwise have caused `mysql` to send the (yet unfinished) command to the server, causing a syntax error. Because of the changed delimiter, we have to close the procedure definition by `END //` (instead of `END ;`). Also, as we do not want to keep the changed delimiter longer than necessary, we revert this to the original form (semicolon) in the next line after the closing `delimiter ;` statement. The `READS SQL DATA` portion of the `CREATE PROCEDURE` command is only instructive and has no implications for the way the procedure is executed.

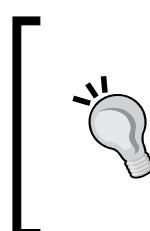
One advantage of this approach is that Step 2 can be repeated often without having to write rather complex export statements again and again.

Another benefit is that the changes made to the database structure can be hidden from the user of the procedure. If, for example, column `c1` needs to be renamed to `foo`, you will need to adapt the stored procedure accordingly by replacing the `CONCAT_WS` portion of the command with `CONCAT_WS(' | ', foo, c2, c3)`.

Finally, this solution makes it possible to provide the ability to export data to a file without having to grant the `FILE` privilege to the respective account. Granting the `FILE` privilege to the account that calls the stored procedure is sufficient.

---

stored procedure, the account that calls the routine does not need to have the FILE privilege attached, as the statements are executed in the security context of the *creator* of the stored procedure (in our example: sample\_install).



When defining a stored procedure in MySQL, the procedure is by default executed with the privileges of the definer of the procedure. If you want to apply the privileges of the invoker instead, you have to add a SQL SECURITY INVOKER clause to the CREATE PROCEDURE statement. The default value is SQL SECURITY DEFINER.

With this arrangement, sample\_guest is able to export data to files (using predefined procedures), but is not allowed to execute SELECT ... INTO OUTFILE statements.

## There's more...

The code example from the recipe exports data to a predefined file. Anyone who calls the routine has to make sure that the target file C:/target.txt does not exist. In some situations, this restriction is not acceptable. Using parameters, it is possible to pass the name of the target file as a parameter. Because the file name in the INTO OUTFILE clause has to be a literal value, we have to use the approach mentioned previously of constructing the command in a string value, preparing a statement from this, and executing it:

```
mysql> delimiter //
mysql> CREATE PROCEDURE sample.export_table2_FileAsParam(IN file
CHAR(255)) READS SQL DATA
      -> BEGIN
      ->   SET @filename := CONCAT('Filename: ', @file);
      ->   SET @description := 'Description: This is a test export
      ->                         of sample.table2 with columns c1, c2, and c3';
      ->   SELECT NOW() INTO @timestamp;
      ->   SELECT COUNT(*) FROM sample.table2 INTO @rowcount;
      ->   SET @rows := CONCAT('Row count: ', @rowcount);
      ->   SET @header := '#Row Nr: Column c1 | Column c2 | Column c3';
      ->   SET @command = CONCAT("SELECT @filename
      ->                         UNION SELECT @description
      ->                         UNION SELECT @timestamp
      ->                         UNION SELECT @rows
      ->                         UNION SELECT @header
      ->                         UNION SELECT CONCAT('#', lpad(@rowcount, 10, 0), ' rows found')");
      ->   PREPARE stmt FROM @command;
      ->   EXECUTE stmt;
      ->   DEALLOCATE PREPARE stmt;
```

## Managing Data

```

">      ': ',
">      CONCAT_WS(' | ', c1, c2, c3),
">      '$')
">      FROM sample.table2
">      INTO OUTFILE '", file, "' ;");
->      SET @counter := 0;
->      PREPARE statement FROM @command;
->      EXECUTE statement;
->      END //"
Query OK, 0 rows affected (0.00 sec)

```

```
mysql> delimiter ;
```

This routine allows passing the file name when calling it:

```
mysql> call sample.export_table2_FileAsParam("C:/data.out");
Query OK, 0 rows affected (0.00 sec)
```

## Importing data from a simple CSV file

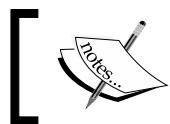
A common task when working with databases is to import data from different sources. Unfortunately, this data will typically not be provided as a convenient set of well-formed statements that you can simply run against your database. Therefore, here you will deal with data in a different format.

As a common denominator, **character-separated values (CSV)** are still a prevalent way of exchanging data. In this chapter, we will show you how to import data stored in CSV format. As a typical example, we will use the file format *Microsoft Excel* produces when storing data in the \*.CSV file type.

This recipe is the counterpart of the *Exporting data to a simple CSV file* recipe in the previous chapter.

### Getting ready

To step through this recipe, we will definitely need a file to import (here: C:/source.csv) and a table to import the data into (here: sample.table1). The source file and target table have to have a matching format concerning the number of columns and the type of data stored in them. Furthermore, an account with INSERT and FILE privileges is required. We will assume an account sample\_install in this recipe.



The source file has to be located on the machine that runs your MySQL server, not on the client side!

## How to do it...

1. Connect to the database using the sample\_install account.
2. Issue the following SQL command:

```
mysql> LOAD DATA INFILE 'C:/source.csv' INTO TABLE sample
FIELDS ENCLOSED BY '"' TERMINATED BY ';' ESCAPED BY '\"'
TERMINATED BY '\r\n';
Query OK, 20 rows affected (0.06 sec)

Records: 20 Deleted: 0 Skipped: 0 Warnings: 0
```

## How it works...

The LOAD DATA INFILE command works analogous to the SELECT ... INTO OUTFILE command discussed in the previous recipes, but as a means for importing data rather than exporting. The format options available for both commands are identical, so you can import data exported by a SELECT ... INTO OUTFILE statement using a LOAD DATA INFILE command with the same format options.

As most files consist of lines terminated by a sequence of a carriage return and a line feed character, we use the LINES TERMINATED BY '\r\n' option. The choice of the line separator character—as a separator for different fields of every line (TERMINATED BY ';' )—is due to the fact that *Excel* uses this format. If you happen to receive CSV files that, however, use a comma instead, you have to adjust this accordingly.

The term FIELDS ENCLOSED BY '\"' tells the import to look for double quotes at the start of every field imported. If there is one, the field is considered to end at the next double quote. To be able to have double quotes inside a field value, we define an escape character (ESCAPED BY '\"'). With this constellation, a sequence of two double quotes is not seen as the end of the field, but as a double-quote character as part of the value.

## There's more...

The data is read from the file using the default character set of the database. If the source file uses a different character encoding, you can specify this by adding a CHARACTER SET clause to the table definition (LOAD DATA INFILE ... INTO TABLE sample.table1 CHARACTER SET utf8 ;). Please note that the character sets ucs2, utf16, and utf32 are not supported (as of MySQL version 5.1.35).

*Managing Data***See also**

- ▶ *Exporting data to a simple CSV file*

## Importing data from custom file formats

In the previous recipe *Importing data from a simple CSV file*, we discussed a way of importing data from a nicely formatted file. Unfortunately, you sometimes have to deal with files that are not so convenient data sources. In this recipe, we will present some more advanced topics of importing data from files with a less strict structure.

Obviously, it is not possible to present a universal recipe for every file format imaginable. Instead, we will use an example that covers some of the common problems one has to tackle when importing data from custom files. For this, we will refer to the same hypothetical file as in the previous recipe *Export data to a custom file format*, which defines four initial lines (containing names, a file, a time stamp, a description, and the number of rows), a header line with the number of columns, and subsequently the rows with the actual data to import. Each data row starts with a hash character (#), the line number, a colon, and a space. The data values that follow are separated by a pipe (|) character and the row closes with a dollar sign (\$).

### Getting ready

Again, the account used in the recipe needs the FILE privilege (besides the INSERT permission for the table the data should be imported into). With a SQL client, a file in the appropriate format, and a table as the import target, we are ready to go. As in the previous recipes, we use sample\_install as the account name, C:/source.txt as the source file, and sample.table2 (consisting of three columns c1, c2, and c3) as the target table. We assume the source file to have the following content:

```

Filename: C:/source.txt
Description: This is a file for test import to sample.table2
c1, c2, and c3
2009-06-14 13:25:05
Row count: 3
#Row Nr: Column c1 | Column c2 | Column c3 $
#1: 209 | Some text in my test data | Some more text $
#2: 308 | Next test text for testing | Text to test $
#3: 406 | "A water | pipe" | Really? $

```

### How to do it...

- 1 Connect your favorite client (for example, the mysql command line client)

- 
2. Execute the following SQL command:

```
mysql> LOAD DATA INFILE "C:/source.txt"
      -> INTO TABLE sample.table2
      -> FIELDS TERMINATED BY ' | '
      -> OPTIONALLY ENCLOSED BY ""
      -> LINES STARTING BY ':'
      -> TERMINATED BY '\r\n'
      -> IGNORE 5 LINES
      -> SET c3=TRIM(TRAILING ' $' FROM c3);

Query OK, 3 rows affected (0.05 sec)

Records: 3 Deleted: 0 Skipped: 0 Warnings: 0
```

## How it works...

Let us dissect the above statement by having a look at the source file: first of all, we want to import data from the file C:\source.txt into the table sample.table2, which is represented by the first two lines (LOAD DATA INFILE ... INTO TABLE ...).

At the top of the file, we have five lines (the initial four lines with information about the header) that should not be imported into the target table. To achieve this, the lines option is added.

The remaining lines are prefixed with a hash character, the row number, and a colon. The colon of every line has to be ignored, which is what the LINES STARTING BY ': ' option tells MySQL to ignore the first colon of the line and any character before it. By doing so, the row number prefix is skipped.

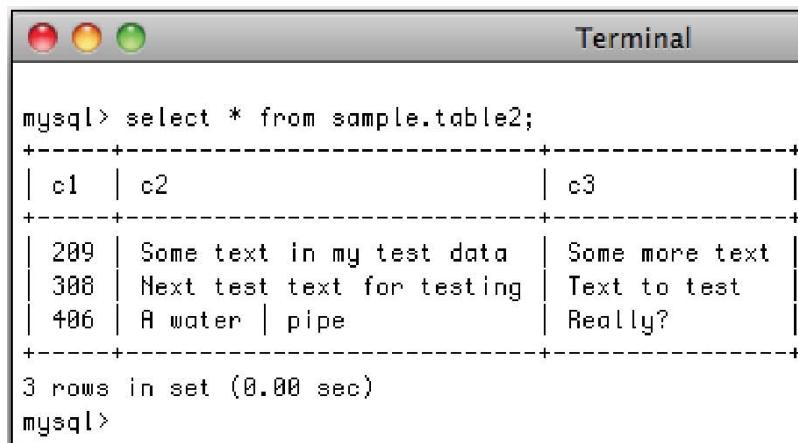
After the prefix, the lines contain the actual values, separated by pipe characters. The FIELDS TERMINATED BY ' | ' option tells MySQL how to identify a field separator. With the additional setting OPTIONALLY ENCLOSED BY "", the value itself might contain a separator sequence—if the whole value is enclosed by double quotes (this is the case of the last row of the sample file).

At this point, there is only one problem left: the lines end with a dollar sign, which is part of the last value. An intuitive approach would be to include this character in the line termination sequence, which means to use \$\r\n as a line ending (instead of \r\n). Unfortunately, this definition of a line end does not work as expected for our example. As a result, the first six lines would be considered as one single line by the import process, because the first five lines are not terminated by the line ending sequence. Only the sixth line actually ends with a character sequence of \$\r\n. To be able to exclude the header lines from the import, we have to rely on the "traditional" line endings.

## Managing Data

Hence, the options for defining the field separators, and the beginning and termina  
a line do not allow us to get rid of the closing dollar sign. Thus it is considered part  
last value, which is assigned to column c3. To finally get rid of this postfix, the SET  
of the LOAD DATA INFILE command comes in handy, which allows to clearly de  
the values that are assigned to the columns in the target table. The closing option  
c3=TRIM(TRAILING ' '\$' FROM c3); defines a way to strip the unwanted pos  
the last field.

If we put it all together, the import works as intended:



```
mysql> select * from sample.table2;
+-----+-----+-----+
| c1   | c2          | c3      |
+-----+-----+-----+
| 209  | Some text in my test data | Some more text | |
| 308  | Next test text for testing | Text to test   |
| 406  | A water | pipe           | Really?       |
+-----+-----+-----+
3 rows in set (0.00 sec)
mysql>
```

### There's more...

As with exporting data, it is recommended to consider using an external program  
language to import more complex data structures into MySQL. While it is possible to  
rather sophisticated file formats using MySQL commands as well, it is often far more  
to have a full-blown programming language at hand to solve the task of parsing input.  
This is most notably the case when it comes to XML files.



For importing data from XML files, consider using an XSLT processor  
to produce corresponding SQL commands!

### See also

- ▶ *Importing data from a simple CSV file*
- ▶ *Exporting data to a custom file format*

# Inserting new data and updating data if it already exists

Manipulating data in a database is part of everyday work and the basic SQL means INSERT, UPDATE, and DELETE make this a pretty straightforward, almost trivial task. Is this always true?

When considering data manipulation, most of the time we think of a situation where we know the content of the database. With this information, it is usually pretty easy to find a way of changing the data the way you intend to. But what if you have to change data in circumstances where you do not know the actual database content beforehand?

You might answer: "Well, then look at your data before changing it!" Unfortunately, this is not always have this option. Think of distributed installations of any software that interact with a database. If you have to design an update option for this software (and the respective databases), you might easily come to a situation where you simply do not know about the actual database content.

One example of a problem arising in these cases is the question of whether to insert or update data: "Does the data in question already (partially) exist?" Let us assume a table config that stores configuration settings. It holds key-value pairs, with name being the name (and thus the key) of the setting and value its value. This table exists in all database installations, one for every branch office of your company. Your task is to create an update package to set a uniform upper limit of 25% for the price discount that is applied by your sales software. If no such limit has been defined yet, there is no respective entry in the config table, and you have to **insert** a new record. If the limit, however, has been defined (for example by the local manager), the entry does already exist, in which case you have to **update** it to hold the new value.

While the update of a potentially existing entry does not pose a problem, an **INSERT** statement that violates uniqueness constraints will simply cause an error. This is, typically not acceptable in an automated update procedure. The following recipe will show how to solve this problem with only one SQL command.

## Getting ready

Besides a running MySQL server, a SQL client, and an account with appropriate user rights (**INSERT**, **UPDATE**), we need a table to update. In the earlier example, we assumed a named sample.config with two character columns name and value. The name column is defined as the primary key:

```
CREATE TABLE sample.config (
    name VARCHAR(64) PRIMARY KEY,
```

*Managing Data***How to do it...**

1. Connect to your database using your SQL client
2. Execute the following command:

```
mysql> INSERT INTO sample.config VALUES ("maxPriceDiscount", "25%") ON DUPLICATE KEY UPDATE value='25%';
Query OK, 1 row affected (0.05 sec)
```

**How it works...**

This command is easily explained because it simply does what it says: it inserts a new row into the table using the given values, as long as this does not cause a duplicate entry in either the primary key or another unique index. If a duplicate record exists, the existing row is updated according to the clauses defined after ON DUPLICATE KEY UPDATE.

While it is sometimes tedious to enter some of the data and columns two times (once for the INSERT and a second time for the UPDATE), this statement allows for a lot of flexibility when it comes to the manipulation of potentially existing data.

Please note that when executing the above statement, the result differs slightly with respect to the number of affected rows, depending on the actual data present in the database. If the record does not exist yet, it is inserted, which results in one affected row. But if the record already exists and is updated rather than inserted, it reports **two** affected rows instead, even if only one row gets updated.

**There's more...**

The INSERT INTO ... ON DUPLICATE UPDATE construct does not work when there is no UNIQUE or PRIMARY KEY defined on the target table. If you have to provide the semantics without having appropriate key definitions in place, it is recommended to use the techniques discussed in the next recipe.

**See also**

- ▶ *Inserting data based on existing database content*

# Inserting data based on existing database content

In the previous recipe *Inserting new data and updating data if it already exists*, we saw a method to either insert or update records depending on whether the records already exist in the database. A similar problem arises when you need to insert data to your database, but the data to insert depends on the data in your database.

As an example, consider a situation in which you need to insert a record with a certain message into a table `logMsgs`, but the message itself should be different depending on the current system language that is stored in a configuration table (`config`).

It is fairly easy to achieve a similar behavior for an `UPDATE` statement because this statement has a `WHERE` clause that can be used to only perform an update if a certain precondition is met:

```
UPDATE logMsgs SET message=
    CONCAT('Last update: ', NOW()) WHERE EXISTS
    (SELECT value FROM config WHERE
        name='lang' AND value = 'en');

UPDATE logMsgs SET message=
    CONCAT('Letztes Update: ', NOW()) WHERE EXISTS
    (SELECT value FROM config WHERE
        name='lang' AND value = 'de');

UPDATE logMsgs SET message=
    CONCAT('Actualisation derniere: ', NOW()) WHERE EXISTS
    (SELECT value FROM config WHERE
        name='lang' AND value = 'fr');
```

Unfortunately, this approach is not applicable to `INSERT` commands, as these do not have a `WHERE` clause. Despite this missing option, the following recipe describes a method by which `INSERT` statements execute only if an appropriate precondition in the database is met.

## Getting ready

As before, we assume a database, a SQL client (`mysql`), and a MySQL user with sufficient privileges (`INSERT` and `SELECT` in this case). Additionally, we need a table to insert data into (here: `logMsgs`) and a configuration table `config` (please refer to the previous recipe for details).

## Managing Data

---

### How to do it...

1. Connect to your database using your SQL client.
2. Execute the following SQL commands:

```
mysql> INSERT INTO sample.logMsgs(message)
      ->   SELECT CONCAT('Last update: ', NOW())
      ->   FROM sample.config WHERE name='lang' AND value='en'
Query OK, 0 rows affected (0.00 sec)

Records: 0  Duplicates: 0  Warnings: 0

mysql> INSERT INTO sample.logMsgs(message)
      ->   SELECT CONCAT('Letztes Update: ', NOW())
      ->   FROM sample.config WHERE name='lang' AND value='de'
Query OK, 1 row affected (0.05 sec)

Records: 1  Duplicates: 0  Warnings: 0

mysql> INSERT INTO sample.logMsgs(message)
      ->   SELECT CONCAT('Dernière actualisation: ', NOW())
      ->   FROM sample.config WHERE name='lang' AND value='fr'
Query OK, 0 rows affected (0.00 sec)

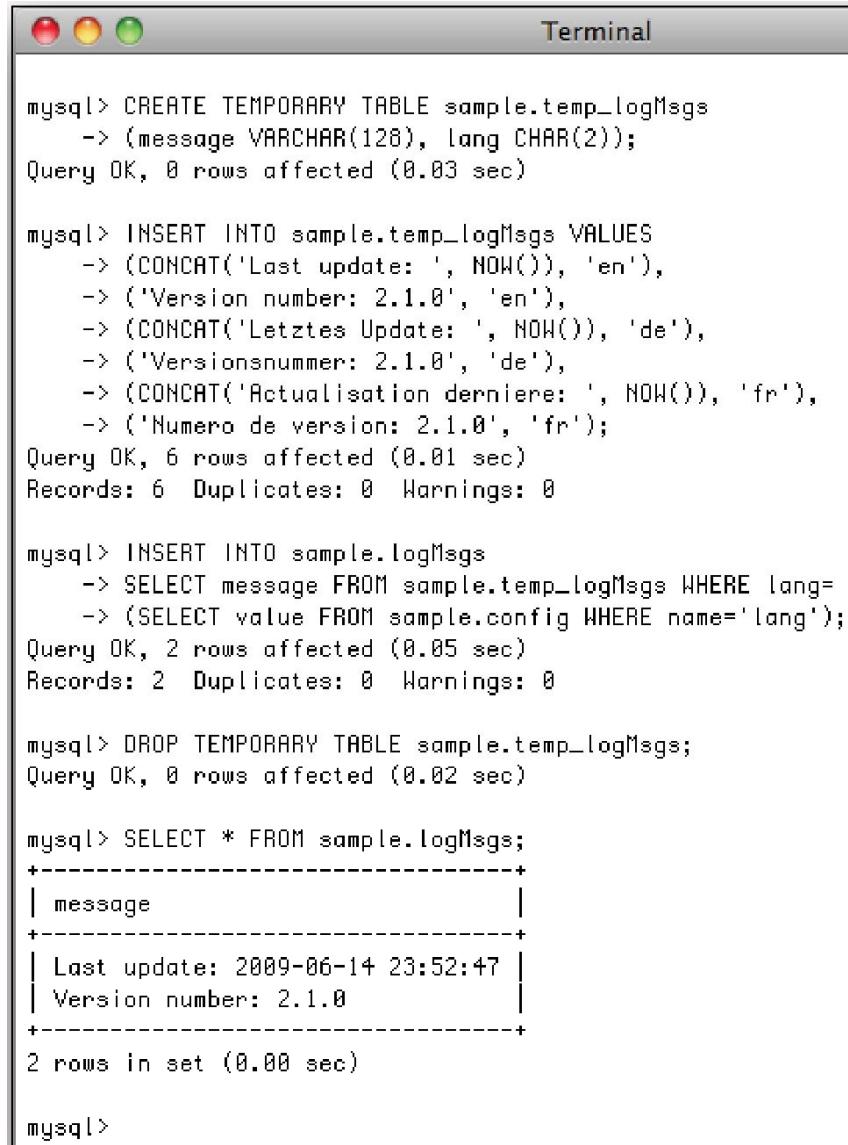
Records: 0  Duplicates: 0  Warnings: 0
```

### How it works...

Our goal is to have an `INSERT` statement take into account the present language of the database. The trick to do so is to use a `SELECT` statement as input for the `INSERT`. A `SELECT` command provides a `WHERE` clause, so you can use a condition that only applies for the respective language. One restriction of this solution is that you can only insert one record at a time, so the size of scripts might grow considerably if you have to insert many data and/or have to cover many alternatives.

### There's more...

If you have more than just a few values to insert, it is more convenient to have them all in one place rather than distributed over several individual `INSERT` statements. In this case, it makes sense to consolidate the data by putting it inside a **temporary table**; the final `INSERT` statement uses this temporary table to select the appropriate data rows for insertion.



```

Terminal

mysql> CREATE TEMPORARY TABLE sample.temp_logMsgs
-> (message VARCHAR(128), lang CHAR(2));
Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO sample.temp_logMsgs VALUES
-> (CONCAT('Last update: ', NOW()), 'en'),
-> ('Version number: 2.1.0', 'en'),
-> (CONCAT('Letztes Update: ', NOW()), 'de'),
-> ('Versionsnummer: 2.1.0', 'de'),
-> (CONCAT('Actualisation dernière: ', NOW()), 'fr'),
-> ('Numero de version: 2.1.0', 'fr');
Query OK, 6 rows affected (0.01 sec)
Records: 6  Duplicates: 0  Warnings: 0

mysql> INSERT INTO sample.logMsgs
-> SELECT message FROM sample.temp_logMsgs WHERE lang=
-> (SELECT value FROM sample.config WHERE name='lang');
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> DROP TEMPORARY TABLE sample.temp_logMsgs;
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM sample.logMsgs;
+-----+
| message          |
+-----+
| Last update: 2009-06-14 23:52:47 |
| Version number: 2.1.0           |
+-----+
2 rows in set (0.00 sec)

mysql>
```

After creating the temporary table with the first statement, we insert data into the the following `INSERT` statement. The next statement inserts the appropriate data target table `sample.logMsgs` by selecting the appropriate data from the temporary table that matches the language entry from the `config` table. The temporary table is then dropped again. The final `SELECT` statement is solely for checking the results of the operation.

## See also

- ▶ *Inserting new data and updating data if it already exists*

*Managing Data*

## Deleting all data from large tables

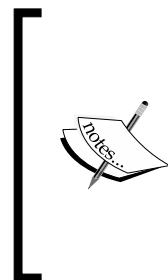
Almost everyone who works with databases experiences the constant growth of the data stored in their database and it is typically well beyond the initial estimates. Because you often end up with rather large data sets. Another common observation is that in databases, there are some tables that have a special tendency to grow especially rapidly.

If a table's size reaches a virtual threshold (which is hard to define, as it depends largely on the access patterns and the data structures), it gets harder and harder to maintain. Performance degradation might occur. From a certain point on, it is even difficult to add new data in the table again, as the sheer number of records makes deletion a pretty slow task. This particularly holds true for storage engines with **Multi-Version Concurrency Control (MVCC)**: if you order the database to delete data from the table, it must not be deleted away because you might still roll back the deletion. So even while the deletion was performed, a concurrent query on the table still has to be able to see all the records (depending on the transaction isolation level). To achieve this, the storage engine will only mark the rows as deleted, but the actual deletion takes place after the operation is committed and all other transactions that access this table are closed as well.

If you have to deal with large data sets, the most difficult task is to operate on the data while other processes concurrently work on the data. In these circumstances, it is important to keep the duration of your maintenance operations as low as possible in order to minimize the impact on the running system. As the deletion of data from a large table (typically containing several millions of rows) might take quite some time, the following recipe shows how to minimize the duration of this operation in order to reduce side effects (like locking or performance degradation).

### Getting ready

Besides a user account with appropriate privileges (DELETE), you need a sufficient amount of data in a table to delete data from.



For this recipe, we will use the `employees` database, which is an example database available from MySQL: <http://dev.mysql.com/doc/employee/en/employee.html>.

This database provides some tables with sensible data and some pretty large tables, the largest having more than 2.8 million records.

We assume that the `Employees` database was installed with an InnoDB storage engine enabled. To delete all rows of the largest table `employees.salaries` in a quick and efficient manner, we can use the following command:

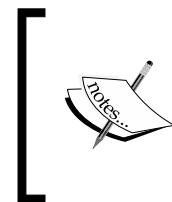
## How to do it...

1. Connect to your database.
2. Enter the following SQL command:

```
mysql> TRUNCATE TABLE employees.salaries;
Query OK, 0 rows affected (0.16 sec)
```

## How it works...

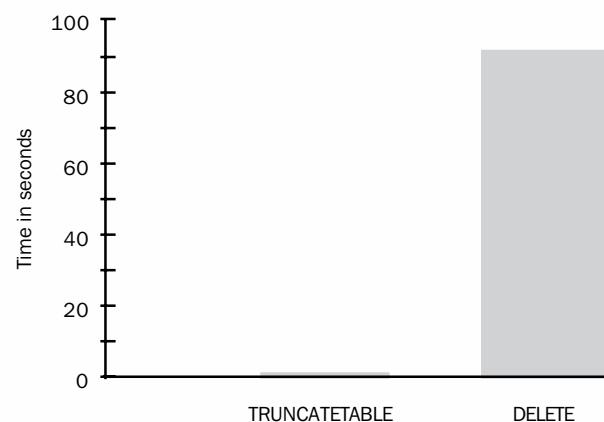
The `TRUNCATE TABLE` command is a rather fast way of deleting all data from a table. It basically drops the table temporarily and recreates the table with the same structure before. This operation has basically a constant time characteristic—the amount of data inside the table does not have any effect in the time needed for the `TRUNCATE` command.



Before MySQL 5.0.3, the `TRUNCATE TABLE` statement for InnoDB tables was always equivalent to a `DELETE` statement, regardless of whether Foreign key constraints exist or not. To take advantage of the speed improvement you have to use MySQL 5.0.3 or later.

In comparison to a classical `DELETE FROM employees.salaries;` operation, the reduction in time needed is striking:

Operation	Time needed
TRUNCATE TABLE	0.16 sec
DELETE	1 min 31.55 sec



## Managing Data

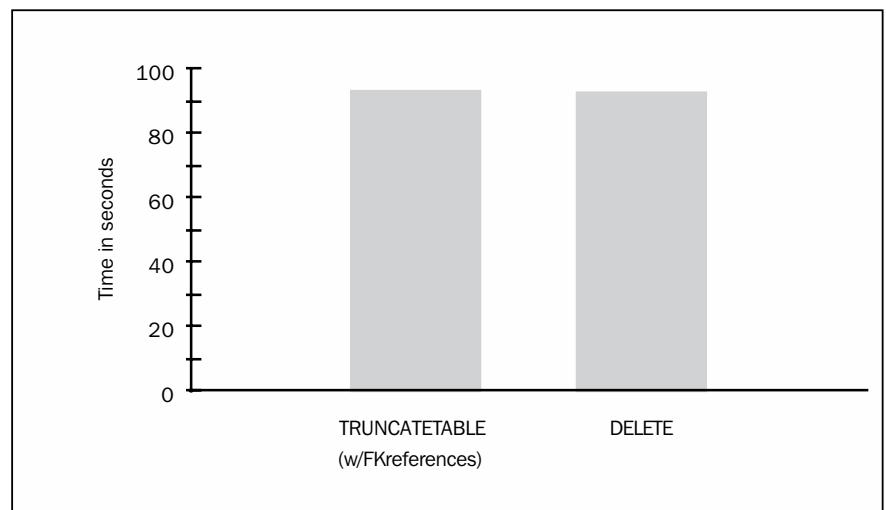
The TRUNCATE TABLE command takes only a fraction of the time needed for the DELETE statement. However, there are some caveats.

First of all, the TRUNCATE command will only have the speed advantage on InnoDB if the table is not referenced by any Foreign key constraints. But if the table is referenced by Foreign keys, the TRUNCATE TABLE command is equivalent to executing a DELETE statement with no WHERE clause, also eliminating all speed differences:

```
mysql> CREATE TABLE employees.salaries_referencer (
    -> emp_no      INT,
    -> from_date   DATE,
    -> CONSTRAINT salaries_fk
    -> FOREIGN KEY (emp_no, from_date)
    -> REFERENCES salaries (emp_no, from_date)
    -> ON DELETE RESTRICT);
Query OK, 0 rows affected (0.08 sec)

mysql> TRUNCATE TABLE employees.salaries;
Query OK, 0 rows affected (1 min 33.44 sec)
```

Operation	Time needed
TRUNCATE TABLE (with foreign key ref)	1 min 33.44 sec
DELETE	1 min 32.96 sec



Furthermore, the TRUNCATE statement requires the DROP privilege (before MySQL 5.7.7 it only requires the DELETE permission), which forbids use of this command for security reasons.

---

And finally, TRUNCATE is not a transaction-safe command. If you execute a TRUNCATE statement, you will not be able to perform a rollback on this operation any more, as an open operation from the current transaction gets automatically committed as well. This is a characteristic that disqualifies this statement for situations in which the possibility of performing a rollback is mandatory; you will have to stick with the (much slower) DELETE command in these cases.

## There's more...

As we have seen, TRUNCATE TABLE only has performance advantages if there is no foreign key reference to the table that is to be deleted. Here we will discuss how to use the command to achieve performance improvements even in case of existing references.

### **Temporarily disabling Foreign key constraints**

To make use of the increased speed of TRUNCATE TABLE although the target table is referenced via Foreign keys, you could temporarily remove the Foreign key constraint with the ALTER TABLE command, execute the TRUNCATE TABLE command, and reestablish the references afterwards. Using the example of a table salaries\_referencer that references salaries, you could use the following sequence:

```
mysql> ALTER TABLE employees.salaries_referencer
      -> DROP FOREIGN KEY salaries_fk;
Query OK, 0 rows affected (0.19 sec)

Records: 0  Duplicates: 0  Warnings: 0

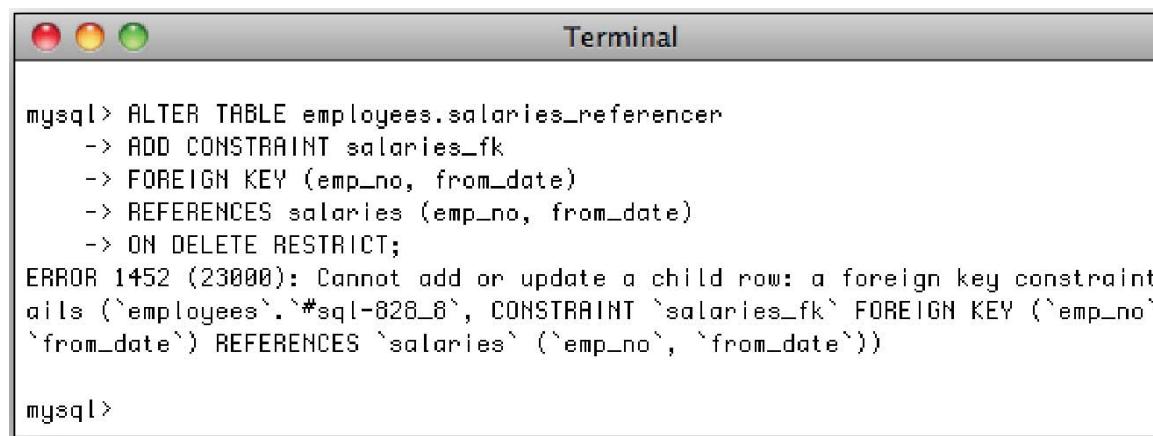
mysql> TRUNCATE TABLE employees.salaries;
Query OK, 0 rows affected (0.44 sec)

mysql> ALTER TABLE employees.salaries_referencer
      -> ADD CONSTRAINT salaries_fk
      -> FOREIGN KEY (emp_no, from_date)
      -> REFERENCES salaries (emp_no, from_date)
      -> ON DELETE RESTRICT;
Query OK, 0 rows affected (0.14 sec)

Records: 0  Duplicates: 0  Warnings: 0
```

## Managing Data

With this sequence, you temporarily disable the Foreign key constraints to have TRUNCATE TABLE use the faster deletion method. Beware, however, that this method might also lead to problems when the deletion of the table produces "loose ends". If the referential table salaries.referencer holds records that referenced the now empty target table salaries, the creation of the Foreign key constraints will fail:



```
mysql> ALTER TABLE employees.salaries_referencer
    -> ADD CONSTRAINT salaries_fk
    -> FOREIGN KEY (emp_no, from_date)
    -> REFERENCES salaries (emp_no, from_date)
    -> ON DELETE RESTRICT;
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails ('employees`.`#sql-828_8`, CONSTRAINT `salaries_fk` FOREIGN KEY (`emp_no` ,`from_date`) REFERENCES `salaries` (`emp_no` ,`from_date`))

mysql>
```

Also, keep in mind that this situation might also occur because of concurrent processes which are able (for the duration of the disabled constraints) to insert data into the table that violate the intended referential integrity.

As an alternative, you might be tempted to temporarily disable the Foreign key checks by setting `foreign_key_checks` to zero. While this works regarding the TRUNCATE TABLE performance, it is strongly discouraged to use this option because the Foreign key constraint is not revalidated when the Foreign key checks are enabled again. So you risk inconsistency with respect to the referential integrity.

## Deleting all but a fragment of a large table's data

In the previous recipe *Deleting all data from large tables*, we discussed a method for removing all data from large tables while avoiding performance hits. But experience has shown that you often must not delete all data, but have to retain some records and delete others. The `TRUNCATE TABLE` command does not allow any additional clauses to define which records to delete and which not; it always deletes all entries.

The intuitive solution to this would be to use a normal `DELETE` command with a WHERE clause that only matches the records to delete. For large tables, this might prove quite an expensive operation (in terms of duration). In this recipe, we will show you how to quickly remove the data from large tables while preserving some of the records.

## Getting ready

We again need a MySQL server up and running and a SQL client (like mysql). For we also need a user account with SELECT, INSERT, DELETE, DROP, and CREATE for the target database (we will use the sample\_install user throughout this se will furthermore use the Employees sample database in an InnoDB context. This d was introduced in the previous recipe and is available for free on the MySQL webs from. In our example, we will delete all records having a from\_date before the th '2002-01-01 00:00:00.0'.

## How to do it...

1. Connect to the database using a SQL client and the sample\_install ac
2. Execute the following commands:

```
mysql> use employees;
Database changed
mysql> CREATE TABLE salaries_part
    -> SELECT * FROM salaries
    -> WHERE from_date >= "2002-01-01 00:00:00.0";
Query OK, 140930 rows affected (11.47 sec)
Records: 140930  Duplicates: 0  Warnings: 0

mysql> TRUNCATE TABLE salaries;
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO salaries SELECT * from salaries_part;
Query OK, 140930 rows affected (4.63 sec)
Records: 140930  Duplicates: 0  Warnings: 0

mysql> DROP TABLE salaries_part;
Query OK, 0 rows affected (0.06 sec)
```

*Managing Data*

## How it works...

For speeding up the deletion of most of the data from a large table, we utilize the advantage of the `TRUNCATE TABLE` statement over a `DELETE` command. In details, the steps are as follows:

The initial `USE` statement is for convenience only, so we do not have to give the `employee` prefix for every table.

With the next statement (`CREATE TABLE ... SELECT * FROM ...`), we simply copy all records that should **not** be removed to a newly created table `salaries_part`.



Be careful to avoid errors when inverting conditions: to delete all entries before time X, you have to copy all records values later **or equal to** X. If you copy only records later than X, all records exactly at time X would be deleted as well.

This table temporarily holds the data while we delete all data from the large `salaries` table using `TRUNCATE` in the next step. Afterwards, we simply copy the partial data from the `salaries_part` table back into the original (now emptied) `salaries` table. With the final step, we scrap the `salaries_part` table again, as it is not needed any more.



You could also create `salaries_part` as a `TEMPORARY` table. In this case, you could also skip the final `DROP` statement. This method is discouraged because it might lead to data loss in case of an error. The reason is that if an error occurs right after all data was deleted from the original `salaries` table, but before the data from the temporary table is restored. If the connection is closed, the data from the temporary table is lost. A non-temporary table does not entail this risk.

---

## There's more...

We will not try to conceal that this approach has some caveats as well. First of all, who performs this operation needs some additional privileges like CREATE and DROP, which renders it unusable for many users with only basic permissions.

You should also keep in mind that the use of either CREATE TABLE or TRUNCATE triggers an automatic commit of any transaction currently active, which basically means that this approach does not provide any transaction safety.

If concurrent database access is possible during the process of deletion, an additional problem comes up. In the period of time between the TRUNCATE and completion of the INSERT INTO ... SELECT FROM ... statements, the salaries table is empty for any other transaction. You have to make sure that this will not cause any problem. You can use the DELETE approach otherwise, as this will not produce intermediate states in which the database table is completely empty.

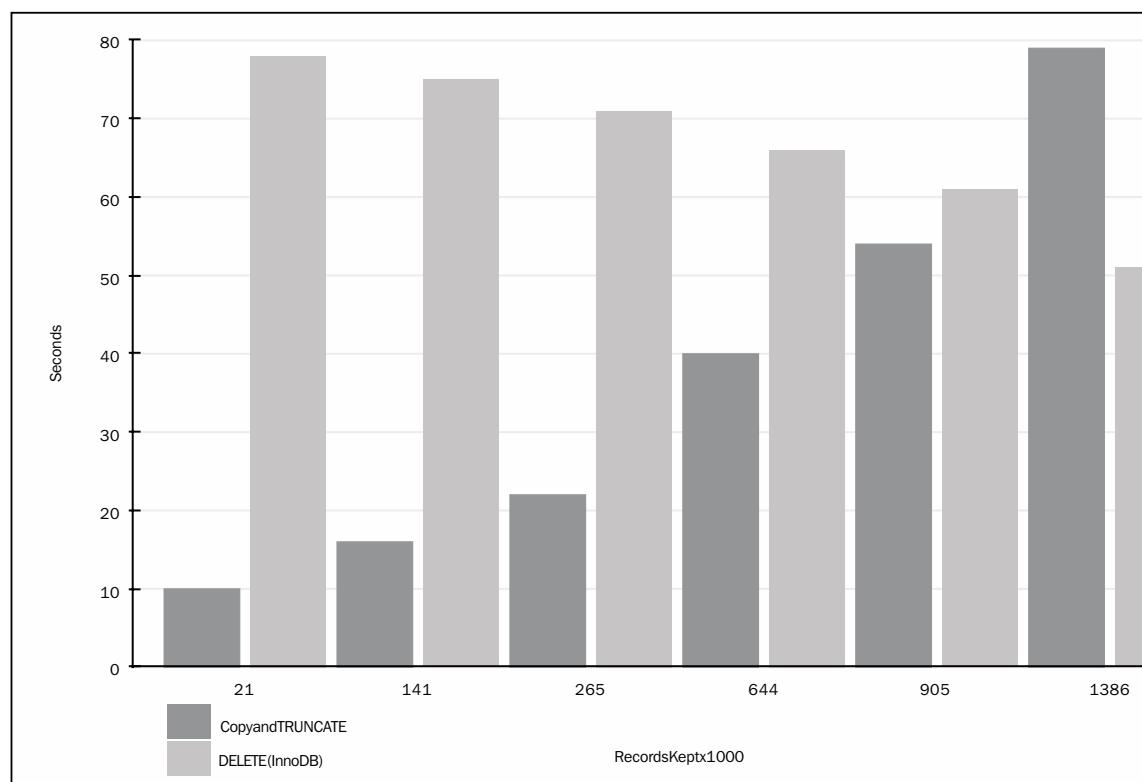
And finally, the performance benefit of this approach for InnoDB greatly depends on the speed of the TRUNCATE TABLE statement. However, if there are tables that reference the target table with a Foreign key, the TRUNCATE will be equivalent to a DELETE statement, destroying all performance improvements. A solution to this problem is to temporarily disable the Foreign key references. Please refer to the *Temporarily disabling Foreign key constraints* section of the previous recipe for a description of how to achieve this.

## Performance considerations

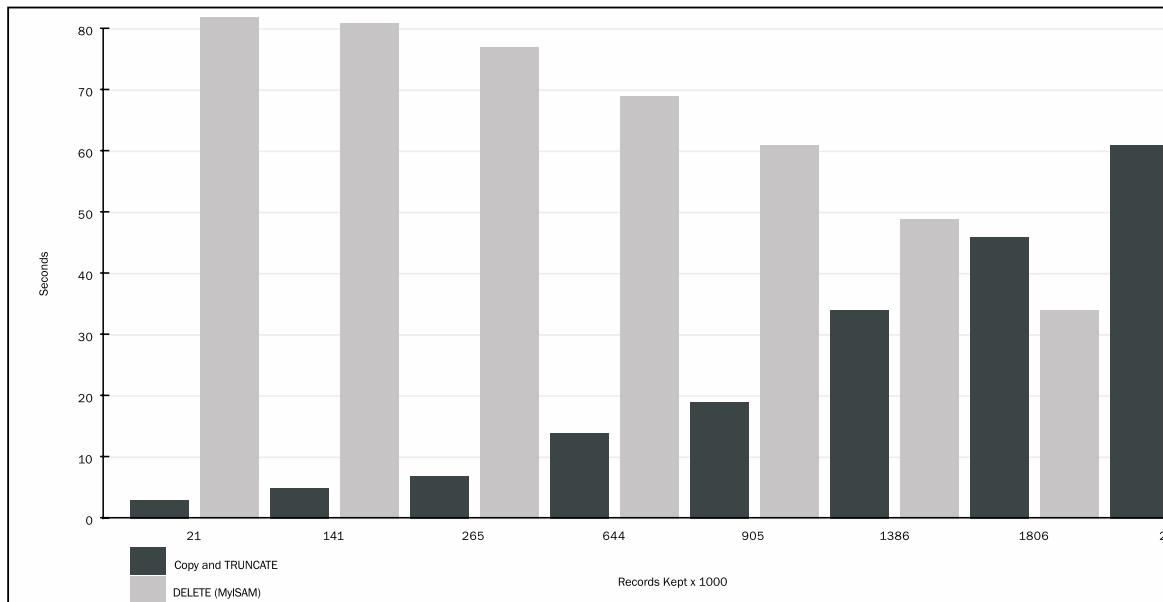
A comparison between the method presented in this recipe and the use of an ordinary DELETE statement shows that the advantages depend on the amount of data that is deleted. The more data is copied to the provisional table, the longer the operation. A DELETE statement, however, behaves conversely: it gets faster if more data is deleted. Once a certain threshold is reached, the normal deletion will even be faster than the Copy-and-Truncate approach. As a rule of thumb for InnoDB tables, if you delete two thirds of the data, you can use the Copy-and-Truncate method; otherwise, a simple DELETE might prove faster. This differs slightly for other storage engines: for MyISAM, the Copy-and-Truncate method typically works faster if more than half of the data is deleted. So when considering the deletion of data from large tables, you should take a second to think about which approach fits better for your particular circumstances.

## Managing Data

The following two diagrams compare the times needed to partially delete data either with a simple DELETE statement or the Copy-and-Truncate solution for different numbers of rows that are left after the operation. The table originally contains about 2.8 million rows. This figure shows the comparison for the InnoDB storage engine:



For MyISAM, the Copy-and-Truncate mechanism is faster even for larger numbers of remaining rows:



# Deleting all data incrementally from large tables

In the previous recipe *Deleting all but a fragment of a large table's data*, we discussed a method to quickly remove all but a small remainder of records from a large table. The downside of the approach presented there is the fact that during the process of deletion, the table temporarily appears completely empty to an observer. Unfortunately, this is not acceptable, especially if your database is used in an environment with many processes concurrently accessing the database, particularly the large table discussed.

On the other hand, the alternative of simply using a `DELETE` statement is sometimes acceptable either. A `DELETE` statement temporarily creates locks on the entries that are deleted. As a result, for the duration of the deletion, a major part of the table gets locked, thus preventing concurrent access to the table by other processes. This typically leads to timeout situations and other errors, as in the following example:

## SQL client 1

```
DELETE FROM employees.salaries
WHERE emp_no < 485000;
```

```
=> Query OK, 2702167 rows affected
(1 min 11.85 sec)
```

## SQL client 2

```
INSERT INTO salaries VALUES
0, "2010-01-01", "2099-12-31"
=> ERROR 1205 (HY000): Lock
timeout exceeded; try rest
transaction
```

The following recipe shows an approach to deleting data from large tables without locking access to the table's data for too long, so the deletion can happily be performed despite concurrent tasks simultaneously accessing the very same table.

## Getting ready

As the recipe uses a stored procedure, we again need a user account with the `CREATE ROUTINE` privilege as well as the `DELETE` permission for the target database. Through the following steps, we will again assume `sample_install` as the user. The example for deletion is `salaries` from the `employees` sample database (see previous recipe once more). In our example, we will delete all entries from the table with an `employee_no` below 485000.

*Managing Data***How to do it...**

1. Connect to the database using the sample\_install account.
2. Enter the following SQL statements:

```
mysql> delimiter //
mysql> CREATE PROCEDURE employees.delete_incrementally()
      -> MODIFIES SQL DATA
      -> BEGIN
      ->     REPEAT
      ->         DELETE FROM employees.salaries
      ->         WHERE emp_no < 485000
      ->         LIMIT 20000;
      ->         UNTIL ROW_COUNT() = 0 END REPEAT;
      ->     END //
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> delimiter ;
```

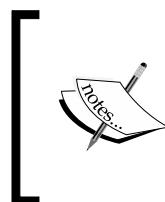
```
mysql> CALL employees.delete_incrementally();
Query OK, 0 rows affected, 137 warnings (3 min 58.09 sec)
```

**How it works...**

The above steps simply create a stored procedure named delete\_incrementally which can be used to delete certain records from the table salaries.

The DELIMITER statements at start and end of the script are necessary to define the procedure, as the statements would otherwise be executed right away. The procedure definition itself is pretty straightforward and basically consists of a REPEAT ... UNTIL loop that deletes data from the salaries table according to the given condition (WHERE emp\_no < 485000). The special part of this DELETE statement is the LIMIT clause, which ensures that no more than 20,000 rows should be deleted. This statement is executed by the REPEAT loop as long as there are any records left to delete. As soon as the number of records affected by the DELETE statement (which can be retrieved using the ROW\_COUNT() function) is zero, the loop ends.

The trick used by this approach is to distribute the period of time needed to delete the data from one block to multiple intervals.



In sum, the incremental deletion in steps of 20,000 actually is considerably slower than a single DELETE statement, but it is much more cooperative when it comes to concurrent write access to the same data.

The benefit lies within the fact that every single partial DELETE statement does not take long, which drastically reduces the period of time in which locks are held for parts of the table. This basically eliminates the locking problems between deletion and other processes.

### SQL client 1

```
CALL delete_incrementally();

=> Query OK, 2702167 rows affected
(1 min 11.85 sec)
```

### SQL client 2

```
INSERT INTO salaries VALUES
0, "2010-01-01", "2099-12-31"
=> Query OK, 1 row affected
(sec)

UPDATE salaries
SET salary=salary+300
WHERE emp_no < "490000";
Query OK, 2669890 rows affected
(59.63 sec)

Rows matched: 2669890  Changed:
2669890  Warnings: 0
```

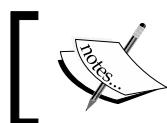
As you can see, even while the deletion still runs, the parallel modifications to the table work concurrently without lock wait timeout errors or similar problems.

### There's more...

While typically the incremental deletion is slower than one single delete operation, it does have one major advantage under heavy load for InnoDB tables: parallel transactions work on a snapshot of the current data at the point in time when the transaction starts. This feature is provided by InnoDB's Multi-Version Concurrency Control (MVCC).

## Managing Data

With many transactions and large amounts of data, the difference between the snapshot and the deleted data has to be maintained by MySQL. This housekeeping data is kept until the transaction that was opened before the deletion was completed is closed. The administration of this delta might have negative impact on the overall MySQL performance. With incremental deletion, this delta data typically does not grow as big as with a long running delete, which often reduces the performance hit.



Under heavy load, the incremental deletion approach might actually cause a gain in overall performance.

### See also

- ▶ *Deleting all but a fragment of a large table's data*
- ▶ *Deleting all data from large tables*

# Monitoring and Analyzing a MySQL Installation

In this chapter, we will discuss some recommendations on how to monitor your MySQL installation and how to analyze possible problems. In detail, the following topics will be covered:

- ▶ Checking free InnoDB tablespace
- ▶ Establishing alerting mechanisms for low remaining tablespace by using triggers
- ▶ Estimating tablespace requirements
- ▶ Identifying and changing MySQL variables
- ▶ Assessing the overall table count
- ▶ Finding the biggest tables
- ▶ Finding all columns with a certain name and/or type
- ▶ Finding all tables referencing each other

## Introduction

Even if MySQL is easy to set up and requires relatively little maintenance once it is installed and running, there are situations in which it is necessary to check certain aspects of MySQL—either to identify and solve certain problems (for example performance problems) or preferably, to prevent trouble in the first place. This chapter will introduce some tools that might prove useful for maintaining your database, so you have some tools at hand to ensure continuous, problem-free operation of your MySQL installation.

*Monitoring and Analyzing MySQL Installation*

## Checking free InnoDB tablespace

One of the most common problems for database administrators is to deal with the data growth most databases will display. You have to keep an eye on the remaining available to avoid the situation where no space is left to add new data. An out-of-space scenario typically leads to a de facto breakdown of your database.

The default storage engine of MySQL is MyISAM, which stores its data in plain files: table, a separate file is created. The file's size is adapted according to the data stored. If more data is written to the table, the file gets bigger. In case of data reduction, the space can be reclaimed by using the `OPTIMIZE TABLE` command, which results in a smaller file size. This way, it is a straightforward task to check how much space is left for further growth: you just have to take a look at the space left on the drive your data (your MySQL data, specifically) is stored in.

For the alternative storage engine InnoDB, this question is not so easy to answer. The following recipe will show you how to retrieve this information.

### Getting ready

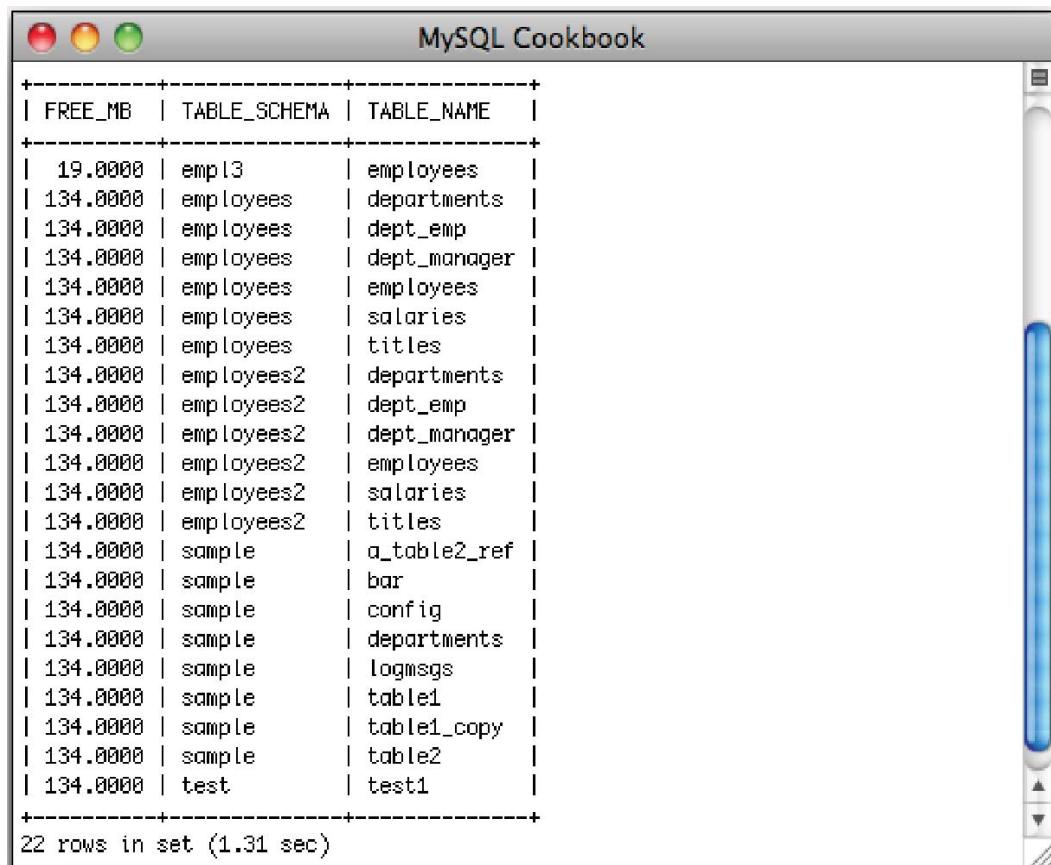
For this recipe, we will need a MySQL user who has `SELECT` access to all schemata in the MySQL installation. We assume the `admin4mysql` account is used throughout the following steps.

### How to do it...

1. Connect to the database using the `admin4mysql` account.
2. Enter the following SQL statement:

```
SELECT DATA_FREE/(1024*1024) AS FREE_MB, TABLE_SCHEMA, TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES WHERE ENGINE="Innodb";
```

3. Read the remaining free tablespace (in MBytes) from the results:



The screenshot shows a MySQL Workbench interface with a results window titled "MySQL Cookbook". The window displays a table with three columns: FREE\_MB, TABLE\_SCHEMA, and TABLE\_NAME. The data shows various tables from different schemas, mostly from the 'employees' schema, with their respective free space in MB. The output ends with a message: "22 rows in set (1.31 sec)".

FREE_MB	TABLE_SCHEMA	TABLE_NAME
19.0000	emp13	employees
134.0000	employees	departments
134.0000	employees	dept_emp
134.0000	employees	dept_manager
134.0000	employees	employees
134.0000	employees	salaries
134.0000	employees	titles
134.0000	employees2	departments
134.0000	employees2	dept_emp
134.0000	employees2	dept_manager
134.0000	employees2	employees
134.0000	employees2	salaries
134.0000	employees2	titles
134.0000	sample	a_table2_ref
134.0000	sample	bar
134.0000	sample	config
134.0000	sample	departments
134.0000	sample	logmsgs
134.0000	sample	table1
134.0000	sample	table1_copy
134.0000	sample	table2
134.0000	test	test1

22 rows in set (1.31 sec)

## How it works...

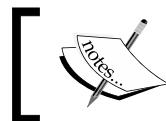
The statement entered in step 2 simply reads the DATA\_FREE column from the table INFORMATION\_SCHEMA.TABLES and displays it in a convenient way (including a conversion to show MBytes instead of bytes). For each table in each schema, the remaining free tablespace is shown (in MByte).

As you have noticed, most tables in the above example show the identical value of 134 MBytes. This is due to the fact that all these tables are stored in a shared MySQL tablespace (as defined by the innodb\_data\_home\_dir variable). For most InnoDB installations, this will be the typical setup: all data is stored in a common tablespace, thus all tables will have an identical value for remaining tablespace.

In our example, however, the first table shows a different value. This is because the first table was created in a table-specific tablespace, which can be achieved by using the innodb\_file\_per\_table parameter in the MySQL configuration. All tables that are created with this parameter are created in separate files. The files are extended in size as needed, but are not automatically deleted if data is deleted from the table.

The value retrieved by the above recipe shows the size available with the current file system (both for shared and file-per-table tablespace). If autoextending tablespaces are used, the remaining space on the file system has to be added to this value to calculate the total available space.

## Monitoring and Analyzing MySQL Installation



Remaining autoextend space = Free tablespace + Remaining file system space

Autoextending tablespace is in place when the `innodb_file_per_table` setting if the `innodb_data_home_dir` definition includes an autoextend suffix.



Note that for `innodb_data_home_dir` the default value `ibdata1:10M:autoextend` is used, which means automatic extension is in place!

In these cases, the free tablespace value discussed above is not the only relevant you should additionally establish a monitoring for the available space on the disk where the tablespace files are stored on.



Please note that for both autoextend and `innodb_file_per_table` the tablespace will be extended as needed, but once the space is allocated for InnoDB use, it will not be released automatically if less storage is needed. With `innodb_file_per_table`, unused tablespace can be released by executing an `OPTIMIZE TABLE` command, resulting in smaller tablespace files.

### There's more...

The above approach will only work for MySQL version 5.1.28 or higher, as these versions added the `TABLE_COMMENT` column to the `INFORMATION_SCHEMA.TABLES`. To retrieve the data from versions before that, you will have to take a look at the `TABLE_COMMENT` column:

```
MySQL Cookbook

mysql> SELECT TABLE_COMMENT, TABLE_SCHEMA, TABLE_NAME FROM INFORMATION_SCHEMA.TABLES WHERE ENGINE="InnoDB";
+-----+-----+-----+
| TABLE_COMMENT | TABLE_SCHEMA | TABLE_NAME |
+-----+-----+-----+
| InnoDB free: 34940928 kB | test | article |
| InnoDB free: 34940928 kB | test | class |
| InnoDB free: 34940928 kB | test | config |
+-----+-----+-----+
3 rows in set (0.82 sec)
```

---

An alternative way to retrieve this information without explicitly accessing INFORMATION SCHEMA is to use the SHOW TABLE STATUS command, which shows the status (including comment) for each table in the currently selected schema.

## See also

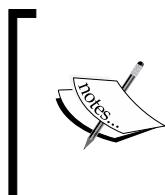
- ▶ *Defining an alternative user for administrative tasks*

# Establishing alerting mechanisms for low remaining tablespace by using triggers

In the previous recipe, we introduced a way to read the remaining InnoDB tablespace size. It is important to have this information at hand, in a professional setting you will have to make sure that your database will not run out of tablespace. To avoid having to manually check the remaining tablespace on a regular basis (you have better things to do, especially on a weekend and on vacation, right?), an automatic monitoring mechanism is needed.

It is not in the scope of this book to describe how to establish a working monitoring and alerting infrastructure. On most platforms you will have means to monitor for certain conditions and/or alert people or groups responsible for a particular issue. Typical solutions for alerting are third-party products like Nagios, Insight Manager, or OpenNMS, or even the good old e-mail. Alerts can be triggered for example by trigger files, certain log entries, SNMP, or specific clients.

In this recipe, we will show you how to establish a monitoring mechanism for remaining InnoDB tablespace that can easily be adapted to use the alerting mechanism of your choice.



Note that this recipe is targeted at installations with a fixed InnoDB tablespace size. Scenarios that rely on the *autoextend* feature of InnoDB have to take the available file system space into account as well.

## Getting ready

First of all, you will have to come up with a threshold value for your free tablespace. If the available free tablespace drops below this threshold, an alert will be raised.



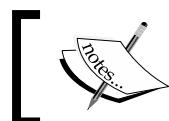
20 percent is a sensible starting point for a tablespace alert, but this value should be adapted according to your needs. Data fluctuation and regular backups for example might require a higher threshold.

## *Monitoring and Analyzing MySQL Installation*

Throughout this recipe, we will assume a fixed size InnoDB tablespace of 10 GB and a threshold value of 2 GB.

We will additionally need a MySQL user with administrative privileges; for the following example, we will use the `admin4mysql` account again.

And finally, you will have to give a directory on your server's file system into which to write the trigger file that is not used by other applications (we will assume `C:/temp/MySQLMonitoring/` in the following steps).



The recipe makes use of MySQL's scheduler feature, so it requires MySQL version 5.1 or greater.

### **How to do it...**

1. Connect to your MySQL database
2. Enter the following commands to create a stored procedure to check for low tablespace:

```
mysql> delimiter //
mysql> CREATE PROCEDURE mysql.check_innodb_ts()
      -> BEGIN
      ->   SELECT MIN(DATA_FREE) FROM INFORMATION_SCHEMA.TABLES
      ->     WHERE ENGINE="InnoDB" INTO @free;
      ->   SET @threshold := 2*1024*1024*1024;
      ->   SET @fileprefix := "C:/log/innodb_free_ts_alert_";
      ->   SELECT DATE_FORMAT(NOW(), "%Y_%m_%d_%H%i%s")
      ->     INTO @timestamp_suffix;
      ->   SELECT CONCAT(@fileprefix, @timestamp_suffix)
      ->     INTO @filename;
      ->   SELECT CONCAT(
      ->     "Free InnoDB table space (",
      ->     @free,
      ->     ") is below warning threshold (",
      ->     @threshold,
      ->     ").") INTO @warning;
      ->   SELECT CONCAT(
      ->     "SELECT @warning INTO OUTFILE ''",
      ->     @filename,
```

---

```

->          PREPARE statement FROM @command;
->          EXECUTE statement;
->  END IF;
-> END //
```

**Query OK, 0 rows affected (0.00 sec)**

**mysql> delimiter ;**

3. Use the following commands to schedule the tablespace check:

```
mysql> CREATE EVENT mysql.check_innodb_ts_event
```

```
    -> ON SCHEDULE EVERY 15 MINUTE
```

```
    -> DO CALL mysql.check_innodb_ts();
```

**Query OK, 0 rows affected (0.00 sec)**

4. Configure your platform-specific monitoring/alerting mechanism to scan for files named `innodb_free_ts_alert_*` in the `C:/log/` directory—if a file exists, an alert should be raised.

## How it works...

In step 2, a stored procedure named `mysql.check_innodb_ts()` is created, which checks the remaining InnoDB tablespace against a threshold value (here: 2 GB, given as arithmetic expression: `2 * 1024 * 1024 * 1024`). If the space is below the threshold value, a file with a given name prefix and the timestamp as a name suffix is written to the `C:/log/` directory. The file contains an alert message that states the actual as well as the threshold value.

Step 3 creates a scheduled event (named `mysql.check_innodb_ts_event`), which causes the stored procedure defined in step 2 to be executed automatically every 15 minutes (the MySQL scheduler has to be enabled, see *There's more* below). You should adapt the interval according to your needs, but keep in mind that longer intervals will increase the probability that a sudden peak in data growth might fill up your database before your alert fires. On the other hand, a shorter interval will be at the cost of an increased server load induced by the monitoring mechanism. You will have to balance these aspects to find a solution for your environment.

Step 4 should be considered a placeholder for the respective steps required to produce an alert in your specific environment. If your monitoring tool is not able to check for the existence of files with a specific file pattern (`C:\log\innodb_free_ts_alert_*` in our example), you might have to introduce an intermediate layer that checks for the trigger files created by the MySQL event scheduler on a regular basis. An example for a Windows environment would be to define a scheduled task that executes a command-line script along the lines of:

```
IF EXIST "C:\log\innodb_free_ts_alert_*(alert.exe "Table space ...")"
```

## *Monitoring and Analyzing MySQL Installation*

For Unix-like systems, a *bash* script like the following run by a *cron* job would have similar effect:

```
files=$(ls /log/innodb_free_ts_alert_* 2> /dev/null)
if [ $files ]; then /etc/bin/alert "Table space low!"; fi
```

For these examples, *alert.exe* and */etc/bin/alert* have to be replaced by the your choice to raise an alert.

### **There's more...**

For the sake of brevity, some prerequisites and further options for improvement were discussed in the above recipe. The following sections will show you how to make sure the prerequisites are given and how to ease future changes in configuration.

## **Enabling the MySQL scheduler**

The scheduled event from step 3 relies on a running MySQL scheduler.



The scheduler feature has been available since version 5.1, so the recipe does not fully work with MySQL 5.0 or lower.

As the scheduler is not enabled by default even in MySQL 5.1, you should make sure the configuration contains the setting `event_scheduler=ON` in the `[mysqld]` section of the MySQL configuration file. Alternatively, you can enable the scheduler using the `SET event_scheduler = ON;` command.

If you happen to work with a version of MySQL that does not support the scheduler for any reason you cannot enable the MySQL scheduler in your installation, you should consider establishing a scheduled task (for Windows) or a cron job (for Unix/Linux), which executes the `CALL mysql.check_innodb_ts();` command using the `-e` option of the `mysql` command-line client.

## **Improving configuration**

As the stored procedure defined in step 2 contains some hard-coded values (most notably `threshold` and `fileprefix`), it is hard to adapt those values to changed needs, for example if the threshold should be changed. Thus you could consider reading the specific value from a configuration table using the `SELECT ... INTO @variable` notation. To change a configuration value, the stored procedure can be left unchanged; you only need to change the values in your configuration table.

## See also

- ▶ *Defining an alternative user for administrative tasks* (Chapter 8 MySQL User Management)

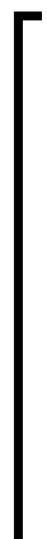
# Estimating tablespace requirements

When planning a database installation, one problem that comes up pretty soon is drive space should be reserved for the data. This is an important aspect to consider of reliability because if the reserved space is too small, your database might grind if no space is left to store additional data. On the other hand, today's cost pressure not allow for demands that are not based on a traceable method of demand estimation. The following recipe will present an approach that allows for a realistic estimate of the requirements of your database.

## Getting ready

For the following recipe, you will need to have some information at hand:

- ▶ The table structure of your database
- ▶ An estimate of the maximum number of records for each table (take data into account)
- ▶ Representative sample records for each table (the more, the better)



### Choosing sample values for variable length columns carefully:

When thinking about sample records for your tables, think about reasonable values for columns with data types of variable length (like VARCHAR or BLOB). The length of the values should match the average size of the expected values of the productive database. If you cannot give a valid estimate, we propose to use sample values half as long as the maximum.

For international operations, take Unicode encoding of VARCHAR values into account. If you have to deal with character sets like for example Cyrillic, Chinese, Japanese, Arabic, or Hebrew, use sample values for these as well, as these characters will require more space than an ASCII character.

The table structure of your database should already be present, which means all tables should be created, but not necessarily filled with data.

## *Monitoring and Analyzing MySQL Installation*

---

For the following recipe, we will assume the `sample_stduser` account is used. For this example, we will use a database `sample` with two tables—`table1` and `table2`—created using the following statements:

```
CREATE TABLE `table1` (
  `id` LONG,
  `name` varchar(255) DEFAULT NULL,
  INDEX `idx_name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `table2` (
  `name` char(16) NOT NULL,
  `description` varchar(128) NOT NULL,
  PRIMARY KEY (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

We will assume that `table1` will have a maximum row count of one million, and `table2` will hold at most 20,000 records.

### **How to do it...**

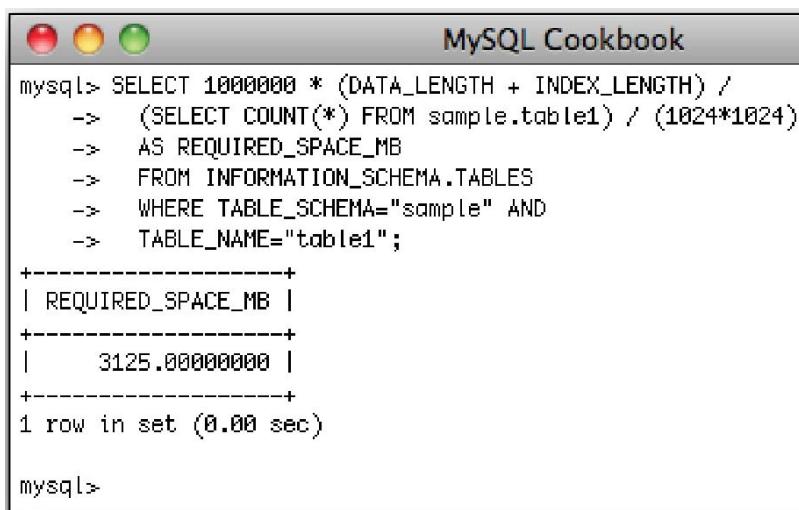
1. Connect to your database using the `sample_stduser` account.
2. Insert as much sample data into your tables as possible:

```
INSERT INTO table1 values
(1, "John Doe"),
(2, "Mickey Mouse"),
(3, "Дмитрий Анатольевич Медведев"),
(4, "Jane Doe"),
(5, "Jeffrey \"The Dude\" Lebowksi"),
(6, "Walter Sobchak"),
(7, "Donny Kerabatsos"),
(8, "Neo"),
(9, "Trinity"),
(10, "Morpheus");
```

```
INSERT INTO table2 values
("Bit", "Smallest piece of binary logic: either 0 or 1"),
("Byte", "Consists of eight bits"),
("Nibble", "Half a byte, consists of four bits"),
("kB", "Kilobyte; either 1,000 or 1,024 bytes"),
("KiB", "Kibibyte; correct IEC term for 1,024 bytes"),
("MB", "Megabyte: either 1,000,000 or 1,048,576 bytes") .
```

3. Calculate the tablespace requirements for each table by using the following

```
mysql> SELECT 1000000 * (DATA_LENGTH + INDEX_LENGTH) /
      ->   (SELECT COUNT(*) FROM sample.table1) / (1024*1024)
      -> AS REQUIRED_SPACE_MB
      -> FROM INFORMATION_SCHEMA.TABLES
      -> WHERE TABLE_SCHEMA="sample" AND
      -> TABLE_NAME="table1";
```



The screenshot shows a MySQL command-line interface window titled "MySQL Cookbook". The query executed is:

```
mysql> SELECT 1000000 * (DATA_LENGTH + INDEX_LENGTH) /
      ->   (SELECT COUNT(*) FROM sample.table1) / (1024*1024)
      -> AS REQUIRED_SPACE_MB
      -> FROM INFORMATION_SCHEMA.TABLES
      -> WHERE TABLE_SCHEMA="sample" AND
      -> TABLE_NAME="table1";
```

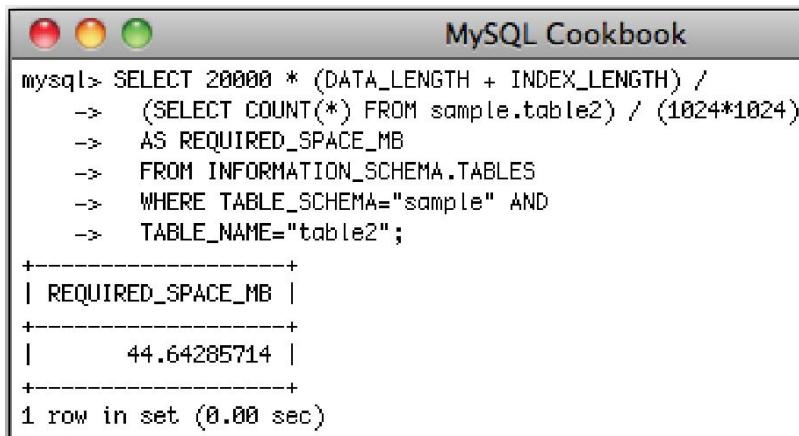
The result is displayed in a table:

REQUIRED_SPACE_MB
3125.00000000

1 row in set (0.00 sec)

mysql>

```
mysql> SELECT 20000 * (DATA_LENGTH + INDEX_LENGTH) /
      ->   (SELECT COUNT(*) FROM sample.table2) / (1024*1024)
      -> AS REQUIRED_SPACE_MB
      -> FROM INFORMATION_SCHEMA.TABLES
      -> WHERE TABLE_SCHEMA="sample" AND
      -> TABLE_NAME="table2";
```



The screenshot shows a MySQL command-line interface window titled "MySQL Cookbook". The query executed is:

```
mysql> SELECT 20000 * (DATA_LENGTH + INDEX_LENGTH) /
      ->   (SELECT COUNT(*) FROM sample.table2) / (1024*1024)
      -> AS REQUIRED_SPACE_MB
      -> FROM INFORMATION_SCHEMA.TABLES
      -> WHERE TABLE_SCHEMA="sample" AND
      -> TABLE_NAME="table2";
```

The result is displayed in a table:

REQUIRED_SPACE_MB
44.64285714

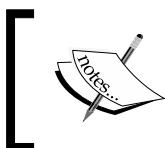
1 row in set (0.00 sec)

mysql>

## Monitoring and Analyzing MySQL Installation

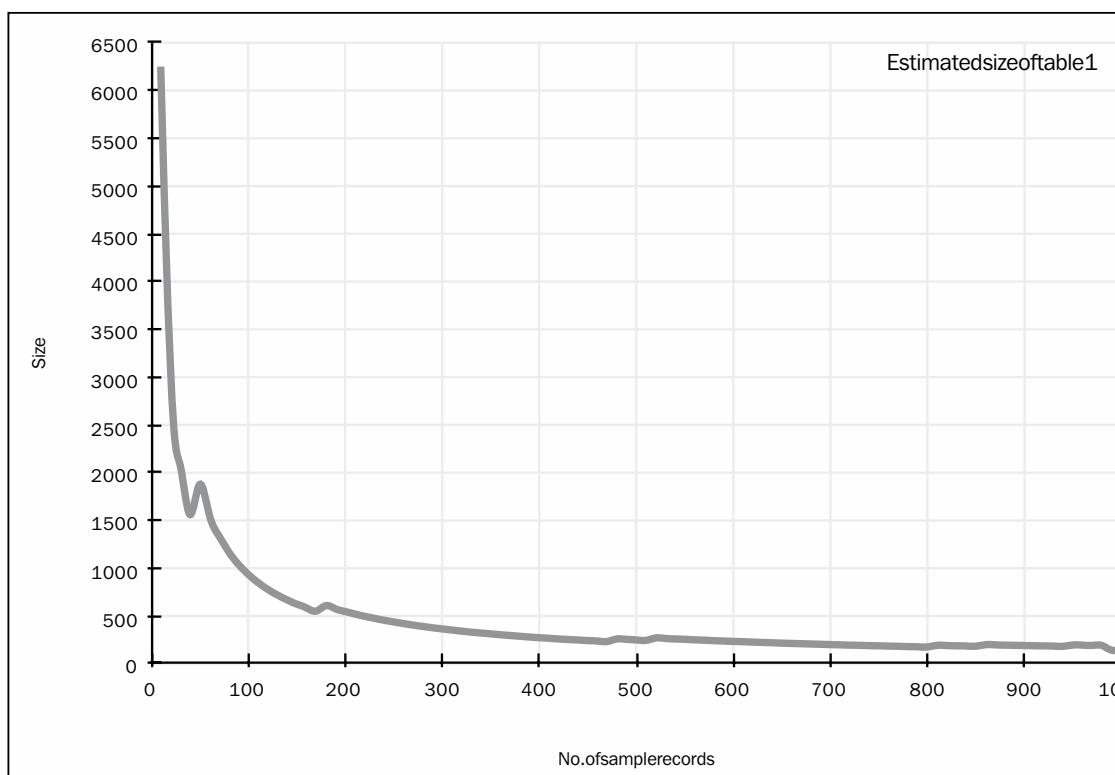
### How it works...

The statements in step 3 use the metadata available in `INFORMATION_SCHEMA`. We calculate the space requirements for the table. We retrieve the current space required by the table, current data and index information, and divide it by the number of records currently present in the table, which gives the average size of a single record. We simply multiply this value by the target number of records in the table. For better readability, the value is scaled to MBytes.



Note that for a low number of sample records, these estimates are typically way too high. The more records are present in the tables, the more accurate the results are.

The following diagram shows how the sample data row count affects the results of the calculations:



As you can see, the initial estimate was way too conservative. For low numbers of sample records, the estimated table size was extremely high. The more sample data is present, the lower (and more precise) the estimates get.

## There's more...

The reason for the inexact figures produced when too little sample data is used is the fact that storage space for data and index information is reserved in pages (a page is 16 KB). So for the first row, 32 KB of storage is reserved (one page for data, one page for the index). Based on this, one million rows of this size would require roughly 32 GB of storage. The next row will fit into the very same page that was already reserved as well, thus reducing the calculated estimate for the space requirements by 50 percent. This is why with sets of sample data, a realistic estimate of how much space is going to be required based on the row count is hardly possible. The advantage is that this effect protects you against estimates that are too low.



Note that simply multiplying the `AVG_ROW_LENGTH` value by the number of rows is not sufficient because this does not take the storage requirements for the index information into account. In some cases, the indexes are larger than the actual data!

## See also

- ▶ *Creating a basic user*

# Identifying and changing MySQL variables

The behavior of MySQL installations can be widely configured using variables. You probably have come across some of these variables, as they are defined in your MySQL configuration file (`my.ini`). But there are many variables that you will probably not have modified because they have sensible default values and are rarely modified.

The typical way to adapt MySQL variables is to edit the MySQL configuration and restart the database server. But if you want to know whether a setting in your MySQL configuration has been fact accepted or if you want to know which setting is in place in the currently running instance without having to resort to the MySQL startup configuration, this recipe will show you how to do this.

We will also show you how to change certain settings during server runtime, which allows you to perform certain changes without the need for a MySQL restart.

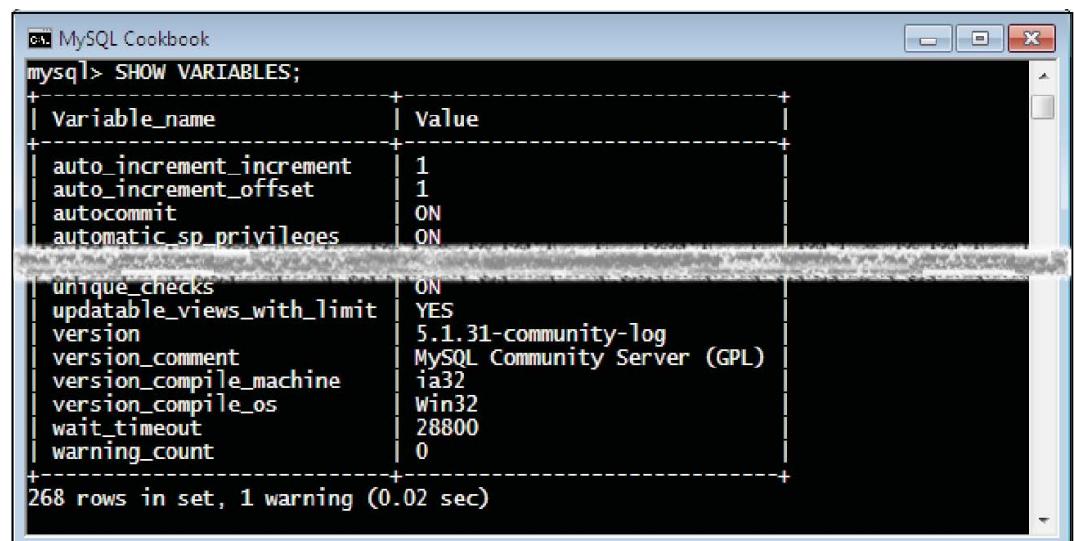
## Getting ready

All we need is any MySQL client (like the `mysql` command-line client) and a MySQL user account. To read a variable setting, any user that allows you to connect to the server will do.

*Monitoring and Analyzing MySQL Installation***How to do it...**

1. Connect to your MySQL database with the SQL client of your choice using admin4mysql account.
2. Show all variables by entering the following command (you will have to scroll the results to see all values):

```
mysql> SHOW VARIABLES;
```

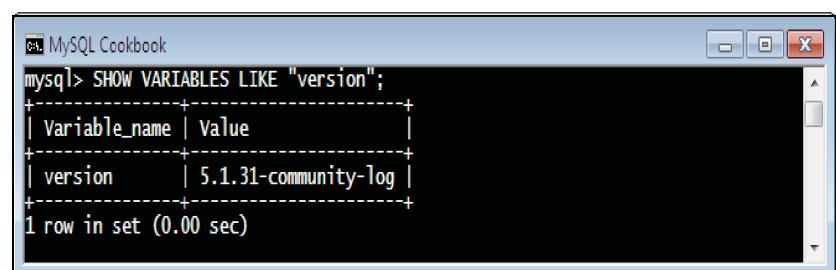


Variable_name	Value
auto_increment_increment	1
auto_increment_offset	1
autocommit	ON
automatic_sp_privileges	ON
unique_checks	ON
updatable_views_with_limit	YES
version	5.1.31-community-log
version_comment	MySQL Community Server (GPL)
version_compile_machine	ia32
version_compile_os	Win32
wait_timeout	28800
warning_count	0

268 rows in set, 1 warning (0.02 sec)

3. To display the value of a certain variable, execute the following statement

```
mysql> SHOW VARIABLES LIKE "version";
```

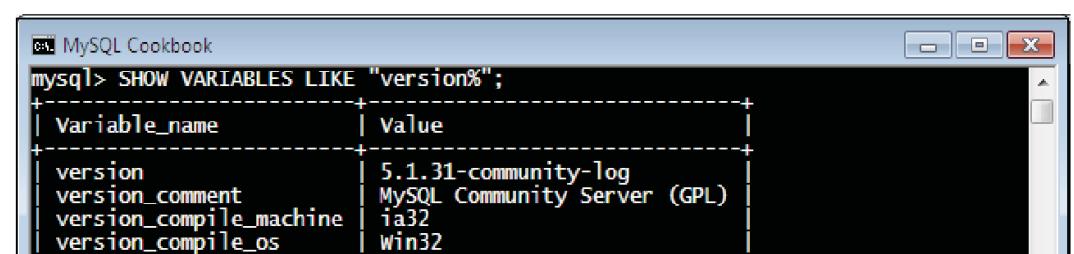


Variable_name	Value
version	5.1.31-community-log

1 row in set (0.00 sec)

4. To display a group of variables with a common name, you can also use SQL wildcards:

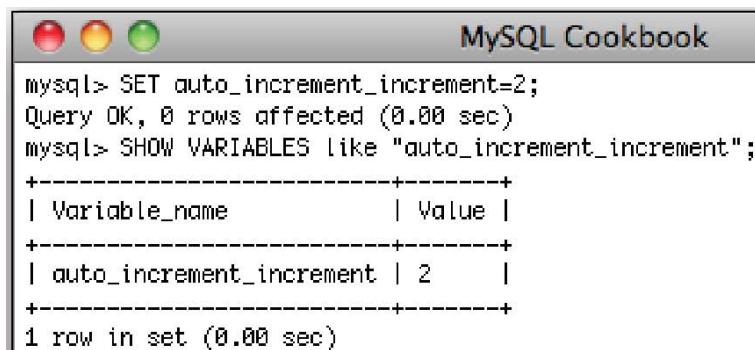
```
mysql> SHOW VARIABLES LIKE "version%";
```



Variable_name	Value
version	5.1.31-community-log
version_comment	MySQL Community Server (GPL)
version_compile_machine	ia32
version_compile_os	Win32

- 
5. To modify a variable **for your connection only**, use the SET command:

```
mysql> SET auto_increment_increment=2;
Query OK, 0 rows affected (0.00 sec)
```



The screenshot shows a MySQL Workbench window titled "MySQL Cookbook". It displays the following SQL session:

```
mysql> SET auto_increment_increment=2;
Query OK, 0 rows affected (0.00 sec)
mysql> SHOW VARIABLES like "auto_increment_increment";
+-----+-----+
| Variable_name      | Value   |
+-----+-----+
| auto_increment_increment | 2      |
+-----+-----+
1 row in set (0.00 sec)
```

6. To globally modify a variable, use set SET GLOBAL statement:

```
mysql> SET GLOBAL auto_increment_increment=3;
Query OK, 0 rows affected (0.00 sec)
```

## How it works...

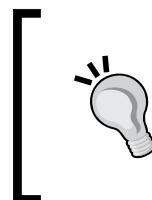
Steps 2 through 4 show different ways to retrieve variable settings, which is pretty forward. Step 5 changes a server variable to a new value (in the above example, we change auto\_increment\_increment to a value of 2). This setting takes effect immediately, but only affects your own connection. Any other connection will use the previous value. As soon as you drop your connection and reconnect to the server, the variable is reset to its original value again.

Step 6 shows you how to apply a change not only to your current connection. With the GLOBAL syntax, the setting is applied to other connections as well. However, note that some changes are applied to other connections immediately: some changes will only affect new connections. The reason for this is that there are basically three kinds of MySQL variables:

1. Variables that apply to your connection (called **session variables**).
2. Variables that are defined globally and affect all connections alike (**global variables**).
3. Variables that exist both globally and for your current session, and which are changed independently for each context.

If a variable that is solely defined globally is changed, then the change will affect all connections, both existing and new ones. For variables that are both session and global variables (auto\_increment\_increment is an example for this), any change will only affect **new** connections. The variable values for connections that already exist will remain unchanged. This behavior is due to the fact that on connection creation the value of the global variable is copied to the session variable. Any change to the global variable will not affect the session variable.

## Monitoring and Analyzing MySQL Installation



Note that all changes made using the `SET GLOBAL` command will be lost on the next MySQL startup! To make permanent changes to the MySQL configuration, you will have to edit the startup configuration (typically the `my.ini` file) as well.

### There's more...

The following sections will introduce some additional options, which can be used to specifically read information from the MySQL variables.

### Displaying more than one named variable at a time

You can also display the values for more than one named variable in one statement using the following syntax:

```
mysql> SHOW VARIABLES
      -> WHERE variable_name IN ("wait_timeout", "autocommit")
      -> OR variable_name LIKE "version%";
```

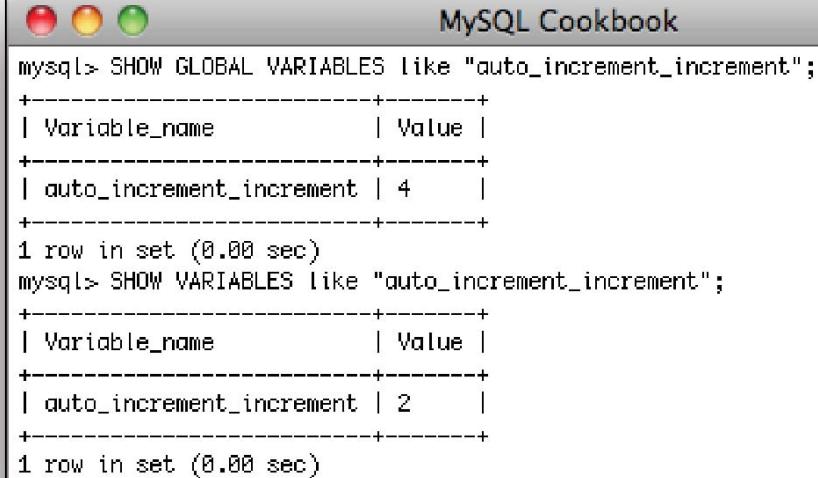
Variable_name	Value
autocommit	ON
version	5.1.31-community-log
version_comment	MySQL Community Server (GPL)
version_compile_machine	ia32
version_compile_os	Win32
wait_timeout	28800

6 rows in set (0.00 sec)

### Displaying global settings

If you have changed any of the settings of your connection and you want to find out the current global setting of this variable, use the `SHOW GLOBAL` statement:

```
mysql> SHOW GLOBAL VARIABLES like "auto_increment_increment";
```



```

mysql> SHOW GLOBAL VARIABLES like "auto_increment_increment";
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| auto_increment_increment | 4    |
+-----+-----+
1 row in set (0.00 sec)

mysql> SHOW VARIABLES like "auto_increment_increment";
+-----+-----+
| Variable_name      | Value |
+-----+-----+
| auto_increment_increment | 2    |
+-----+-----+
1 row in set (0.00 sec)

```

## See also

- ▶ *Defining an alternative user for administrative tasks* (Chapter 8. MySQL User Management)

# Assessing the overall table count

In some MySQL installations, you will have a lot of databases (of schemata) in place. To keep track of the databases, it sometimes comes in handy to get an overview of the tables that reside in each schema. The following recipe will show you how to achieve this.

## Getting ready

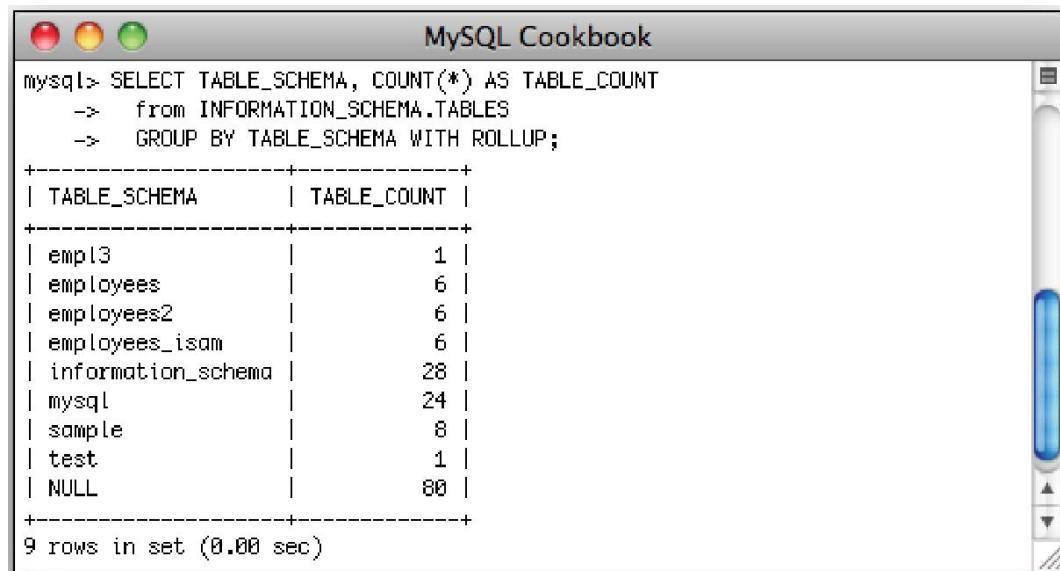
You will need a MySQL account to reproduce the following recipe. To get an overview of all the databases of your installations, you need at least SELECT privileges on all databases. That is why we assume the administrative user account named `admin4mysql` is used. The mentioned methods will work for more restricted accounts as well, but will produce results only for the databases accessible to the user.

## Monitoring and Analyzing MySQL Installation

### How to do it...

1. Connect to your MySQL database using the admin4mysql account.
2. Execute the following command:

```
mysql> SELECT TABLE_SCHEMA, COUNT(*) AS TABLE_COUNT
      ->   from INFORMATION_SCHEMA.TABLES
      -> GROUP BY TABLE_SCHEMA WITH ROLLUP;
```



The screenshot shows a window titled "MySQL Cookbook" displaying the output of a MySQL query. The query is:

```
mysql> SELECT TABLE_SCHEMA, COUNT(*) AS TABLE_COUNT
      ->   from INFORMATION_SCHEMA.TABLES
      -> GROUP BY TABLE_SCHEMA WITH ROLLUP;
```

TABLE_SCHEMA	TABLE_COUNT
empl3	1
employees	6
employees2	6
employees_isam	6
information_schema	28
mysql	24
sample	8
test	1
NULL	80

9 rows in set (0.00 sec)

### How it works...

The result from step 2 displays an overview of the databases that are present in our installation and the number of tables defined for each database. The last line shows the overall table count of all databases.

### See also

- ▶ [Defining an alternative user for administrative tasks](#)

## Finding the biggest tables

On the quest for performance during the daily struggle against uncontrolled data growth, the biggest tables are often the most promising candidates for optimization. This recipe will show you how to get an overview of the largest tables in your installation.

## Getting ready

You will only need an appropriate MySQL user account to perform the steps of this recipe. To retrieve the information for all databases, an administrative user like admin4mysql (which we will use here) is best. To get an overview of the tables in your database, with access only to the respective database can be used as well.

## How to do it...

1. Connect to your MySQL database using the admin4mysql account.
2. Perform the following SQL statement:

```
mysql> SELECT TABLE_SCHEMA,
        ->   TABLE_NAME,
        ->   (INDEX_LENGTH+DATA_LENGTH)/(1024*1024) AS SIZE_MB,
        ->   TABLE_ROWS
        ->   FROM INFORMATION_SCHEMA.TABLES
        ->   WHERE TABLE_SCHEMA NOT IN("mysql", "information_schema")
        ->   ORDER BY SIZE_MB DESC;
```

The screenshot shows a terminal window titled "MySQL Cookbook". The command entered is:

```
mysql> SELECT TABLE_SCHEMA,
        ->   TABLE_NAME,
        ->   (INDEX_LENGTH+DATA_LENGTH)/(1024*1024) AS SIZE_MB,
        ->   TABLE_ROWS
        ->   FROM INFORMATION_SCHEMA.TABLES
        ->   WHERE TABLE_SCHEMA NOT IN("mysql", "information_schema")
        ->   ORDER BY SIZE_MB DESC;
```

The output displays a table with the following data:

TABLE_SCHEMA	TABLE_NAME	SIZE_MB	TABLE_ROWS
employees2	salaries	130.1875	2844513
employees	titles	30.1094	444546
employees2	titles	30.1094	442465
employees_isam	titles	25.8517	443308
employees	dept_emp	22.5469	331883
employees_isam	dept_manager	0.0044	24
employees_isam	departments	0.0031	9
employees_isam	salaries	0.0010	0

28 rows in set (1.08 sec)

## How it works...

The above statement simply makes use of the table metadata available from the INFORMATION\_SCHEMA.TABLES and displays it in a readable way. It is sorted by

## *Monitoring and Analyzing MySQL Installation*

```
SELECT TABLE_SCHEMA,
       TABLE_NAME,
       TABLE_ROWS,
       (INDEX_LENGTH+DATA_LENGTH) / (1024*1024) AS SIZE_MB
  FROM INFORMATION_SCHEMA.TABLES
 WHERE TABLE_SCHEMA NOT IN("mysql", "information_schema")
 ORDER BY TABLE_ROWS DESC;
```

In both cases, the WHERE TABLE\_SCHEMA NOT IN ("mysql", "information\_schema") clause helps to filter out any results from the mysql and information\_schemata, which are typically not in the focus of interest.

### See also

- ▶ *Defining an alternative user for administrative tasks*

## Finding all columns with a certain name and/or type

In large databases, it often makes sense to agree on some kind of data modeling to avoid unnecessary effort. For example, you decide to standardize that all columns containing a name should be of type VARCHAR(64). If, however, the situation changes and the standard type for name columns should be changed to, say, VARCHAR(128) to accommodate very long names, the question arises how many columns or tables have to be adapted. In large installations, this is not a trivial question to answer. This recipe will show you how to retrieve this information.

### Getting ready

You will need a MySQL user with SELECT rights for the databases that are to be checked. You will also need to know what data type and column name you are looking for. In the following steps, we will be looking for name columns and columns of data type VARCHAR.

### How to do it...

1. Connect to your MySQL database using the sample\_guest user.
2. Execute the following query to find all name columns:

```
mysql> SELECT TABLE_SCHEMA,
```

```

-> DATA_TYPE,
-> CHARACTER_MAXIMUM_LENGTH AS SIZE
-> FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE COLUMN_NAME="name" AND
-> TABLE_SCHEMA NOT IN ("mysql", "information_schema");

```

**MySQL Cookbook**

```

mysql> SELECT TABLE_SCHEMA,
-> TABLE_NAME,
-> COLUMN_NAME,
-> DATA_TYPE,
-> CHARACTER_MAXIMUM_LENGTH AS SIZE
-> FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE COLUMN_NAME="name" AND
-> TABLE_SCHEMA NOT IN ("mysql", "information_schema");
+-----+-----+-----+-----+-----+
| TABLE_SCHEMA | TABLE_NAME | COLUMN_NAME | DATA_TYPE | SIZE |
+-----+-----+-----+-----+-----+
| sample      | config     | name       | varchar   | 64  |
| sample      | table1    | name       | varchar   | 255 |
| sample      | table2    | name       | char      | 16   |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

```

3. To find all columns with data type VARCHAR ( 64 ) , execute this command:

```

mysql> SELECT TABLE_SCHEMA,
-> TABLE_NAME,
-> COLUMN_NAME,
-> DATA_TYPE,
-> CHARACTER_MAXIMUM_LENGTH AS SIZE
-> FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE DATA_TYPE="VARCHAR" AND
-> CHARACTER_MAXIMUM_LENGTH=64 AND
-> TABLE_SCHEMA NOT IN ("mysql", "information_schema");

```

**MySQL Cookbook**

```

mysql> SELECT TABLE_SCHEMA,
-> TABLE_NAME,
-> COLUMN_NAME,
-> DATA_TYPE,
-> CHARACTER_MAXIMUM_LENGTH AS SIZE
-> FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE DATA_TYPE="VARCHAR" AND
-> CHARACTER_MAXIMUM_LENGTH=64 AND
-> TABLE_SCHEMA NOT IN ("mysql", "information_schema");
+-----+-----+-----+-----+-----+
| TABLE_SCHEMA | TABLE_NAME | COLUMN_NAME | DATA_TYPE | SIZE |
+-----+-----+-----+-----+-----+
| sample      | config     | name       | varchar   | 64  |
| sample      | config     | value      | varchar   | 64  |
| sample      | nametest   | surname    | varchar   | 64  |
+-----+-----+-----+-----+-----+

```

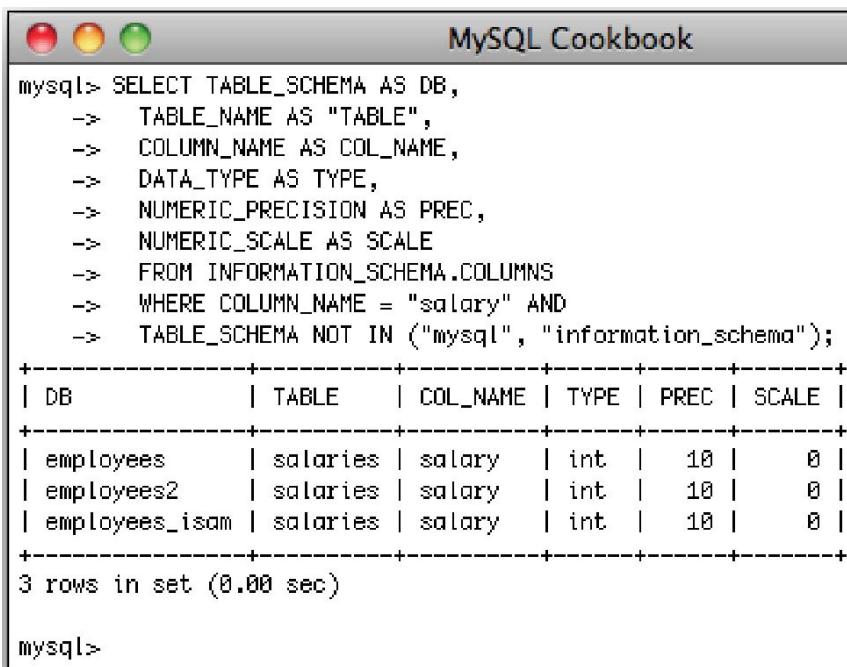
## Monitoring and Analyzing MySQL Installation

### How it works...

The preceding statements simply make use of the information available in the INFORMATION\_SCHEMA.COLUMNS metadata table. It helps you identify columns with certain attributes. The mysql and information\_schema schemata are filtered out using the WHERE TABLE\_SCHEMA NOT IN("mysql", "information\_schema") clause to display only results of interest.

### There's more...

If the data type you are looking for is a numeric data type, you will have to adapt the query to take the NUMERIC\_PRECISION and NUMERIC\_SCALE attributes into account:

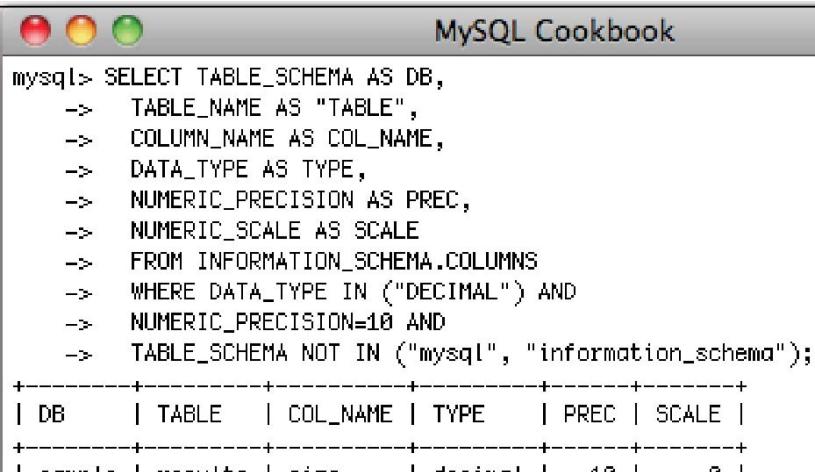


```
MySQL Cookbook

mysql> SELECT TABLE_SCHEMA AS DB,
->   TABLE_NAME AS "TABLE",
->   COLUMN_NAME AS COL_NAME,
->   DATA_TYPE AS TYPE,
->   NUMERIC_PRECISION AS PREC,
->   NUMERIC_SCALE AS SCALE
->   FROM INFORMATION_SCHEMA.COLUMNS
->   WHERE COLUMN_NAME = "salary" AND
->   TABLE_SCHEMA NOT IN ("mysql", "information_schema");
+-----+-----+-----+-----+-----+
| DB    | TABLE | COL_NAME | TYPE  | PREC | SCALE |
+-----+-----+-----+-----+-----+
| employees | salaries | salary | int   | 10   | 0    |
| employees2 | salaries | salary | int   | 10   | 0    |
| employees_isam | salaries | salary | int   | 10   | 0    |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql>
```

You can also narrow down the results to columns with a specific precision or scale:



```
MySQL Cookbook

mysql> SELECT TABLE_SCHEMA AS DB,
->   TABLE_NAME AS "TABLE",
->   COLUMN_NAME AS COL_NAME,
->   DATA_TYPE AS TYPE,
->   NUMERIC_PRECISION AS PREC,
->   NUMERIC_SCALE AS SCALE
->   FROM INFORMATION_SCHEMA.COLUMNS
->   WHERE DATA_TYPE IN ("DECIMAL") AND
->   NUMERIC_PRECISION=10 AND
->   TABLE_SCHEMA NOT IN ("mysql", "information_schema");
+-----+-----+-----+-----+-----+
| DB    | TABLE | COL_NAME | TYPE  | PREC | SCALE |
+-----+-----+-----+-----+-----+
| employees | results | price | decimal | 10   | 2    |
+-----+-----+-----+-----+-----+
```

## See also

- ▶ *Creating a read-only account*

# Finding all tables referencing each other

For both manual data modifications and changes to the table structure, it is important to know whether the referential integrity of the database is affected. For complex databases, it is often the case that the details about which table references are in place are not well known. This recipe helps you to make the dependencies between tables visible.

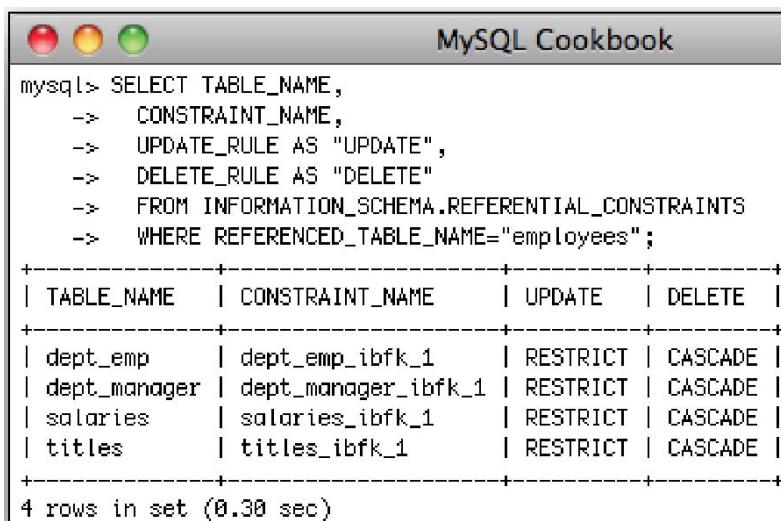
## Getting ready

Again, we only need a user who has the privileges to access the database in question (SELECT privileges are sufficient). We will use an account named `employees_guest` in this account here, which has SELECT privileges for the employees database.

## How to do it...

1. Connect to your MySQL database using the `sample_guest` user.
2. To display all tables referencing the `employees` table, execute the following query:

```
mysql> SELECT TABLE_NAME,
    ->     CONSTRAINT_NAME,
    ->     UPDATE_RULE AS "UPDATE",
    ->     DELETE_RULE AS "DELETE"
    ->     FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
    ->     WHERE REFERENCED_TABLE_NAME="employees";
```



The screenshot shows a terminal window titled "MySQL Cookbook". The command entered is:

```
mysql> SELECT TABLE_NAME,
->     CONSTRAINT_NAME,
->     UPDATE_RULE AS "UPDATE",
->     DELETE_RULE AS "DELETE"
->     FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
->     WHERE REFERENCED_TABLE_NAME="employees";
```

The output displays four rows of data from the INFORMATION\_SCHEMA.REFERENTIAL\_CONSTRAINTS table:

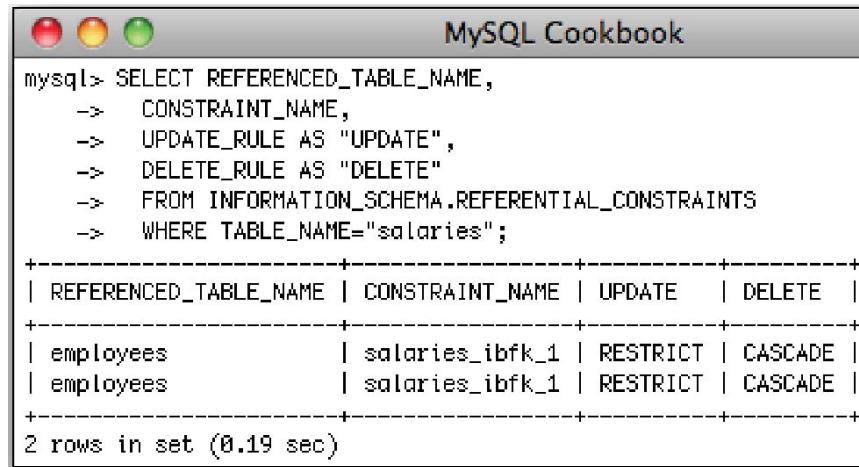
TABLE_NAME	CONSTRAINT_NAME	UPDATE	DELETE
dept_emp	dept_emp_ibfk_1	RESTRICT	CASCADE
dept_manager	dept_manager_ibfk_1	RESTRICT	CASCADE
salaries	salaries_ibfk_1	RESTRICT	CASCADE
titles	titles_ibfk_1	RESTRICT	CASCADE

At the bottom of the terminal window, it says "4 rows in set (0.30 sec)".

## Monitoring and Analyzing MySQL Installation

3. Use the following query to retrieve information about the tables reference salaries table:

```
mysql> SELECT REFERENCED_TABLE_NAME,
->      CONSTRAINT_NAME,
->      UPDATE_RULE AS "UPDATE",
->      DELETE_RULE AS "DELETE"
->      FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
->      WHERE TABLE_NAME="salaries";
```



The screenshot shows a MySQL command-line interface window titled "MySQL Cookbook". The query executed is:

```
mysql> SELECT REFERENCED_TABLE_NAME,
->      CONSTRAINT_NAME,
->      UPDATE_RULE AS "UPDATE",
->      DELETE_RULE AS "DELETE"
->      FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS
->      WHERE TABLE_NAME="salaries";
```

The output displays two rows of data from the REFERENTIAL\_CONSTRAINTS table:

REFERENCED_TABLE_NAME	CONSTRAINT_NAME	UPDATE	DELETE
employees	salaries_ibfk_1	RESTRICT	CASCADE
employees	salaries_ibfk_1	RESTRICT	CASCADE

2 rows in set (0.19 sec)

### How it works...

This recipe uses the information available in the REFERENTIAL\_CONSTRAINTS table in the INFORMATION\_SCHEMA schema. The information retrieved in step 2 means that updating information in the employees table might become difficult, as it is referenced by other tables, and an ON UPDATE RESTRICT policy is in place. Deletion of data, however, will be easy, as the deletion rules for all four dependencies are marked as ON DELETE CASCADE. Because of this, you should be even more careful, however, not to delete the wrong data.

Especially, the information delivered by step 2 is often very important to get an overview of the dependencies of large data models that evolved over a longer period of time and involved many people involved.

### See also

- ▶ [Creating a read-only account](#)

# Configuring MySQL

In this chapter, we will shed some light on MySQL configuration settings. There is a multitude of dials and knobs available for both the MySQL server and the different storage engines underneath it. Covering them all would by far exceed the limitations of any book. Instead, we will just go for a selection of some of the most important ones:

- ▶ Setting up a fixed InnoDB tablespace
- ▶ Setting up an auto-extending InnoDB tablespace
- ▶ Storing InnoDB data in one file per table
- ▶ Decreasing InnoDB tablespace
- ▶ Enabling and configuring binary logging
- ▶ Configuring the InnoDB redo log
- ▶ Understanding and configuring important MySQL and InnoDB timeout options
- ▶ Adjusting table and database name letter case handling for better platform independence
- ▶ Installing MySQL as a Windows service with custom options
- ▶ Running multiple MySQL server instances in parallel on a Linux server
- ▶ Preventing invalid date values from being stored in DATE or DATETIME columns

## Configuring MySQL

# Introduction

Installing a MySQL server is one thing that is fairly easy to do. Setting up the vast array of configuration options available can be far more challenging, as there is no one-size-fits-all combination of settings. MySQL comes with some presets that can serve as a starting point for different server sizes. They are called `my-large.cnf`, `my-huge.cnf`, `my-medium.cnf` and `my-innodb-heavy-4G.cnf`. On Ubuntu Linux they can be found in compressed form in the `/usr/share/doc/mysql-server-5.0/examples` directory. On Windows, similar examples exist but have file names ending in `.ini` and are located in the MySQL program directory.

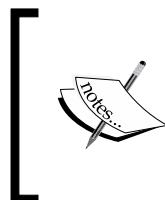
In this chapter, we will delve into a selection of the most relevant configuration options, focusing on important InnoDB settings, as this is the storage engine most relevant for enterprise applications.



Please refer to the appendix for a description of how to set up the `innodb_buffer_pool_size` setting and make the best use of the available memory.

## Setting up a fixed InnoDB tablespace

When using the InnoDB storage engine of MySQL, the data is typically not stored in a per-database or per-table directory structure, but in several dedicated files, which contain the so-called **tablespace**. By default (when installing MySQL using the community wizard) InnoDB is configured to have one small file to store data in, and this file grows as needed. While this is a very flexible and economical configuration to start with, this also has some drawbacks: there is no reserved space for your data, so you have to free disk space every time your data grows. Also, if your database grows bigger, the files grow to a size which makes it hard to handle—a dozen files of 1 GB each are typically easier to manage than one clumsy 12 GB file.



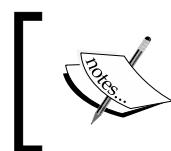
Large data files might, for example, cause problems if you try to put those files into an archive for backup or data transmission purposes. Even if the 2 GB limit is not present any more for the current file systems, many compression programs still have problems dealing with large files.

And finally, the constant adaptation of the file in InnoDB's default configuration size can result in a (small, but existent) performance hit if your database grows.

The following recipe will show you how to define a fixed tablespace for your InnoDB database.

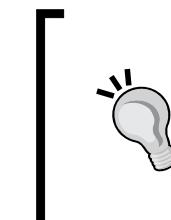
## Getting ready

To install a fixed tablespace, you will have to reflect about some aspects: how much tablespace should be reserved for your database, and how to size the single data files in sum constitute the tablespace.



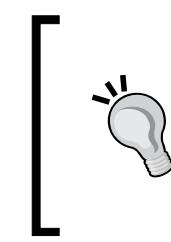
Note that once your database completely allocates your tablespace, you will run into *table full* errors (error code 1114) when trying to add new data to your database.

Additionally, you have to make sure that your current InnoDB tablespace is completely empty. Ideally, you should set up the tablespace of a freshly installed MySQL instance, in which this prerequisite is given.



To check whether any InnoDB tables exist in your database, execute the following statement and delete the given tables until the result is empty:

```
SELECT TABLE_SCHEMA, TABLE_NAME FROM information_schema.tables WHERE engine="InnoDB";
```



If your database already contains data stored in InnoDB tables that you do not want to lose, you will have to create a backup of your database and recover the data from it when you are done with the recipe. Please refer to the chapter *Backing Up and Restoring MySQL Data* for further information on this.

And finally, you have to make sure that the InnoDB data directory (as defined by the `innodb_data_home_dir` variable) features sufficient free disk space to store the InnoDB data.

For the following example, we will use a fixed tablespace with a size of 500 MB and a file size of 200 MB.

## How to do it...

1. Open the MySQL configuration file (`my.ini` or `my.cnf`) in a text editor.
2. Identify the line starting with `innodb_data_file_path` in the `[mysqld]` section. If no such line exists, add the line to the file.
3. Change the line `innodb_data_file_path` to read as follows:

## Configuring MySQL

5. Shut down your database instance (if running).
6. Delete previous InnoDB data files (typically called `ibdata1`, `ibdata2`, and so forth) from the directory defined by the `innodb_data_home_dir` variable.
7. Delete previous InnoDB logfiles (named `ib_logfile0`, `ib_logfile1`, and so forth) from the directory defined by the `innodb_log_group_home_dir` variable.



If `innodb_log_group_home_dir` is not configured explicitly, it defaults to the `datadir` directory.

8. Start your database.
9. Wait for all data and log files to be created.



Depending on the size of your tablespace and the speed of your disk system, creation of InnoDB data files can take a significant amount of time (several minutes is not an uncommon time for larger installations). During this initialization sequence, MySQL is started but it will not accept any requests.

## How it works...

Steps 1 through 4—and particularly 3—cover the actual change to be made to the configuration, which is necessary to adapt the InnoDB tablespace settings. The value of the `innodb_data_file_path` variable consists of a list of data file definitions that are separated by semicolons. Each data file definition is constructed of a file name and a size with a colon as a separator. The size can be expressed as a plain numeric value that defines the size of the data file in bytes. If the numeric value has a K, M, or G postfix, the number is interpreted as Kilobytes, Megabytes, or Gigabytes respectively. The list is limited to the three entries of our example; if you want to split a large tablespace into small files, the list can easily contain dozens of data file definitions.



If your tablespace consists of more than 10 files, we propose naming the first nine files `ibdata01` through `ibdata09` (instead of `ibdata1` and so forth; note the zero), so that the files are listed in a more consistent order when they are displayed in your file browser or command line interface.

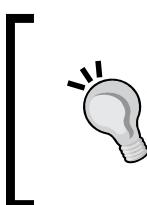
Step 5 is prerequisite to the steps following after it, as deletion of vital InnoDB files while the system is still running is obviously not a good idea. In step 6, old data files are deleted to prevent collision with the new files. If InnoDB detects an existing file whose size differs from the new file, it will ignore the new file.



Note that deletion of the InnoDB data files is only sufficient if all InnoDB tables were deleted previously (as discussed in the *Getting ready* section).

Alternatively, you could delete all \*.frm files for InnoDB tables from the MySQL data directory, but we do not encourage this approach (clean deletion using `DROP TABLE` statements should be preferred over manual intervention in MySQL data directories whenever possible).

Step 7 is necessary to prevent InnoDB errors after the data files are created, as the InnoDB engine refuses to start if the log files are older than the tablespace files. With step 7 completed, the new settings take effect.



When starting the database for the first time after changes being made to the InnoDB tablespace configuration, take a look at the MySQL error log to make sure the settings were accepted and no errors have occurred.

The MySQL error log after the first start with the new settings will look similar to the following:

```
InnoDB: The first specified data file E:\MySQL\InnoDBTest\ibdata1
      not exist!
InnoDB: a new database to be created!
091115 21:35:56  InnoDB: Setting file E:\MySQL\InnoDBTest\ibdata1
      to 200 MB
InnoDB: Database physically writes the file full: wait...
InnoDB: Progress in MB: 100 200
...
InnoDB: Progress in MB: 100
091115 21:36:19  InnoDB: Log file .\ib_logfile0 did not exist,
      be created
InnoDB: Setting log file .\ib_logfile0 size to 24 MB
InnoDB: Database physically writes the file full: wait...
...
InnoDB: Doublewrite buffer not found: creating new
InnoDB: Doublewrite buffer created
InnoDB: Creating foreign key constraint system tables
InnoDB: Foreign key constraint system tables created
091115 21:36:22  InnoDB: Started; log sequence number 0 0
091115 21:36:22 [Note] C:\Program Files\MySQL\MySQL Server 5.1\mysqld: ready for connections.
Version: '5.1.31-community-log' socket: '' port: 3306  MySQL Community Server (GPL)
```

## Configuring MySQL

### There's more...

If you already use a fixed tablespace, and you want to increase the available space, you can simply append additional files to your fixed tablespace by adding additional database definitions to the current `innodb_data_file_path` variable setting. If you simply add additional files, you do not have to empty your tablespace first, but you can change the configuration and simply restart your database. Nevertheless, as with all changes to the configuration, we strongly encourage creating a backup of your database first.

### See also

- ▶ *Backing Up and Restoring MySQL Data*
- ▶ *Estimating tablespace requirements*

## Setting up an auto-extending InnoDB tablespace

The previous recipe demonstrates how to define a tablespace with a certain fixed size. While this provides maximum control and predictability, you have to block disk space based on an estimate of the maximum size required in the foreseeable future. As long as you store less data in your database than the reserved tablespace allows for, this basically means that disk space is wasted. This especially holds true if your setting does not allow for a separate file system exclusively for your MySQL instance, because then other applications compete for disk space as well. In these cases, a dynamic tablespace that starts with little space and grows as needed could be an alternative. The following recipe will show you how to achieve this.

### Getting ready

When defining an auto-extending tablespace, you should first have an idea about the minimum tablespace requirements of your database, which will set the initial size of the tablespace. Furthermore, you have to decide whether you want to split your initial tablespace into files of a certain maximum size (for better file handling).

If the above settings are identical to the current settings and you only want to make your tablespace grow automatically if necessary, you will be able to keep your data. Otherwise, you will have to empty your current InnoDB tablespace completely (please refer to the previous recipe *Setting up a fixed InnoDB tablespace* for details).

As with all major configuration changes to your database, we strongly advise you to take a backup of your data first. If you have to empty your tablespace, you can use this backup to recover your data after the changes are completed. Again, please refer to the chapter *Backing Up and Restoring MySQL Data* for details.

---

And as before, you have to make sure that there is enough disk space available in `innodb_data_home_dir` directory—not only for the initial database size, but also anticipated growth of your database.

The recipe also requires you to shut down your database temporarily; so you have to ensure all clients are disconnected while performing the required steps to prevent conflicting operations.

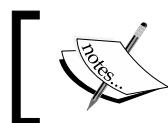
As the recipe demands changes to your MySQL configuration file (`my.cnf` or `my.ini`), you need write access to this file.

For the following example, we will use an auto-extending tablespace with an initial size of 50 MB and a file size of 50 MB.

## How to do it...

1. Open the MySQL configuration file (`my.ini` or `my.cnf`) in a text editor.
2. Identify the line starting with `innodb_data_file_path` in the `[mysqld]` section. If no such line exists, add the line to the file.
3. Change the line `innodb_data_file_path` to read as follows:

```
innodb_data_file_path=ibdata1:50M;ibdata2:50M:autoextend
```



Note that no file definition except the last one must have the `:autoextend` option; you will run into errors otherwise.

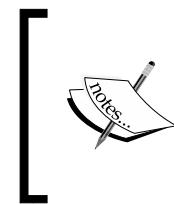
4. Save the changed configuration file.
5. Shut down your database instance (if running).
6. Delete previous InnoDB data files (typically called `ibdata1`, `ibdata2`, and so on) from the directory defined by the `innodb_data_home_dir` variable.
7. Delete previous InnoDB logfiles (named `ib_logfile0`, `ib_logfile1`, and so on) from the directory defined by the `innodb_log_group_home_dir` variable.



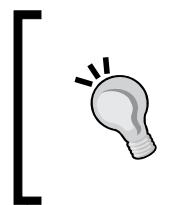
If `innodb_log_group_home_dir` is not configured explicitly, it defaults to the `datadir` directory.

8. Start your database.
9. Wait for all data and log files to be created.

## Configuring MySQL



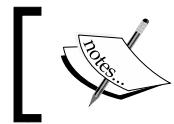
Depending on the size of your tablespace and the speed of your disk system, creation of InnoDB data files can take a significant amount of time (several minutes is not an uncommon time for larger installations). During this initialization sequence, MySQL is started but will not accept any requests.



When starting the database for the first time after changes being made to the InnoDB tablespace configuration, take a look at the MySQL error log to make sure the settings were accepted and no errors have occurred.

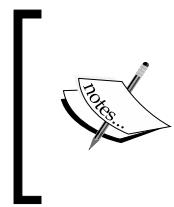
## How it works...

The above steps are basically identical to the steps of the previous recipe *Setting up an InnoDB tablespace*, the only difference being the definition of the `innodb_data_file_path` variable. In this recipe, we create two files of 50 MB size, the last one having an `:autoextend` property.



If the `innodb_data_file_path` variable is not set explicitly, it defaults to the value `ibdata1:10M:autoextend`.

As data gets inserted into the database, parts of the tablespace will be allocated. Once the 100 MB of initial tablespace is not sufficient any more, the file `ibdata2` will grow larger to match the additional tablespace requirements.



Note that the `:autoextend` option causes the tablespace files to be extended automatically, but they are not automatically reduced in size if the space requirements decrease. Please refer to the *Decreasing InnoDB tablespace* recipe for instructions on how to free unused tablespace.

## There's more...

The recipe only covers the basic aspects of auto-extending tablespaces; the following links provide insight into some more advanced topics.

### Making an existing tablespace auto-extensible

If you already have a database with live data in place and you want to change your

---

Let us assume a current configuration like the following:

```
innodb_data_file_path=ibdata1:50M;ibdata2:50M
```

The respective configuration with auto-extension will look like this:

```
innodb_data_file_path=ibdata1:50M;ibdata2:50M:autoextend
```

In this case, do not empty the InnoDB tablespace first, you can simply change the configuration file and restart your database, and you should be fine. As with all configuration changes, however, we strongly recommend to back up your database before editing settings even in this case.

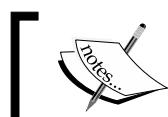
## Controlling the steps of tablespace extension

The amount by which the size of the auto-extending tablespace file is increased is controlled by the `innodb_autoextend_increment` variable. The value of this variable defines the number of Megabytes by which the tablespace is enlarged. By default, 8 MB are added to the file if the current tablespace is no longer sufficient.

## Limiting the size of an auto-extending tablespace

If you want to use an auto-extending tablespace, but also want to limit the maximum size that the tablespace will grow to, you can add a maximum size for the auto-extended tablespace by using the `:autoextend:max:[size]` option. The `[size]` portion is a placeholder for a size definition using the same notation as the size description for the tablespace file, which means a numeric value and an optional K, M, or G modifier (for sizes in Kilobytes, Megabytes, or Gigabytes). As an example, if you want to have a tiny initial tablespace of 10 MB, which will be extended as needed, but with an upper limit of 2 GB, you would enter the following line in your MySQL configuration file:

```
innodb_data_file_path=ibdata1:10M:autoextend:max:2G
```



Note that if the maximum size is reached, you will run into errors when trying to add new data to your database.

## Adding a new auto-extending data file

Imagine an auto-extending tablespace with an auto-extended file, which grew so large that you want to prevent the file from growing further and want to append a new auto-extended data file to the tablespace. You can do so using the following steps:

1. Shut down your database instance.
2. Look up the exact size of the auto-extended InnoDB data file (the last file in the current configuration).

## Configuring MySQL

3. Put the exact size as the tablespace file size definition into the `innodb_data_file_path` configuration (number of bytes without any K, M, or G modifier), and add a new auto-extending data file.
4. Restart your database.

As an example, if your current configuration reads `ibdata1:10M:autoextend` and the `ibdata1` file has an actual size of 44,040,192 bytes, change configuration to `innodb_data_file_path=ibdata1:44040192;ibdata2:10M:autoextend`.

### See also

- ▶ *Backing Up and Restoring MySQL Data*
- ▶ *Estimating tablespace requirements*
- ▶ *Setting up a fixed InnoDB tablespace*
- ▶ *Decreasing InnoDB tablespace*

## Storing InnoDB data in one file per table

In the previous recipes, we presented a way to define a common tablespace in which the InnoDB storage engine stores all data for InnoDB tables. While this has some advantages (for example, dynamic reuse of free space across tables), this approach is completely different from the MyISAM technique that stores the data for each table in a separate file. The following recipe will show you how to configure InnoDB to store data in separate files, one for each table.

### Getting ready

The following steps include a database downtime, so you have to prepare a maintenance window for your database that allows you to complete the steps without interfering with still accessing the database.

### How to do it...

1. Create a SQL dump of your entire database.
2. Shut down your database.
3. Open the MySQL configuration file (`my.ini` or `my.cnf`) in a text editor.

- 
4. Add the following line to the [mysqld] section:

`innodb_file_per_table`

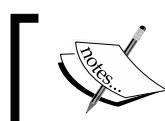
5. Save the changed configuration file.
6. Start your database instance.
7. Recover all data from the SQL dump created in step 1.

## How it works...

This recipe is pretty straightforward: create a backup, add the `innodb_file_per_table` option to the [mysqld] configuration, and recover data from the backup (please refer to chapter *Backing Up and Restoring MySQL Data* for details). But why the recovery?

After the restart of the database in step 6, you will not notice any changes to the data. At this point, all data is still stored in the InnoDB tablespace. This is because the `innodb_file_per_table` setting only applies to newly created tables! Step 7 takes care of this: during recovery, each table is dropped and created again (this time in a separate file) before the original data is inserted.

The `innodb_file_per_table` setting allows you to use the operating system's file mapping to map tables to different physical disks or to back up and recover certain tables on their own.



However, note that with this approach, each table has its own auto-extending tablespace—extension yes, but no reduction!

If you delete data from a table, the file size will not reduce automatically. As each table uses its own tablespace, this means that the space that gets freed by deleting from one table cannot be reused by another table, as is the case for the "classical" shared InnoDB tablespace. There is a way to free unused space again: please refer to the following chapter *Decreasing InnoDB tablespace* for instructions on how to do this.

## See also

- ▶ *Backing Up and Restoring MySQL Data*
- ▶ *Decreasing InnoDB tablespace*

*Configuring MySQL*

---

## Decreasing InnoDB tablespace

The previous recipes enable you to define a tablespace; and for the autoextend `innodb_file_per_table` approaches, the tablespace will grow until you run out of space (or you hit a file size boundary set by your operating system). The opposite operation, however, does not work as easily: InnoDB lacks an automatic shrink option if you have deleted data from your database. The following recipe will introduce ways to reduce the size of your InnoDB tablespace if your storage requirements decrease.

### Getting ready

As the following recipe involves creation of a backup dump of your database, you have to reserve enough disk space to temporarily store the dump file. Furthermore, if you want to establish a shared tablespace with a reduced size, you have to decide about the size of the tablespace and how to split it into separate files. For a fixed size tablespace, you additionally have to calculate the space requirements of your current data (please refer to the *Estimating tablespace requirements* recipe in Chapter 6 for details) to decide about the new size of the fixed tablespace.

The recipe also requires you to shut down your database temporarily, so you have to make sure you have a maintenance interval in which the database is not accessed by anyone.

### How to do it...

1. Create a full backup of your database (for further instructions please see the *Backing Up and Restoring MySQL Data* chapter).
2. Drop all databases (except for the `mysql` and `information_schema` databases).
3. Shut down your database.
4. Open the MySQL configuration file (`my.ini` or `my.cnf`) using a text editor.
5. Configure the InnoDB tablespace according to your new reduced space requirements (please refer to the previous recipes for further details).
6. Save the changed configuration file.
7. Delete the old InnoDB data files (typically called `ibdata1`, `ibdata2`, and `ibdata3`) from the directory defined by the `innodb_data_home_dir` variable.
8. Delete previous InnoDB logfiles (named `ib_logfile0`, `ib_logfile1`, and `ib_logfile2`) from the directory defined by the `innodb_log_group_home_dir` variable.



If `innodb_log_group_home_dir` is not configured explicitly, it defaults to the `datadir` directory.

9. Start your database instance.
10. Wait for all data and log files to be created.
11. Recover all data from the SQL dump created in step 1.

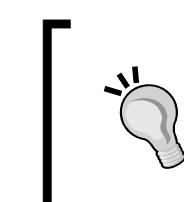
## How it works...

This recipe is basically a slight modification of the previous recipes to set up a shared tablespace. The additional steps are backing up your database and subsequently dropping databases initially, and the recovery from the backup as the final step. This ensures that the directory entries for all InnoDB tables are removed flawlessly, and (in the case of the `innodb_file_per_table` table option) all storage files are removed. The shared tablespace is created whenever you start the MySQL instance after the configuration changes. If the `innodb_file_per_table` option is used, the table files are created during the data recovery. In case of an auto-extending tablespace, the restoration of the data might cause the tablespace to increase in size only to the size actually needed by the current data.

As this approach might require a significant downtime for any client, we propose performing a tablespace reduction only if absolutely necessary—with thorough planning of your requirements, your disk structure, and your tablespace sizing, you hopefully will not need to resize your tablespace on a regular basis.

## There's more...

If you use the `innodb_file_per_table` feature and you want to reduce the size of the separate files to the currently needed size, you have an alternative that does not involve a full dump, deletion, and recovery procedure:

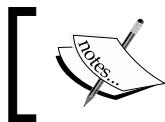


To free unused tablespace with `innodb_file_per_table` in place, you can execute an empty `ALTER TABLE` statement:

```
ALTER TABLE example_table ENGINE=InnoDB;
```

You have to execute this command for each InnoDB table whose storage file you want to resize. However, note that this temporarily creates a copy of the whole table and locks the table during the process.

## Configuring MySQL



Due to potential locking conflicts, try to avoid freeing unused tablespaces (using the `ALTER TABLE` command) with large and/or heavily used tables in a running system.

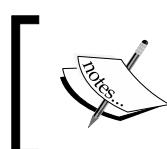
### See also

- ▶ *Backing Up and Restoring MySQL Data*
- ▶ *Estimating tablespace requirement*
- ▶ *Setting up a fixed InnoDB tablespace*
- ▶ *Setting up an auto-extending InnoDB tablespace*
- ▶ *Storing InnoDB data in one file per table*

## Enabling and configuring binary logging

Binary logging (or binlogging for short), describes a feature of MySQL that will write a transcript of all statements issued that actually modified or could have modified data. This includes `UPDATE`, `INSERT`, and `DELETE` statements, regardless of whether they affected any rows on the server, as well as data definition language statements (`CREATE TABLE`, `DROP TABLE`, and the like).

This protocol is written in a special format that contains metadata about transactions, settings, and more, which makes it suitable as a basis for both replication, backup, and change auditing, with the former two being the most important ones.



Generally, we do not recommend running MySQL without binlogging, as the performance penalty is very low—the MySQL manual speaks of a speed degradation of about 1%—and the benefits clearly outweigh this.

You can find detailed information about the binary log in section 5.2.4 *The Binary Log* in the online manual at <http://dev.mysql.com/doc/refman/5.1/en/binary-logs.html>.

In this recipe, we will show you how to make sure your MySQL servers are configured to use binary logs in the first place and also keep them maintainable in terms of file sizes.

### Getting ready

As the binary log setup is a part of the server configuration, you will need an operating system user account and sufficient rights to modify the server's configuration file. It is recommended to have sufficient rights to restart the MySQL service because this

## How to do it...

1. Open the MySQL configuration file, typically `my.cnf` or `my.ini` (on Windows).
2. Locate the `[mysqld]` section in the file.
3. Add the following settings or edit them if they are already present. Adapt the path after `log_bin` and fill in a path valid on your server. Substitute your server's name for `HOSTNAME`:

```
log_bin=/var/log/mysql/HOSTNAME-bin
expire_logs_days=10
max_binlog_size=200M
```
4. Save the file.
5. Restart the MySQL service.
6. Check the directory you specified for `log-bin`. You should see a file called `HOSTNAME-bin.000001` there and a corresponding `HOSTNAME-bin.index`.

## How it works...

The first parameter `log_bin` tells the server which directory is intended to store the binlogs. This should be a storage volume with sufficient space and ideally on a different disk than the data directory for better performance. The amount of disk space required depends on the amount of write access to your databases.



All databases of one server share the same binlog files, meaning you have to consider this in your space estimation.

The `expire_logs_days` setting is meant to prevent excessive disk space usage of binlogs. A setting of 10 means that any binary log file older than 10 days will be deleted the next time a new binary log is started. You can use the `PURGE BINARY LOGS` command to force a new binary log to be started manually.

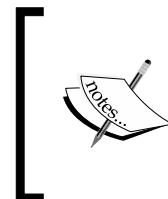
`max_binlog_size` is meant to keep the binlog files manageable by automatically truncating and rotating them once they exceed the configured size threshold. Please note that the size might in fact become slightly larger than the configured limit because transactions are written as a whole. This means that when a binlog file has almost reached its size limit and a large transaction is then committed, it will still be written to that file, pushing it past the configured threshold.

## Configuring MySQL

### There's more...

Binary logs are very important for point-in-time backup and recovery. Make sure you set the `expire_logs_days` to a value that is large enough to span the time between backups. Otherwise there would be a chance that you cannot do a complete disaster recovery when the binlogs have already been deleted before you have taken the next full backup. To completely switch off the automatic deletion of old binlog files, set the `expire_logs_days` parameter to 0 or remove it from the configuration file.

Even though it is possible, it is not recommended to delete older binlogs manually from the file system. Depending on the server version you are using, MySQL might fail to start if the index file and the binlogs actually present do not match.



To prevent problems of this kind, we recommend always using the `PURGE BINARY LOGS` command. For details on its options, see the MySQL documentation manual at <http://dev.mysql.com/doc/refman/5.1/en/purge-binary-logs.html>.

## Configuring the InnoDB redo log

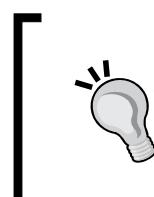
In order to prevent the transactional nature of InnoDB from completely thwarting its performance, it implements what is called the **redo log**.

In this recipe, we will present the relevant settings to (re-)configure a database server's redo log.

### Getting ready

As the redo log setup is a part of the server configuration, you will need an operating user account and sufficient rights to modify the server's configuration file. You will also need rights to restart the MySQL service because the redo log cannot be reconfigured once it has been initialized.

Moreover, an administrative MySQL user account is required to prepare the server for shutdown, necessary as part of the procedure.



#### Caution:

As this recipe will modify the configuration of parameters critical to data integrity, you should make a backup copy of the configuration file before editing it!

## How to do it...

1. Connect to the server using your administrative account.
2. Issue the following command:

```
mysql> SET GLOBAL innodb_fast_shutdown=0;
Query OK, 0 rows affected (0.00 sec)
```

3. Verify the setting like this:

```
mysql> SHOW VARIABLES LIKE 'innodb_fast_shutdown';
```

Variable_name	Value
innodb_fast_shutdown	0

4. Log off from MySQL and stop the MySQL server.
5. Locate the MySQL configuration file, usually called `my.cnf` or `my.ini` (or `my.cfg`) and open it in a text editor.
6. Locate the following parameters in the `[mysqld]` section (you values will of course):
 

```
[mysqld]
...
innodb_log_group_home_dir=/var/lib/mysql/redolog
innodb_log_file_size=32M
innodb_log_buffer_size=64M
innodb_log_files_in_group=2
...
```
7. Edit the above configuration settings to their new values. If you require help to find suitable values, see the *There's more...* section of this recipe.
8. Save the configuration file.
9. Navigate to the directory configured for `innodb_log_group_home_dir`. If there is no such setting in your configuration file, navigate to MySQL's data directory and then take it as the default.

## Configuring MySQL

10. Move the files whose names start with `ib_logfile` to a backup location; copy them; they must be removed from their original location.
11. Restart the MySQL server.
12. Verify that new files are created as you configured them:

```
$ ls -l /var/lib/mysql/redo
```

```
ds@ubuntu: /var/lib/mysql/redo$ ls -l
total 65536
-rw-r----- 1 root root 33554432 2009-11-06 22:11 ib_logfile0
-rw-r----- 1 root root 33554432 2009-11-06 22:11 ib_logfile1
ds@ubuntu:/var/lib/mysql/redo$
```

If you do not see the new files appear and the server does not start up correctly, check the MySQL error log for messages. Usually, the only thing that can go wrong here is that either mistyped the directory name or did not actually remove the previous `ib_logfile`

To restore everything back to the original configuration, restore your configuration file from backup and restore the `ib_logfile` files you moved out to the backup to their original locations.

## What just happened...

By setting `innodb_fast_shutdown` to 0, you told the server to finish writing any changes to the disk before actually exiting. This makes sure there are no remaining transactions in the current redo logs that could get lost when these files are replaced.

After that you could change the configuration to new values, possibly using a different number of files and different sizes.

Then, before restarting, you could move the old redo log files out of the way. This is because otherwise MySQL would complain about a mismatch between the setting and the actual situation on disk. When it comes up finding no redo log files, it will create new ones with the settings just configured.

## There's more...

Often when talking about transactions, the word **rollback** comes up. It means that if something goes wrong in the middle of a possibly complex data manipulation operation and it has to be aborted, the database server will safely restore everything back to the state it was in when the operation began, not leaving any data only partially deleted or modified.

The opposite term—**rollforward** or **redoing**—is less commonly used. It means that when the complex operation mentioned earlier completes successfully, you are guaranteed that nothing short of actual hardware failure could lead to these changes being lost again. This might appear obvious because one would expect the database server not to mark anything as successful unless it was actually completely done. However, if that were the case, operations would become painfully slow, as the underlying I/O subsystem (generally hard disks) is very often the bottleneck component.

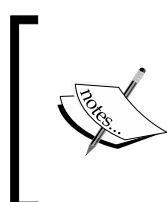
To evade this potential performance problem, most transactional databases, InnoDB included, no different, uses the concept of a transaction—or redo—log that allows it to more efficiently handle write operations without risking data integrity. The redo log works as a sort of log buffer or pad, containing information on what remains to be done to the data files. With its knowledge of the redo log, the database server can optimize disk access to improve performance.

The ideal size for the redo log depends on the size and number of transactions the database needs to process. Generally speaking, the log should be large enough to store any single transaction plus about 10 percent. As a rule of thumb, the total log size (the number of log files times their individual size) need not exceed about 50% of the InnoDB buffer pool size.

For more information on redo logs and how to determine a sensible size setting, visit <http://mysqldump.azundris.com/archives/78-Configuring-InnoDB-Redo-Logs-and-Log-Sizing-InnoDB-tutorial.html> for a detailed description of InnoDB log configuration.

## Understanding and configuring important MySQL and InnoDB timeout options

MySQL's configuration file can contain a variety of different timeout settings, each designed for a specific kind of operation or connection. In this recipe, we present a selection of timeout settings and a suggested value to go along with each. The *How it works...* section provides details on each value presented.



In general, the values suggested here should be appropriate for both MySQL versions 5.0 and 5.1. However, please note that any of these options may well vary for your environment, depending on what the requirements are. Please do not simply use these values verbatim.

### Getting ready

To apply timeout configuration settings, you will need access to the MySQL configuration file—typically `my.cnf` or `my.ini` (on Windows)—and the rights to restart the server so that any changes made to the configuration take effect.

*Configuring MySQL***How to do it...**

1. Locate the MySQL configuration file and open it in a text editor.
2. In the [mysqld] section, set up the following values. Some of the options may already be present; others will likely have to be added. Make sure each option appears once.

```
[mysqld]
.. in-
nodb_rollback_on_timeout
innodb_lock_wait_timeout=50

interactive_timeout=1200
wait_timeout=28800

net_read_timeout=30
net_write_timeout=120
...
```

3. Save the file.
4. Restart the MySQL server.

**How it works...**

By setting the values as described in the above section, you tell MySQL to use different settings than the defaults for the options mentioned. The new settings take effect with the next server restart.

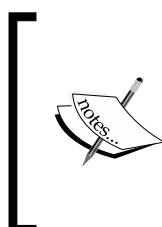
### **Setting values for innodb\_rollback\_on\_timeout / innodb\_lock\_timeout**

These two settings are probably the most important as regards the locking behavior of your MySQL setup. Starting with version 5.0.13, MySQL changed the rollback behavior when a timeout occurred because a transaction could not acquire a lock for a row. This used to happen when another transaction is still working on that row, and this is to be expected in normal database operations. The database server rolls back the entire transaction in this case. Applications should be designed to respond to such conditions by retrying the entire transaction.

Prior to 5.0.13, this was the default behavior, maintaining the rule that a transaction succeeds or fails and is rolled back completely as an *atomic entity*. In 5.0.13 and later versions, the default was modified to roll back only the very last statement of the transaction instead, keeping the transaction open. While there is a reason behind this change,

---

level to be able to handle this very MySQL-specific scenario.



Setting the `innodb_rollback_on_timeout` option in the configuration file restores the more standard way of rolling back the whole transaction in case of a lock wait timeout. We strongly recommend enabling this option unless you are perfectly sure your application is aware of the MySQL-specific behavior configured as the default.

`innodb_lock_wait_timeout` defines the number of seconds a transaction will acquire a necessary lock when a second transaction is working concurrently on the same records. The default value is 50 seconds and if the lock could not be acquired by the time the timeout error will occur and the transaction will be rolled back. Depending on how long transactions in your setup typically take, this value often needs to be adjusted. If you have bulk data operations that affect a lot of rows, you will want to increase this value; on the other hand your system normally uses very short transactions, reducing this value can help you find out about problems with lock contention earlier.

## Setting values for `interactive_timeout` / `wait_timeout`

`interactive_timeout` defines how long an interactive client connection can be idle before the server closes it automatically. 'Idle' in this context refers to the time between statements being executed with no activity in between.

We recommend reducing this from the default value of 28,800 seconds (8 hours) to a lower value like 1,200 seconds (20 minutes). This allows the server to close idle connections and conserve some resources.

The counterpart variable for non-interactive sessions, such as those from an application's server's connection pool, is called `wait_timeout` and has the same semantics. If you rely on your application, you might want to leave this setting on a higher value as most connection pools can be configured to release connections automatically depending on current load conditions.

## Setting values for `net_read_timeout` / `net_write_timeout`

The protocol MySQL uses to handle communication between server and clients is limited in design, allowing only one operation to be carried out at a time. A side effect of this is that once a data transfer in either direction has started, there is no way for it to be interrupted in a controlled manner.

The `net_read_timeout` controls how long a piece of information can be sent from the client to the server, before the connection is aborted. This is usually not a problem—the default setting is 30 seconds. Under no regular circumstances will a communication in that

## Configuring MySQL

---

The `net_write_timeout` is more problematic because for large result sets, the value of 60 seconds might be too short. This is especially true for clients that fetch in streaming mode, potentially performing time-consuming operations on each row thereby making the overall operation take longer than the timeout.

The exact value required for your setup depends on how clients fetch data and what you will need to experiment and find a suitable value.

One caveat to consider with `net_write_timeout` is that it may lead to seemingly failures of `mysqldump` like this:

```
mysqldump: Error 2013: Lost connection to MySQL server during
when dumping table `tablename` at row: 935578
```

This can happen if the following conditions apply:

- ▶ `net_write_timeout` is set to a low value
- ▶ `max_allowed_packet` is set to a large value

Depending on the speed of the network over which `mysqldump` has connected and the rows being dumped, it may be necessary to increase `net_write_timeout` by at least to as long as it takes to transmit `max_allowed_packet` bytes over the network to write it to the output.

`mysqldump` is a regular client program and subject to the `net_write_timeout` setting. When the server sends rows to be dumped to `mysqldump` in chunks of up to `max_allowed_packet` bytes, depending on the network connection in between this might take longer than `net_write_timeout` allows, making the server cut the connection even though nothing is really wrong. Increasing `net_write_timeout` for the `mysqldump` tool's session would remedy this, but unfortunately as of the time of writing there is no such setting for `mysqldump`. A workaround, if you encounter this problem, is to temporarily increase the global server `net_write_timeout` via:

```
$ mysql -uroot -e "SELECT @@GLOBAL.net_write_timeout AS oldvalue;
  SET GLOBAL net_write_timeout=600;""
$ mysqldump ...
$ mysql -e "SET GLOBAL net_write_timeout=oldvalue;"
```

The first command will display the current value for `net_write_timeout` and then set it to 600 seconds (you can change this to 10 minutes. After that the `mysqldump` can take place. Finally, the old value is restored (make sure you fill in the correct old value).

See MySQL Bug #46103 at <http://bugs.mysql.com/bug.php?id=46103> for more details.

# Adjusting table and database name letter case handling for better platform independence

MySQL is available for a variety of platforms—the major ones being Windows and Linux. Although data files are compatible and can be transferred between platforms, and MySQL mostly follows the same general principles, there is an important caveat to know about how different operating systems handle file names.

In this recipe, we will show you how to set up MySQL in a way such that it is much less likely to run into problems when moving data files between platforms. Because MySQL database tables correlate to file system objects (directories and files), differences in how the operating system (or rather the file system) handles file and directory names can lead to unexpected effects, especially when working in heterogeneous environments.



We generally recommend setting up all your MySQL servers as described in this recipe to prevent any problems.

## Getting ready

You will need an operating system user account and sufficient rights to modify the MySQL configuration file. You will also need rights to restart the MySQL service because the case handling cannot be reconfigured on the fly.



Please note that for best results this setting should be applied before you start creating databases and tables on that server.

## How to do it...

1. Make sure MySQL is not running.
2. Locate the MySQL configuration file, usually called `my.cnf` or `my.ini` (on Windows) and open it in a text editor.
3. Locate the following parameter in the `[mysqld]` section. If it is not there, otherwise edit it to match the value shown here:

```
[mysqld]
```

...

## Configuring MySQL

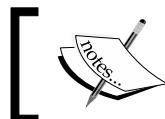
4. Save the configuration file.
5. Restart the MySQL server.

### How it works...

MySQL table and database names are mapped to file system names. Most Unix-like file systems are case sensitive, meaning that the two files TableA and tablea are from each other. On Windows, these two names will refer to the very same file.

Setting the `lower_case_table_names` configuration to 1 tells MySQL to always convert any database or table names to lowercase letters, both when creating and using them. This will ensure that no matter what casing any SQL statements use, they will affect the same tables.

This is especially useful in replication scenarios where you replicate between master and slave machines using different operating systems.



Manually configuring this setting is highly recommended because depending on which platform MySQL is run on, the default setting will vary!

The only downside of setting up MySQL in this way is that the output of `SHOW TABLES` and `SHOW DATABASES` commands do not preserve the casing in which databases or tables were created, but this is merely a cosmetic issue.

### See also...

- ▶ *MySQL online manual on identifier case sensitivity, the relevant options and sequences, section 8.2.2 at <http://dev.mysql.com/doc/refman/5.7/en/identifier-case-sensitivity.html>.*

## Installing MySQL as a Windows service with custom options

While for development purposes it can be very handy to have MySQL run as a console application on Windows, for regular operations a background service is the option. It has the advantage of starting up and shutting down automatically with Windows, without the need for a user to log in to the machine.

In this recipe, we will show you how to install MySQL as a Windows service manually using the ZIP distribution available from the MySQL homepage and specify a custom configuration file.

## Getting started

Naturally, this is a Windows-only recipe. You will need a Windows user account with administrative privileges to register a new Windows service. Moreover, we assume you have already downloaded the MySQL distribution called "Without installer (unzip it)".

Make sure you choose the release matching your operating system (32 or 64 bit). In this recipe we will be using MySQL 5.1 from <http://dev.mysql.com/downloads/mysql/5.1.html>.



Be advised that security software on your computer might interfere with installation of a Windows service, as some malicious software may try to hook into the system that way. If you encounter problems, you may have to disable anti-virus programs and other security products for the duration of the process. Do not forget to re-enable them when you are finished with MySQL service setup!

## How to do it...

1. Unpack the downloaded ZIP file. Put the contents in `c:\mysql\5.1.xx\` (replacing `xx` with the actual release number you are using).
2. On a command prompt (`cmd.exe`) enter the following commands to install the service. Make sure to enter the full path, instead of changing the working directory with the `cd` command:

```
c:\> c:\mysql\5.1.xx\service\bin\mysqld.exe --install MySQL51
```

```
c:\>c:\mysql\5.1.40\service\bin\mysqld.exe --install MySQL51 --defaults-file=c:\mysql\5.1.40\service\my.ini
Service successfully installed.

c:\>
```

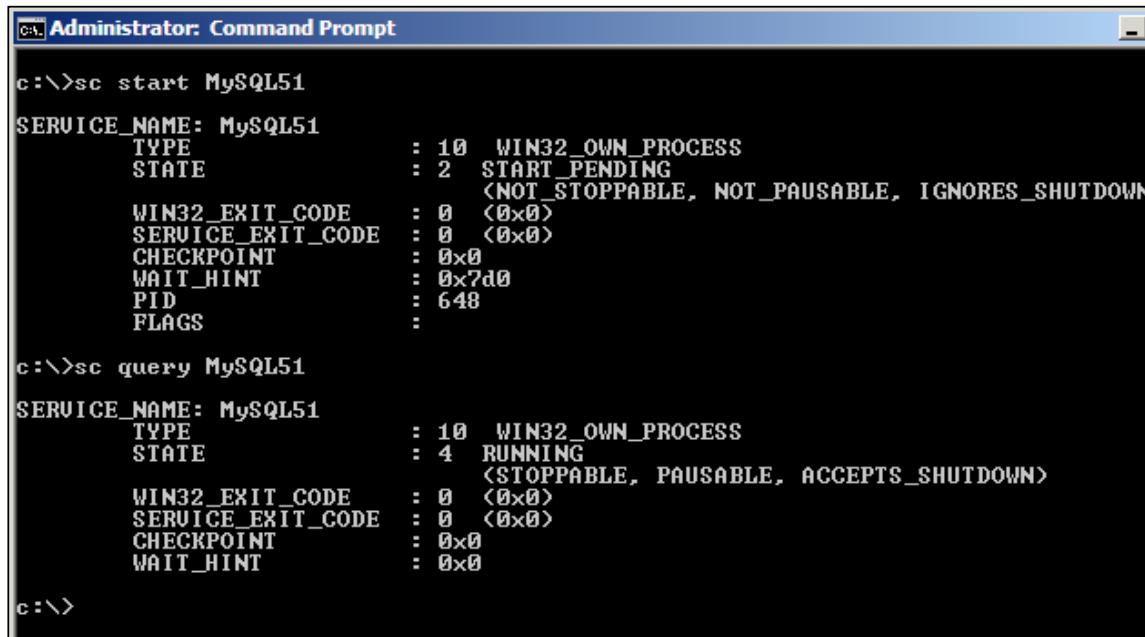
3. Edit the `my.ini` configuration file specified in the command above to meet your requirements.

## Configuring MySQL

- Start the service and verify its status using these commands:

```
c:\> sc start MySQL51
```

```
c:\> sc query MySQL51
```



```
Administrator: Command Prompt
c:\>sc start MySQL51
SERVICE_NAME: MySQL51
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 2   START_PENDING
                           <NOT_STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN>
    WIN32_EXIT_CODE    : 0   <0x0>
    SERVICE_EXIT_CODE : 0   <0x0>
    CHECKPOINT        : 0x0
    WAIT_HINT         : 0x7d0
    PID                : 648
    FLAGS              :

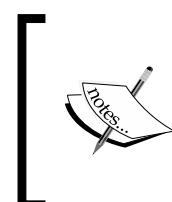
c:\>sc query MySQL51
SERVICE_NAME: MySQL51
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 4   RUNNING
                           <STOPPABLE, PAUSABLE, ACCEPTS_SHUTDOWN>
    WIN32_EXIT_CODE    : 0   <0x0>
    SERVICE_EXIT_CODE : 0   <0x0>
    CHECKPOINT        : 0x0
    WAIT_HINT         : 0x0

c:\>
```

You should see **STATE: 4 RUNNING** in the status output of the second command.

## How it works...

The MySQL server binary executable file `mysqld.exe` contains the necessary functions to register itself with Windows as a background service. There are two options you can provide: `--install` and `--defaults-file`. The first one will specify the name of the service to be created, MySQL51 in this case. The latter is used to define which configuration file the service will read its settings from.



Note that after the `--install` parameter, only a single parameter may follow. While this could be any parameter the MySQL server accepts, using `--default-file` gives you the greatest flexibility, as you can put all the required settings there.

## There's more...

Apart from being able to run without a user having to log in to the server machine, you can define dependencies; so, for example, you could make sure your application starts up and gets started when the database is ready. For details on how to do this, refer to Microsoft's TechNet article #102000 at [http://www.microsoft.com/technet/itpro/tutorials/WindowsServer2003/11/11\\_01.aspx](http://www.microsoft.com/technet/itpro/tutorials/WindowsServer2003/11/11_01.aspx).

---

As services do not have access to a console or the graphical user interface in general, problems encountered while starting the service will not tell you anything about them. On Windows, MySQL will report problems to the system event log, viewable from the Control Panel, and to the MySQL error log file, usually located in the data directory with a name composed from the machine name and a .err extension.

Should your service fail to start, inspect that log file to get an idea of what is wrong. If you are sure your configuration file is OK, we recommend you to start the MySQL daemon directly from the command line like this:

```
c:\> c:\mysql\5.1.xx\service\bin\mysqld.exe --defaults-file=c:\mysql\5.1.xx\service\my.ini --console
```

This will allow you to see any potential problems right away before installing the service.

## Running multiple MySQL server instances parallel on a Linux server

On most Linux setups, MySQL comes as a readymade installation package, making it easy to get started. It is, however, a little more complicated to run multiple instances in parallel. This is because in contrast to Windows, MySQL is not installed in a self-contained directory, but most Linux distribution packages spread the appropriate system folders for programs, configuration files, and so on. You can also install MySQL in its own directory, for example, if you need to use a version not available as a prepared package for your Linux distribution. While this gives you the greatest flexibility as a downside you will have to take care of wiring up your MySQL server with the operating system manually. For example, you will need to hook up the startup and shutdown scripts to the appropriate facilities of your distribution.

In more recent distributions, you can make use of a tool called `mysqld_multi`, a script that lets you set up multiple instances of MySQL daemons with varying configurations. In this recipe, we will show you how to set up two parallel MySQL servers, listening on different ports and using separate data directories for their respective databases.

### Getting ready

This recipe is based on an Ubuntu Linux machine with the 8.04 LTS version. MySQL comes with the MySQL packages for that operating system. If you are using other distributions, you need to make sure you have `mysqld_multi` installed to be able to follow along. Check with your distribution's package repositories for information on which packages you need to install.

*Configuring MySQL*

You will also need an operating system user with sufficient privileges to edit the MySQL configuration file—typically /etc/mysql/my.cnf on Ubuntu—and restart services. For AppArmor or SELinux, we assume these have been disabled before you start to follow the process.

## How to do it...

1. Locate and open the my.cnf configuration file in a text editor.
2. Create the following two sections in the file:
 

```
# mysqld_multi test, instance 1
[mysqld1]
server-id=10001socket=/var/run/
mysqld/mysqld1.sockport=23306pid-
file=/var/run/mysqld/mysqld1.pid
datadir=/var/lib/mysqllog_bin=/var/
log/mysql1/mysql1-bin.log

# mysqld_multi test, instance 2
[mysqld2]
server-id=10002socket=/var/run/
mysqld/mysqld2.sockport=33306pid-
file=/var/run/mysqld/mysqld2.pid
datadir=/var/lib/mysqllog_bin=/var/
log/mysql2/mysql2-bin.log
```
3. Save the configuration file.
4. Issue the following command to verify the two sections are found by mysqld\_multi report
 

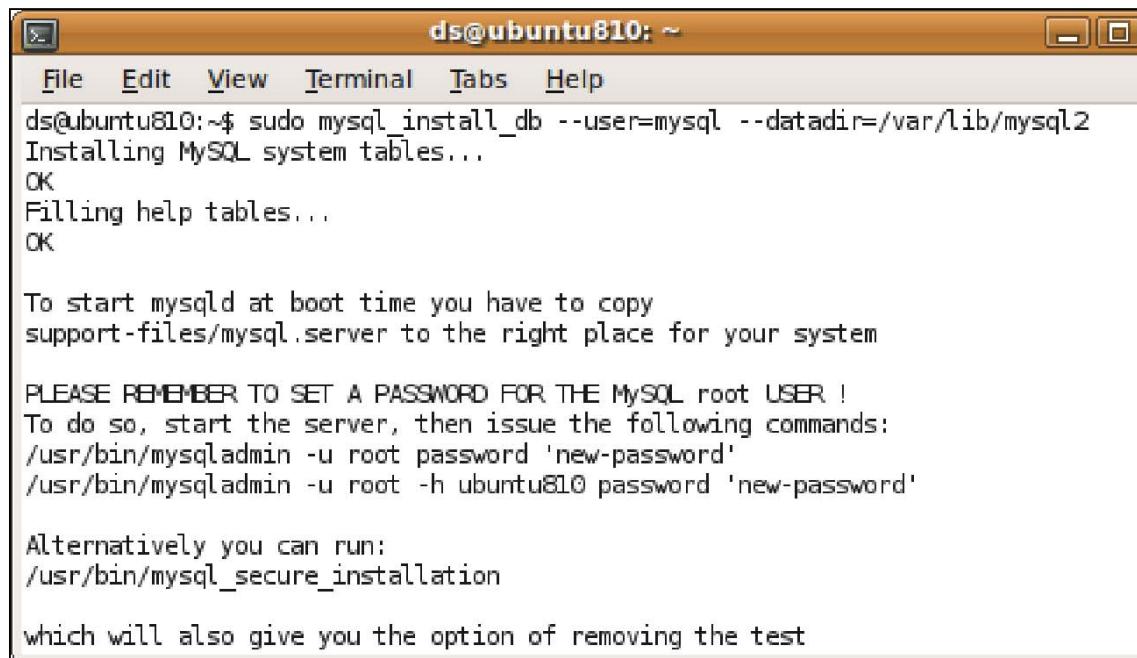
```
$ sudo mysqld_multi report
```



---

5. Initialize the data directories:

```
$ sudo mysql_install_db --user=mysql --datadir=/var/lib/mysql  
$ sudo mysql_install_db --user=mysql --datadir=/var/lib/mysql
```



The screenshot shows a terminal window titled "ds@ubuntu810: ~". It displays the output of the MySQL installation command. The terminal shows the progress of installing system tables and help tables, followed by instructions to set a root password and run mysql\_secure\_installation. It also notes that the test database will be removed.

```
ds@ubuntu810:~$ sudo mysql_install_db --user=mysql --datadir=/var/lib/mysql  
Installing MySQL system tables...  
OK  
Filling help tables...  
OK  
  
To start mysqld at boot time you have to copy  
support-files/mysql.server to the right place for your system  
  
PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !  
To do so, start the server, then issue the following commands:  
/usr/bin/mysqladmin -u root password 'new-password'  
/usr/bin/mysqladmin -u root -h ubuntu810 password 'new-password'  
  
Alternatively you can run:  
/usr/bin/mysql_secure_installation  
  
which will also give you the option of removing the test
```

6. Start both instances and verify they have been started:

```
$ sudo mysqld_multi start 1  
$ sudo mysqld_multi report
```



The screenshot shows a terminal window titled "ds@ubuntu810: ~". It displays the output of the mysqld\_multi report command. It shows that two MySQL servers, mysqld1 and mysqld2, are running.

```
ds@ubuntu810:~$ sudo mysqld_multi start 1,2  
ds@ubuntu810:~$ sudo mysqld_multi report  
Reporting MySQL servers  
MySQL server from group: mysqld1 is running  
MySQL server from group: mysqld2 is running  
ds@ubuntu810:~$
```

## Configuring MySQL

7. Connect to both instances and verify their settings:

```
$ mysql -S /var/run/mysqld/mysql1.sock
mysql> SHOW VARIABLES LIKE 'server_id';
```

```
$ mysql -S /var/run/mysqld/mysql2.sock
mysql> SHOW VARIABLES LIKE 'server_id';
```

The screenshot shows a terminal window titled "ds@ubuntu810: ~". It displays two separate MySQL sessions. The first session connects to instance 1 (server\_id 10001) and the second to instance 2 (server\_id 10002). Both sessions show the MySQL monitor welcome message and the result of the "SHOW VARIABLES LIKE 'server\_id'" command.

```
ds@ubuntu810:~$ mysql -S /var/run/mysqld/mysql1.sock
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.0.67-Ubuntu6-log (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW VARIABLES LIKE 'server_id';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| server_id     | 10001 |
+-----+-----+
1 row in set (0.00 sec)

mysql> Bye
ds@ubuntu810:~$
ds@ubuntu810:~$ mysql -S /var/run/mysqld/mysql2.sock
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.67-Ubuntu6-log (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW VARIABLES LIKE 'server_id';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| server_id     | 10002 |
+-----+-----+
1 row in set (0.00 sec)

mysql>
```

## How it works...

`mysqld_multi` uses a single configuration file for all MySQL server instances, but each instance has its individual [mysqld] section with its specific options. `mysqld` then takes care of launching the MySQL executable with the correct options to use those from its corresponding section.

The sections are distinguished by a positive number directly appended to the word `mysqld` in the section header. You can specify all the usual MySQL configuration file options in these sections, just as you would for a single instance. Make sure, however, to specify the

---

## There's more...

Some special preparation might be needed, depending on the particular operating systems you are using.

### **Turning off AppArmor / SELinux for Linux distributions**

If your system uses the AppArmor or SELinux security features, you will need to make sure that they are either turned off while you try this out, or configured (for permanent use) so that the configuration has been finished) to allow access to the newly defined directories and files. See the documentation for your respective Linux distribution for more details on how to do this.

### **Windows**

On Windows, running multiple server instances is usually more straightforward. MySQL is normally installed in a separate, self-contained folder. To run two or more independent instances, you only need to install a Windows service for each of them and point them to their individual configuration file. For information on how to set up MySQL as a Windows service and how to specify which settings file to use, see the relevant recipe in this chapter.

### **Considering the alternative MySQL Sandbox project**

As an alternative to `mysqld_multi` you might want to have a look at MySQL Sandbox. It offers a different approach to hosting multiple independent MySQL installations on the same operating system. While `mysqld_multi` manages multiple configurations in a single file, MySQL Sandbox aims at completely separating MySQL installations from each other, allowing even several MySQL releases to run side by side. For more details, visit the MySQL Sandbox website at <http://mysqlsandbox.net>.

## See also

- ▶ *Installing MySQL as a Windows service with custom options*
- ▶ *Ubuntu Linux Wiki on AppArmor* at <https://wiki.ubuntu.com/AppArmor>
- ▶ *Fedoraproject.org Wiki on SELinux* <http://fedoraproject.org/wiki/SELinux>

*Configuring MySQL*

## Preventing invalid date values from being stored in DATE or DATETIME columns

In this recipe, we will show you how to configure MySQL in a way such that invalid dates are rejected when a client attempts to store them in a DATE or DATETIME column. This is done by combining a combination of flags for the SQL mode setting.

See the *There's more...* section of this recipe for some more detailed information on the SQL server mode setting in general and on how to use it on a per-session basis.

### Getting ready

The configuration options shown in this recipe can be applied to individual sessions or server-wide defaults. For production systems, we recommend specifying them in the MySQL configuration file. You will need the necessary operating system level privileges to then restart the service to activate the settings.

The final step in the recipe is the attempt to insert some invalid dates. You can safely skip this step. If you want to try it, you will need a table set up like this in the test database:

```
CREATE TABLE table_a (
    test_date DATE NOT NULL
) ;
```

### How to do it...

1. Locate the MySQL configuration file, typically `my.cnf` or `my.ini` (on Windows), open it in a text editor.
2. In the `[mysqld]` section make sure the following line is present, adding it if needed:
 

```
[mysqld]
..sql-
mode=STRICT_ALL_TABLES,NO_ZERO_DATE,NO_ZERO_IN_DATE ..
```
3. Save the file.
4. Restart the MySQL server.

- 
5. Verify whether the setting was applied using this statement from a MySQL

```
mysql> SELECT @@GLOBAL.sql_mode;
```

```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
mysql> SELECT @@GLOBAL.sql_mode;
+-----+
| @@GLOBAL.sql_mode |
+-----+
| STRICT_ALL_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE |
+-----+
1 row in set (0.00 sec)

mysql>
```

6. Optionally try to insert some false values:

```
mysql> INSERT INTO table_a VALUES ('2009-02-31');
mysql> INSERT INTO table_a VALUES ('2009-00-31');
mysql> INSERT INTO table_a VALUES ('0000-00-00');
```

```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
mysql> INSERT INTO table_a VALUES ('2009-02-31');
ERROR 1292 (22007): Incorrect date value: '2009-02-31' for column 'test_date' at
mysql> INSERT INTO table_a VALUES ('2009-00-31');
ERROR 1292 (22007): Incorrect date value: '2009-00-31' for column 'test_date' at
mysql> INSERT INTO table_a VALUES ('0000-00-00');
ERROR 1292 (22007): Incorrect date value: '0000-00-00' for column 'test_date' at
mysql>
```

## Getting ready

Setting the SQL mode to `STRICT_ALL_TABLES` will enable validation on all tables (as opposed to only those in transactional storage engines if you were to use `STRICT_TRANS_TABLES`). While setting up the SQL mode like this would already prohibit the insertion of values like Feb 31st, one could still insert all zero dates or dates with zero fields in them. This is what the other two options `NO_ZERO_DATE` and `NO_ZERO_IN_DATE` take care of.

## Configuring MySQL

### There's more...

Starting with MySQL 5.0, the concept of *SQL modes* was introduced to provide granular control over the degree of leniency the server will apply for invalid values. MySQL has traditionally been very forgiving when receiving invalid values to be inserted into its tables. There are truncation and approximation rules on what will happen when, for example, you try to insert a value that exceeds the maximum length of a column's definition.

For enterprise systems, this clearly is unwanted behavior. Whenever an application stores values in the database that do not meet the previously defined criteria of leniency, such as valid date or value ranges, an error must be thrown to prevent silent data corruption. One might argue that data validation must be done at the application level and invalid data never be stored in the database anyway. But we are strong believers in the database being the "last line of defense". Of course, any decent application will reject invalid inputs, but in reality there can be situations where an administrator accessing the database independently might just make a mistake. Using MySQL to verify incoming data (again) can be invaluable in these situations. Even though it may take a slight performance hit, data integrity should be considered as an even higher priority.

The so-called **strict mode** enables the general use of MySQL server-side data validation. The remaining options described in the MySQL online manual section 5.1.8 at <http://dev.mysql.com/doc/refman/5.1/en/server-sql-mode.html> allow a somewhat granular control over what exactly get validated and how.

We recommend going through all the SQL mode options to see if there are any that you would like to enable on your servers.

### Configuring SQL mode for the current session only

For experimenting with the different SQL modes, it is often easier to configure them for the current session only. The following statement disables the global settings configuration for the current session only:

```
mysql> SET @@session.sql_mode='';
```

```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
mysql> SET @@session.sql_mode='';
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO table_a VALUES ('0000-00-00');
Query OK, 1 row affected (0.00 sec)

mysql>
```

This can also come in handy for maintenance scripts that need to temporarily disable strict mode.

# MySQL User Management

In this chapter, we will cover the basic tasks related to MySQL user management. You will learn about the following topics:

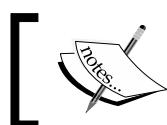
- ▶ Configuring MySQL Administrator to display global privileges and hosts
- ▶ Defining an alternative user for administrative tasks
- ▶ Disabling the default accounts
- ▶ Creating a basic user
- ▶ Creating an installation user
- ▶ Creating a read-only account
- ▶ Defining a specific user for backup
- ▶ Defining a specific user for replication
- ▶ Allowing access from specific hosts only
- ▶ Synchronizing user permissions across servers
- ▶ Regaining access to your database in case of lost account information
- ▶ Avoiding plain text passwords in administrative scripts

## Introduction

While MySQL has a reputation for being easy to set up in the first place (and rightly so), it's important to keep in mind that the initial configuration, for example, as provided by the MySQL installation wizard, needs some more tweaking for production use. This particular chapter will help you do just that.

## MySQL User Management

Whenever you connect to a MySQL server, the connection is associated with a specific user account. Even if you do not specify the user explicitly, in which case a default account is selected. If you tell MySQL Windows installer to create an anonymous user, the resulting configuration allows for full local access to the database without having to provide any kind of credentials. Whoever is able to access the machine that runs your database will be able to play with your data—which is not something you typically want for your production systems.



Do NOT use an anonymous user on production systems!



The definition of users and their respective access privileges is often considered a downstream configuration task that can be tackled shortly before production use. We strongly recommend defining the basic roles for database users in advance. This often helps to structure the way the database is accessed, which in turn might improve the systems architecture and prevent certain development flaws.

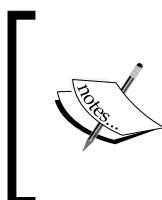
In this chapter, we will have a look at some typical user roles found in MySQL production environments. First of all, we will discuss how to create a hardened administration account that is not accessible anonymously, and then how to get rid of the default users present in the MySQL installation. Additionally, we will introduce accounts for basic operational access, and refer to technical users for replication, backup, and recovery.

Besides this, we will also cover how to restrict database access for certain users to specific clients. This makes for an additional line of defense because knowledge of a user's password are not sufficient to connect to the database. A potential attacker would need to know both the user name and the password to gain access.

---

As MySQL installations with more than one database server are not an exotic configuration any more, the sometimes tedious task of keeping the user definitions in sync through multiple instances will be addressed as well. And finally, we will not conceal the downside of security: if a potential attacker will not be able to access the database without knowing your username and password, you will not, either. We will show you how to access your database even if the sticky notes with your credentials are lost.

The recipes in this chapter will primarily focus on MySQL Administrator as the tool of choice when manipulating the user accounts and their privileges.



MySQL Administrator is an administration client provided by MySQL as part of the MySQL GUI Tools Bundle, which is available for free at <http://dev.mysql.com/downloads/gui-tools/5.0.html>.

However, in some situations a graphical user interface just does not fit the requirements. This is particularly the case if scripting is required, for example, for automated and unattended changes. For these situations, we will also show alternative ways to change the user rights without the help of MySQL Administrator.

While the recipes in this chapter will give you an overview of a typical role configuration, encourage you to adapt this proposal according to your needs. And from our experience, it is typically a good idea to discuss the possible user configuration with your IT security department—if your company happens to have one—well in advance, so as to prevent discussions later on.

## Configuring MySQL Administrator to disable global privileges and hosts

Throughout the following recipes, we will use MySQL Administrator as the main tool for managing user rights. Some of the privileges, however, are not visible in MySQL Administrator by default configuration. This recipe will show you how to change the relevant options so that you will be able to manage even the global privileges.

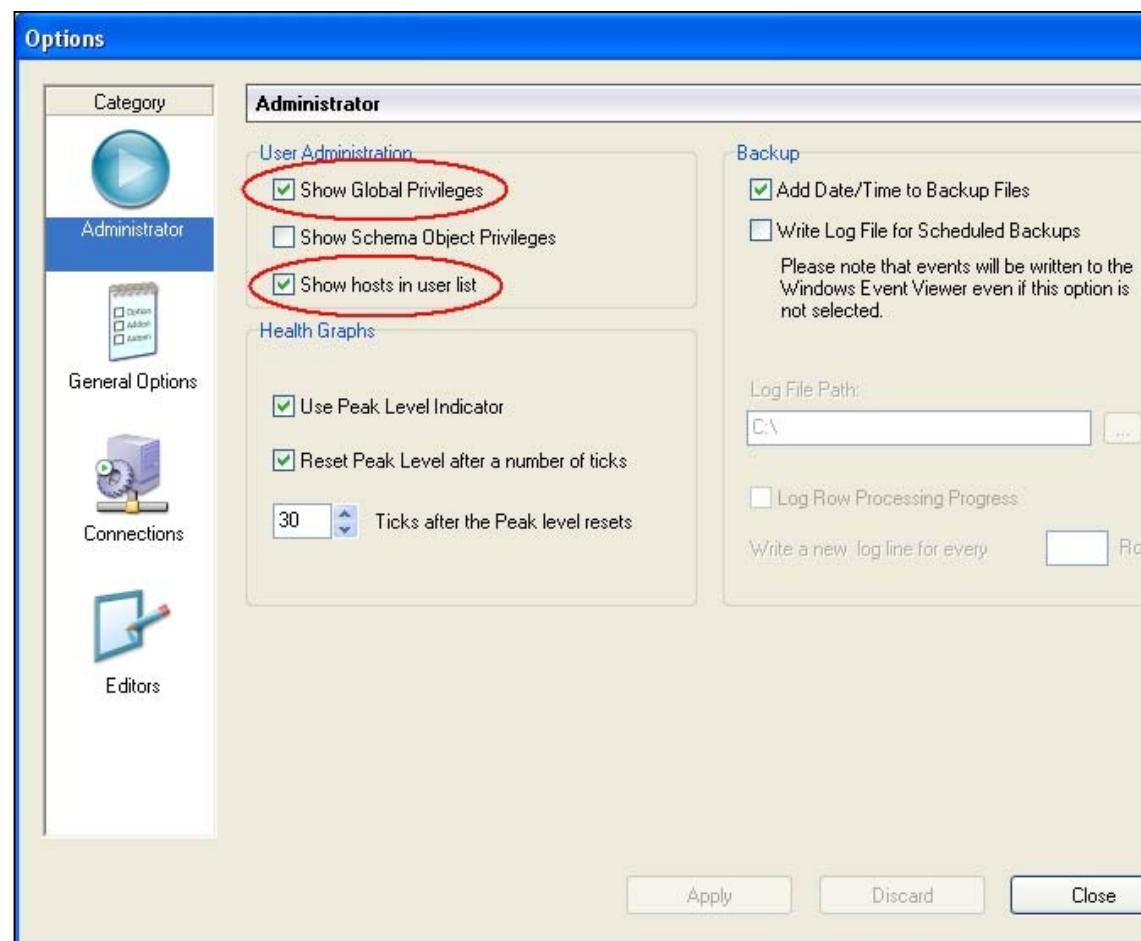
### Getting ready

To step through this recipe, you will need a running MySQL database server and a local installation of MySQL Administrator. No other prerequisites are required.

## MySQL User Management

### How to do it...

1. Start MySQL Administrator. Connect to your database server using any account.
2. Select the entry **Options...** from the **Tools** menu.
3. Make sure the options **Show Global Privileges** and **Show hosts in user list** are selected.



4. Select **Close**. If a **Save changes?** dialog comes up, press the **Yes (Apply changes permanently)** button.

### How it works...

With the options selected throughout this recipe, MySQL Administrator will allow you to view and change not only the schema privileges that define the operations allowed on specific schemata, but also the global privileges.

The available global privileges are partially identical to the schema privileges, for example **SELECT** or **UPDATE**, but there are also some additional privileges, such as **SHUTDOWN**, which affect the whole database regardless of specific schemata.

The options affect the **User Administration** view of MySQL Administrator. You will see an additional tab **Global Privileges**, and when you select a user account in the list to the left, the hosts for which the specific rights were defined are listed.

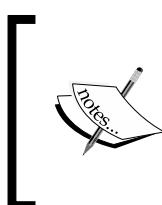
## Defining an alternative user for administrative tasks

During MySQL installation, an account for user *root* is created. This account receives all privileges without any exceptions. In most installations, this configuration is basically unchanged, which makes the *root* account a very rewarding target for attacks. An attacker will not have to guess a username and a corresponding password—the password *root* is a safe bet because it is a safe bet to assume the existence of a username *root*. This is why we recommend that you create an administration account with a different name.

This recipe will show you how to create a user account that can act as a replacement for the *root* user.

### Getting ready

You are going to need a catchy username for your administration account and a corresponding password. When making up the password, keep in mind the typical recommendations for good passwords (hard to guess, but easy to remember; should contain upper and lower case letters, numbers, and special characters). Throughout this recipe, we will assume a username *admin4mysql* and a password *As,ysp4M* ("A simple, yet strong password for MySQL") throughout this recipe.



The *admin4mysql* user is only used as an example. For security reasons, please do NOT use this account name (or any of the other users introduced in this chapter) for real-world installations! We strongly suggest using your individual usernames instead.

Additionally, you will have to make sure that manipulation of global privileges is enabled in the MySQL Administrator options. Please refer to the *Configuring MySQL Administrator* and *global privileges and hosts* recipe if in doubt.

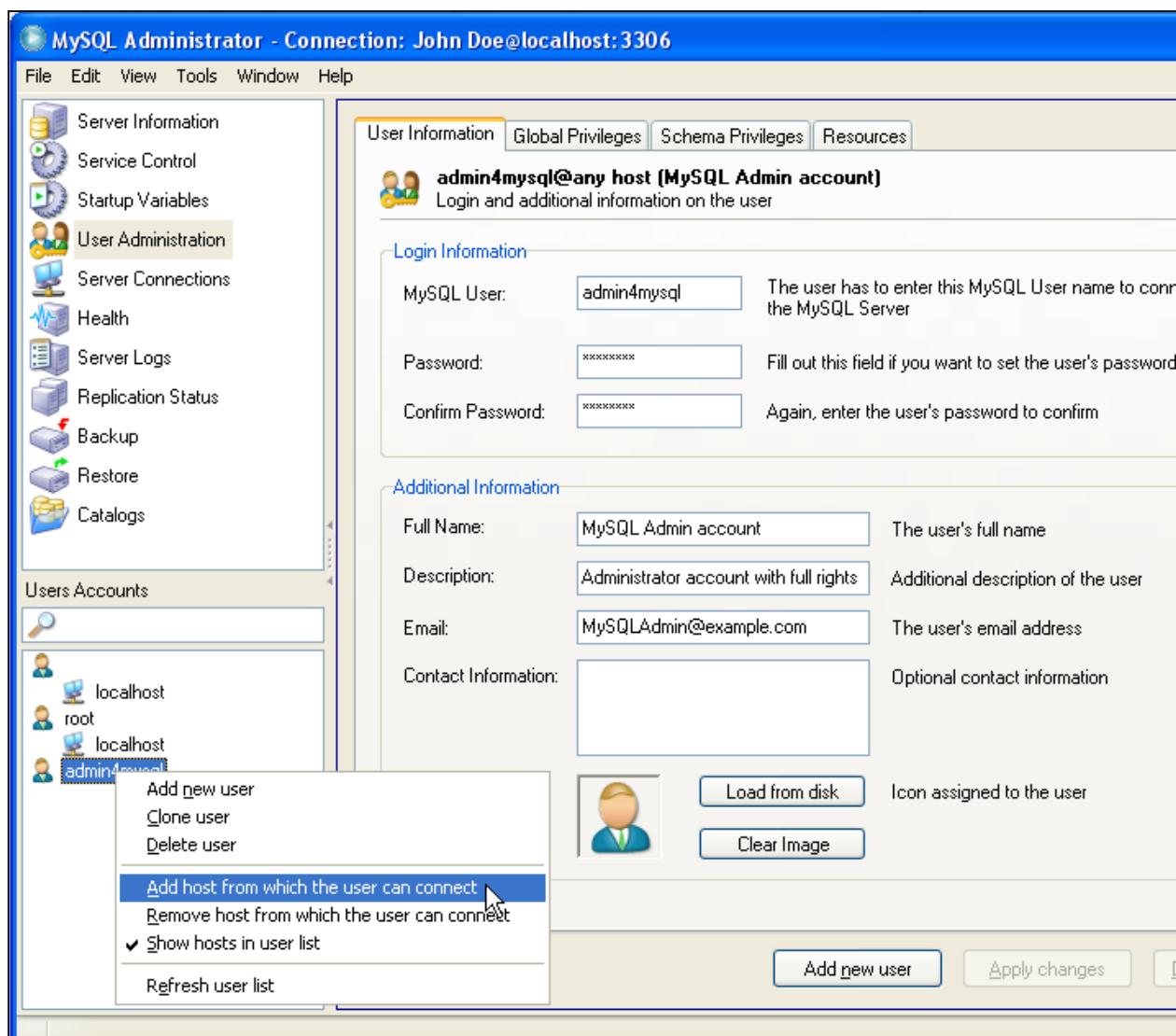
Finally, you have to think about from which host you are going to connect to your MySQL instance to perform administrative tasks. In most cases, this will be `localhost`, which we will assume as host in the following recipe. In some situations, however, this might differ. For example, if you have no login rights on the database host itself. In these situations, define a certain host as your base for administrative tasks. Have its host name or IP address ready for the following tasks.

## MySQL User Management

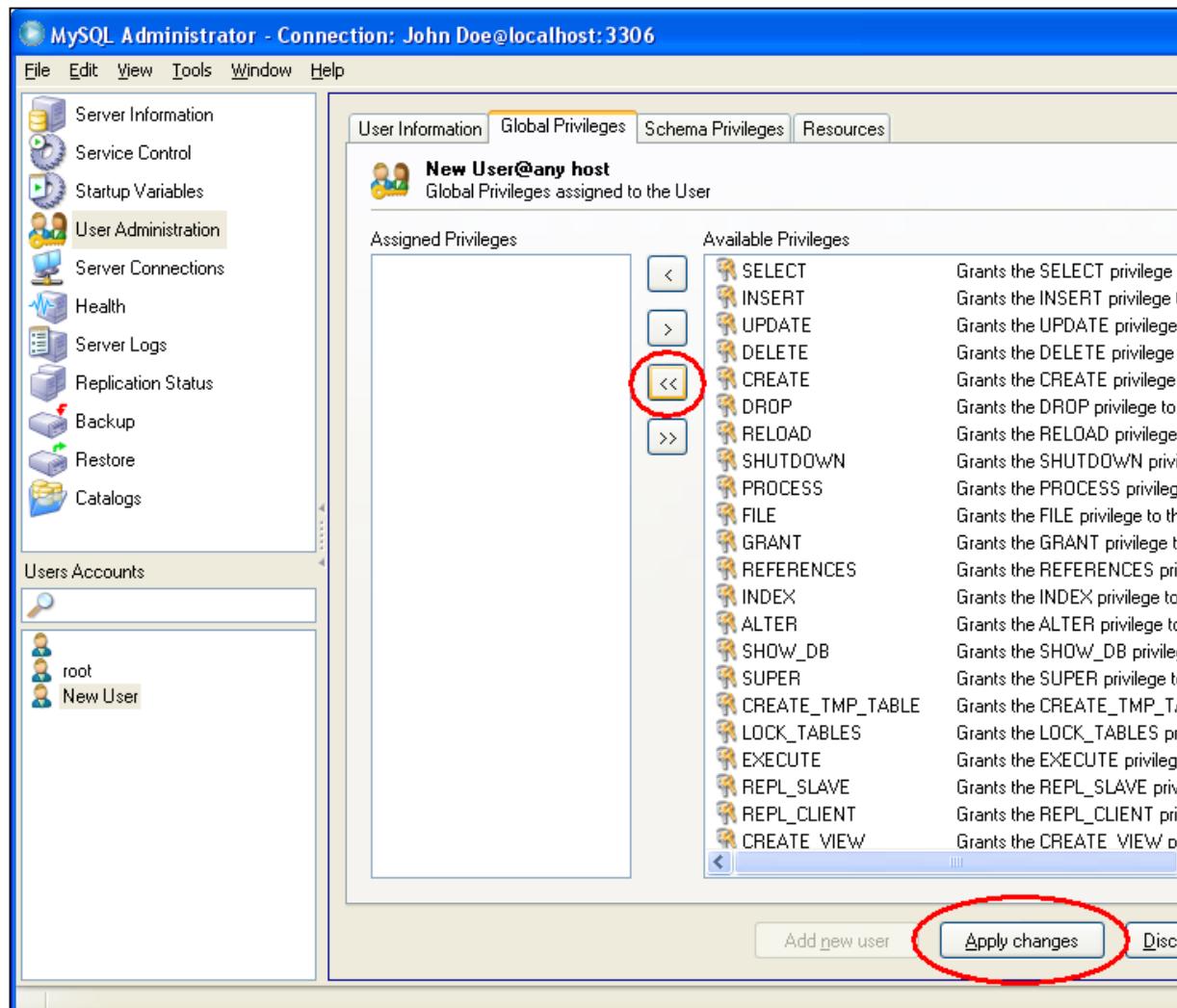
The steps below will guide you through the process of creating an alternative administration account.

### How to do it...

1. Start MySQL Administrator. Connect to your database server using the **root** user.
2. Select the **User Administration** entry either from the list on the left or from the **View** menu.
3. Click on the **Add new user** button.
4. Enter the basic user information (username, password, contact information) by a click on the **Apply changes** button.
5. Right-click on the new user *admin4mysql* (in the user list on the lower left) choose the option **Add host from which the user can connect**.



6. In the following form, enter the host from which you are going to perform your administration tasks (typically localhost).
7. Select the tab **Global Privileges**. Choose the << button to grant all global privileges to the user, followed by a click on the **Apply changes** button:



8. Right-click on the **admin4mysql** entry on the user list and select **Remove which the user can connect** from the context menu.
9. Confirm the message box indicating **The any-host (%) entry has been deleted** followed by the **Apply changes** button.

## How it works...

Let's take a look at what we did throughout the above recipe. In steps 1 through 4, we created a new user named `admin4mysql`. At this point, this user could connect to the database from any host, but as new users have no initial rights whatsoever, he or she would not be able to actually do anything. With steps 5 and 6, we defined a specific host from which he or she would be allowed to log on. Step 7 finally assigns all rights to the newly created user.

## MySQL User Management

With steps 8 and 9, the user `admin4mysql` is no longer able to connect from any host other than `localhost`. This makes it necessary for a possible attacker to log in on the host itself if he or she can try to gain access to the database. This in turn ensures brute force attacks against the MySQL server via the network from a random host for this user will not be successful.

### There's more...

MySQL Administrator features a graphical user interface—it is not the tool of choice in situations such as if scripting capabilities are needed. In these circumstances, a simple script will perform the same changes as the above recipe. This script could be executed, for example, by using the `mysql` command-line client and connecting with a privileged user like your current `root` account:

```
GRANT ALL PRIVILEGES ON *.* TO 'admin4mysql'@'localhost'
IDENTIFIED BY 'As,ysp4M' WITH GRANT OPTION;
```

The values for password and maybe host name have to be adapted according to your needs.

### See also

- ▶ *Configuring MySQL Administrator to display global privileges and hosts*
- ▶ *Avoiding plain text passwords in administrative scripts*

## Disabling the default accounts

In the previous recipe, we created a new user for administration tasks as a replacement for the default `root` user. If and only if another user with full rights is available, we can disable the default users that were created during MySQL installation. This helps harden your installation against possible intruders who will not be able to attack well-known accounts.

### Getting ready

You have to assure yourself that you have a second user with full rights (in addition to the `root` user) at your disposal. By deleting the `root` account, you risk a database with no means of administration—unless there is an equivalent user available. This is why MySQL Administrator won't let you delete this user and hence we will have to resort to the command-line client to delete the `root` account.

The following steps will show you how to remove the default accounts.

## How to do it...

1. Connect to your database server with the `mysql` command-line client using an additional administration account (`admin4mysql`).
2. To delete the `root` user, issue the following commands:

```
DROP USER 'root'@'localhost';
DROP USER 'root'@'%';
```

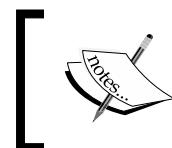
3. To delete the anonymous user, execute these commands:

```
DROP USER ''@'localhost';
DROP USER ''@'%';
```

## How it works...

MySQL creates up to four default accounts, depending on the choices made during installation. These four accounts use two usernames, namely `root`, and the anonymous user, with an empty username. For each username the two hosts—`localhost` and `%`—are listed, with the latter being the place holder for any host. Thus the default accounts are as follows:

- ▶ `'root'@'localhost'`
- ▶ `'root'@'%'`
- ▶ `'@'localhost'`
- ▶ `'@'%'`



The first account `'root'@'localhost'` is always available, while the other three are only created in certain installation scenarios.

With the commands in the above steps, all possible default accounts will be deleted. If the users do not exist, you will receive an error message (error code 1396 "operation failed for user 'root'@'localhost' (user does not exist)") indicating that the account does not exist. After having executed all of these commands, you can be sure that no unwanted default accounts are left.

You can test the effect of the steps performed by using MySQL Administrator. The absence of the `root` user and the anonymous user (with the empty name) will be gone in the **Administration** view if you reconnect to your database server.

## See also

- ▶ *Regaining access to your database in case of lost account information*

## MySQL User Management

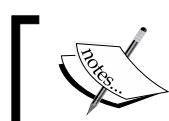
# Creating a basic user

If installed from scratch, MySQL does not provide a default user suitable for everyone. You could of course use the *root* user or an alternative administrative user (for example *admin4mysql*) with full rights. However, this is strongly discouraged for security reasons, especially if you have different users and/or applications that need access to your database. In the following recipe, we will show you how to create a typical user that has access to a certain database (schema).

## Getting ready

You will need some information before stepping through this recipe:

In the first place, you will need a username and a password. If the user is a person going to use the database interactively, you will mostly use his or her real name as the name. For accounts that will be used by applications, it is typically a good idea to include the application (and—if applicable—the role for which the account will be used) in the account name. Examples would be *john\_doe*, *hotelbooking*, or *carrental\_stdusr*. Keep in mind, however, that the length of the username (as of MySQL 5.1) is limited.



The length of a MySQL username must not exceed 16 characters!

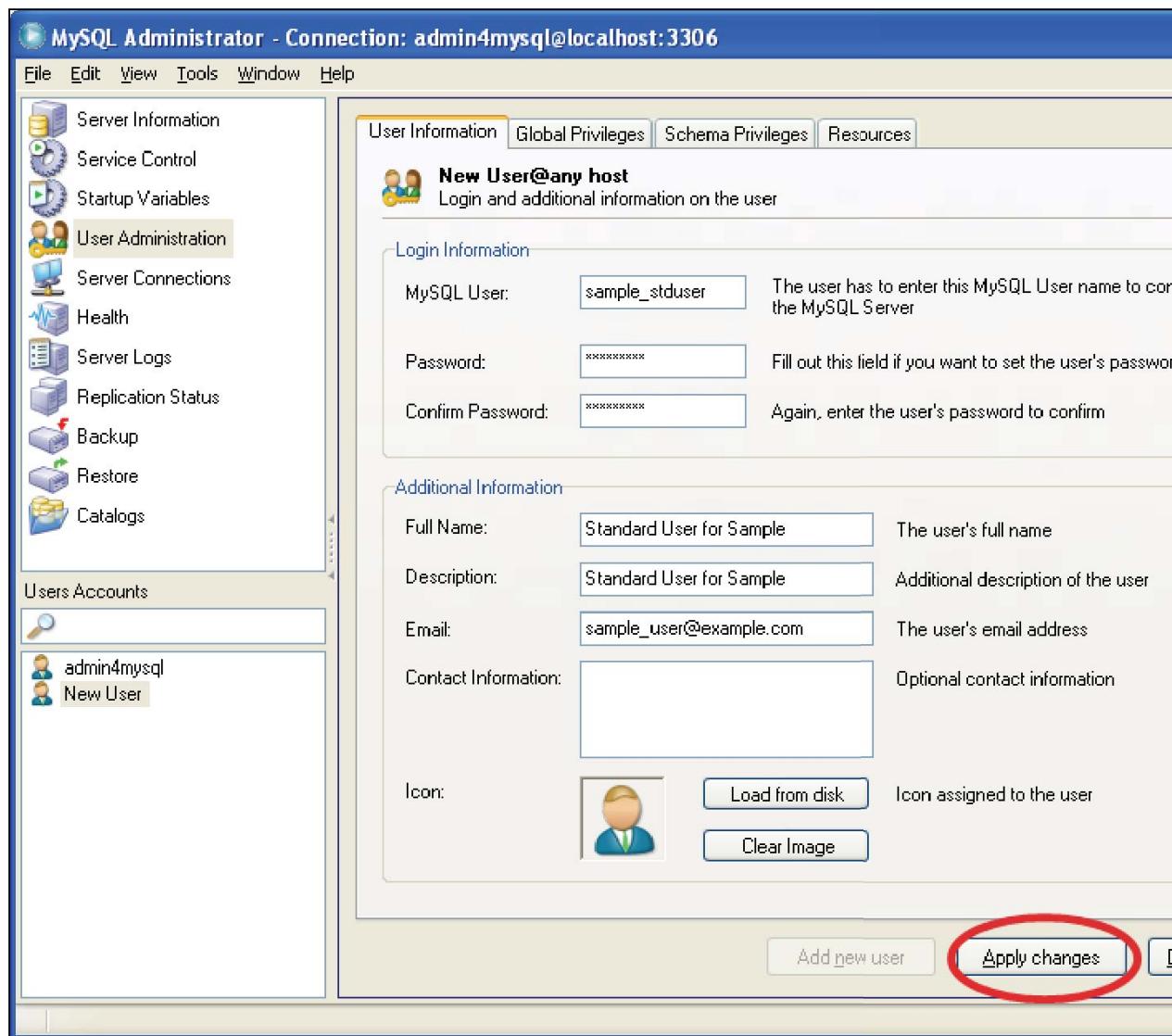
Second, you should find out about which database of your MySQL installation has been accessed by your user. This database has to exist already, otherwise you will not be able to assign the respective rights using MySQL Administrator (however, the script solution in the *There's more...* section would even work without an existing target schema).

Throughout this recipe, we will show you how to create a basic user *sample\_stduser* with read-write access to a database named *sample* and a password *S4mp13-Pw*. As a prerequisite, we assume that the database *sample* already exists.

## How to do it...

1. Start MySQL Administrator. Connect to your database server using the *admin4mysql* account.
2. Select the entry **User Administration** either from the list on the left or from the **View** menu.
3. Click on the **Add new user** button.

4. Enter the basic user information (username, password, contact information) by a click on the **Apply changes** button.

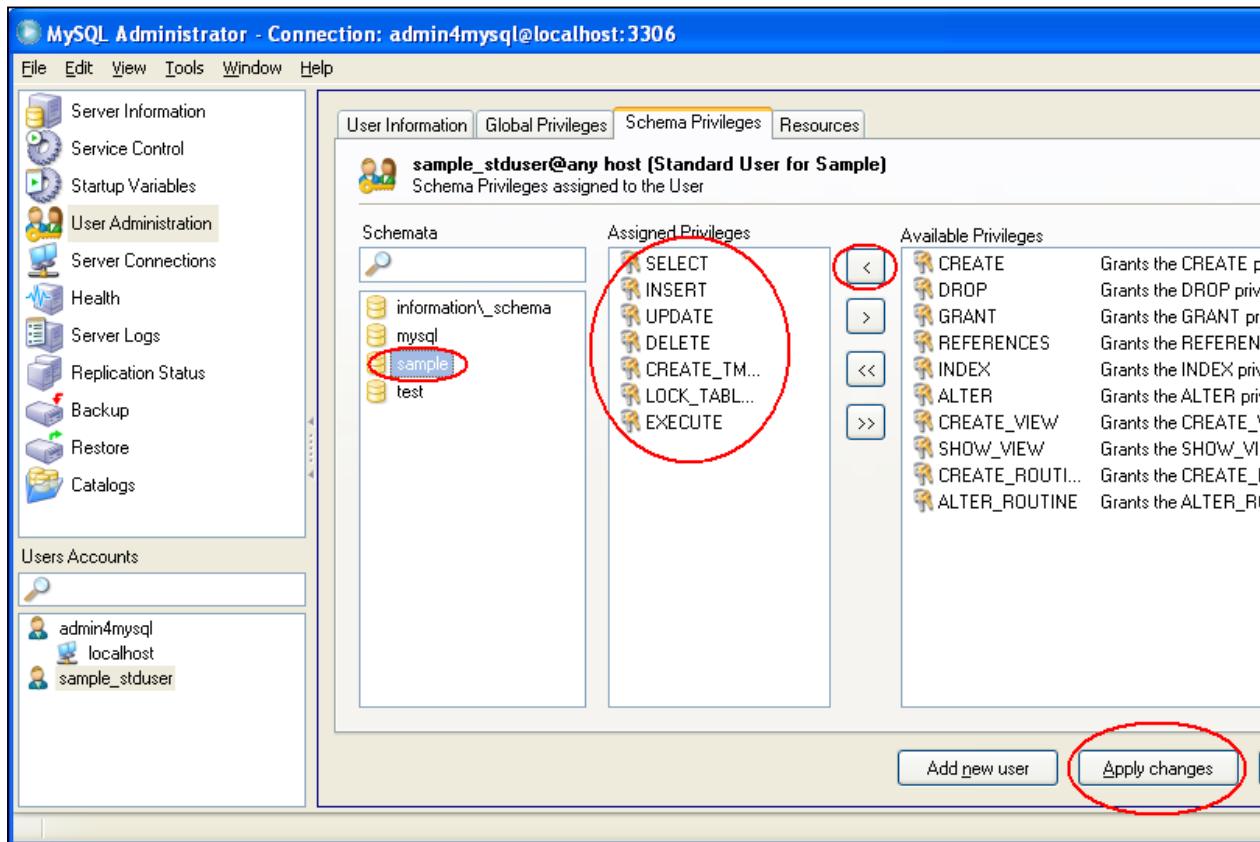


5. Select the tab **Schema Privileges** and select the schema sample from the list on the left. Select the privileges **SELECT, INSERT, UPDATE, DELETE, CREATE, TEMPORARY TABLES, LOCK TABLES**, and **EXECUTE** from the Available Privileges list on the right, and for each selected privilege press the < button. Choose **changes** button afterwards.

## MySQL User Management



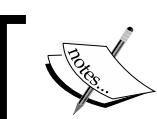
To select and assign more than one privilege at once, hold down the **Ctrl** key while selecting the privileges.



## How it works...

Step 4 creates a new user who is allowed to connect from any host, but has no privileges whatsoever. By assigning the privileges in step 5, the user is granted the rights to read (SELECT) and change (INSERT, UPDATE, DELETE) data from any table in the sample schema. These are the most basic rights that are typically necessary for a basic user.

Additionally, we grant the CREATE TEMPORARY TABLE privilege. This sometimes comes in handy for complex statements and the corresponding risks are typically acceptable. Temporary tables are not visible to other users and are limited to the database connection by which they are created. Moreover, they are purged as soon as the connection is closed. Typically, it is not a problem to grant this right. However, as with all privileges the best rule holds true for this right as well.



Don't assign unnecessary privileges!

---

If you are sure that this privilege will not be used, it is recommended not to assign user you created.

The `LOCK TABLES` privilege allows the user to completely lock tables. This is somewhat helpful to coordinate concurrent access to data by different sessions and is needed in some applications that make use of `LOCK TABLES` statements to prevent concurrent modifications. On the other hand, this privilege enables a user to block read and write access to all tables at discretion, which might basically render the database useless to other users in case of extensive or incautious use of `LOCK TABLES` statements. If you are sure that your user won't need such statements, it is recommended not to assign this privilege.

Finally, the `EXECUTE` privilege gives the user the right to call stored routines. As the user has not been granted the rights to create or change such routines, granting this privilege does not pose a noteworthy additional risk even if there are no stored routines to execute. If there are stored routines, however, the `EXECUTE` privilege has to be assigned for the user to be able to call these stored routines. Some applications make use of stored routines, so this right was included for typical MySQL installations. But as stored procedures and functions are still a fairly new feature of MySQL, they are not yet very widely used. So in many cases, you can do without this privilege.

With this set of privileges, we created a user that has full access to all tables in our sample database. This account is a reasonable compromise between rights granted and restrictions placed on the user. However, you will need a corresponding installation user that can be used to create the schema in the first place. The creation of this user is described in the following recipe.

## There's more...

If you want (or need) to avoid using MySQL Administrator for creating the basic user account, you can alternatively issue a single SQL statement instead (the values for password and database name have to be adapted according to your choices):

```
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE TEMPORARY TABLES,
    TRIGGER, EXECUTE ON sample.* TO 'sample_stduser'@'%' IDENTIFIED BY
    'S4mpl3-Pw';
```

This is helpful if you need a scripting solution for user definitions, but it also gives you the possibility to create the user before the target database schema exists. With MySQL Administrator, you have to define the schema first because it will only let you assign privileges for existing databases. This script could be executed, for example, by using the MySQL command-line client.

## See also

- ▶ [Defining an alternative user for administrative tasks](#)
- ▶ [Avoiding plain text passwords in administrative scripts](#)

## Creating an installation user

In the previous recipe *Creating a basic user*, we discussed how to define a typical user for accessing a certain database schema. Additionally, there is an administration user that has full rights. For installations that are managed by one single administrator, this might be sufficient. But often you do not want to perform all administrative tasks by yourself. For example, you might want to delegate the task of defining the database structure to another person. Some applications also have their own installation routines that require rights to set up a database schema. For these tasks, you should consider creating a specific installation user for certain databases. This user should not have global rights, but should be able to make changes to one specific database. This way, you can delegate certain administrative tasks without the risk of users tampering with other databases that are not their business.

Even if you manage your database installation all by yourself, it might be a good idea to use a specific account to perform these tasks, as this helps prevent accidental changes to other databases.

This recipe will guide you through the steps of creating such an installation user for a specific database (schema).

### Getting ready

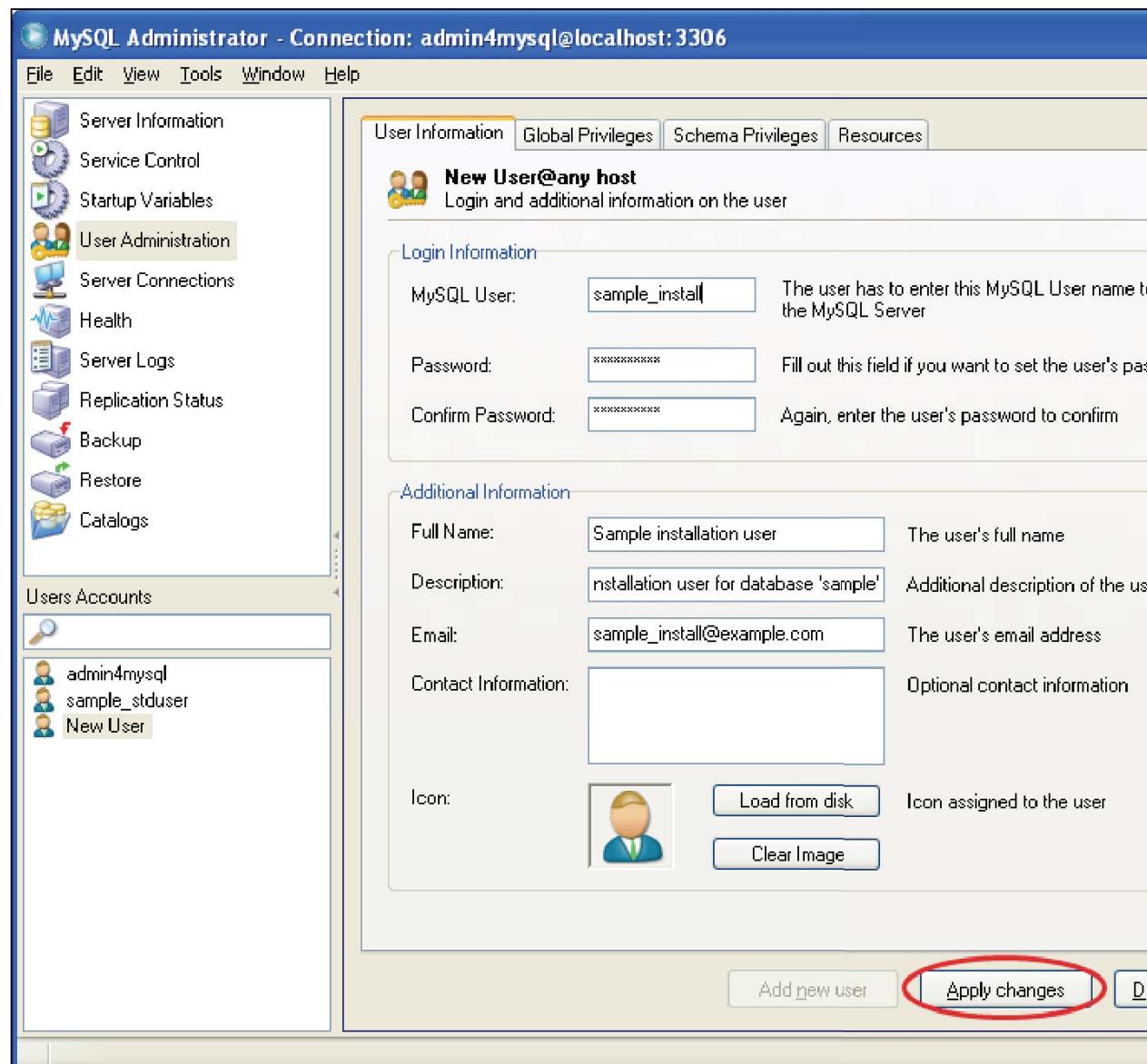
For this recipe, you will again need to come up with a username (remember the 16-character length limit), a password, and the name of the database that will be accessible to this user. As before, this database should already exist, otherwise you will have to resort to the *There's more...* solution from the *There's more...* section of this recipe.

We will assume *sample\_install* as the username, *sample* as the database, and *1mypass* as the password.

### How to do it...

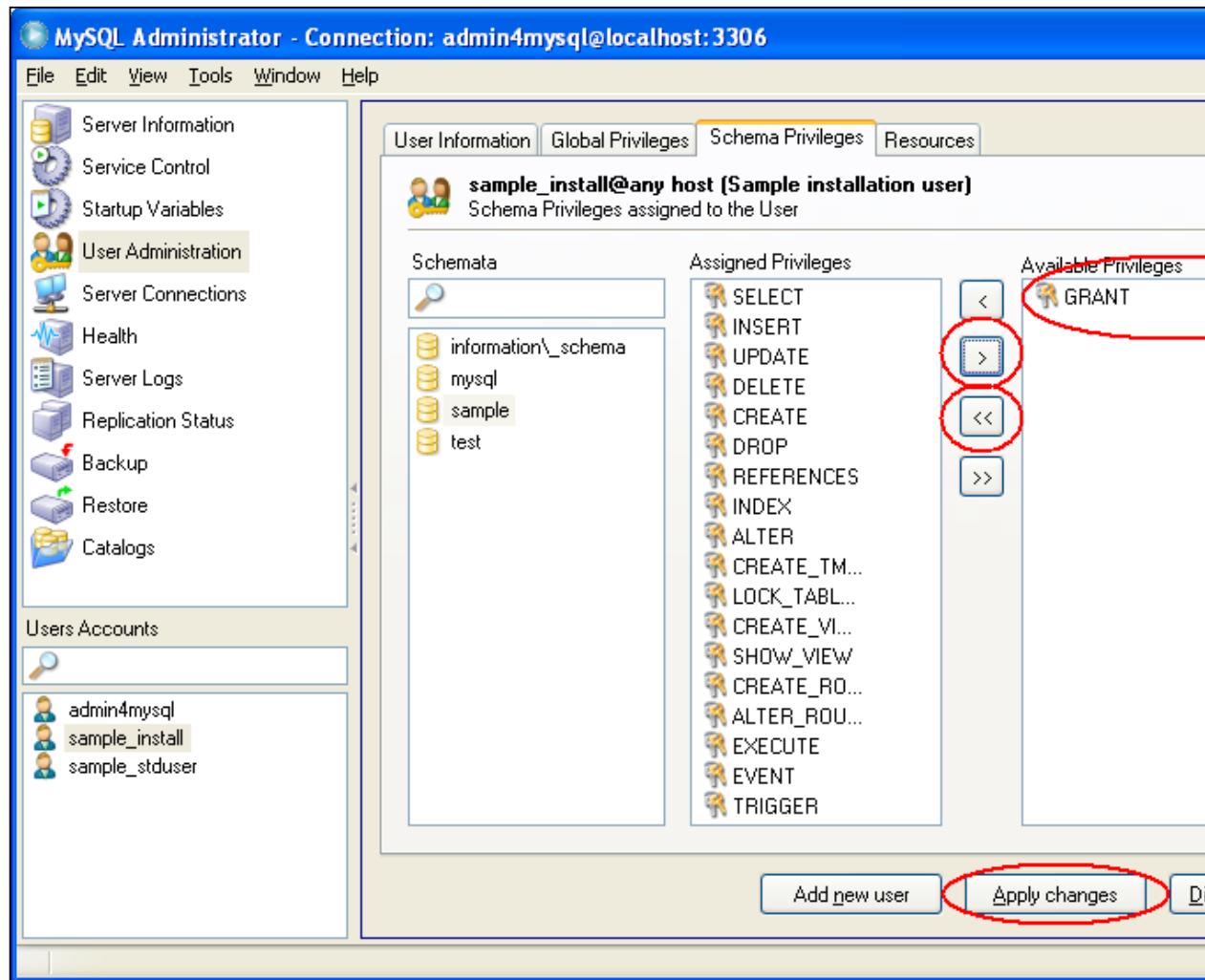
1. Start MySQL Administrator. Connect to your database server using the administration account (*admin4mysql*).
2. Select the entry **User Administration** either from the list on the left or from the **View** menu.
3. Click on the **Add new user** button.

4. Enter the basic user information (username, password, contact information) by a click on the **Apply changes** button.



## MySQL User Management

5. Select the tab **Schema Privileges** and select the schema **sample** from the list on the left. Press the << button to assign all privileges for the sample. Next, select the **GRANT** privilege from the **Assigned Privileges** list and press the >> button to exclude the **GRANT** right. Choose the **Apply changes** button after.



## How it works...

The new user is created in steps 1 through 4, while step 5 assigns all rights on the database to the user, with the **GRANT** privilege being the only exception (also see [more...](#) section of this recipe).

This user has basically full rights for the whole database schema. This account can be used to set up the database, which might, for example, involve creation, deletion, or making changes to tables, management of stored routines, or definition of views. However, it's recommended to use such a powerful account for normal operations. You should probably stick to a basic user for this instead, as we have defined in the previous recipe.

---

## There's more...

In this section, we will discuss some advanced aspects of the user creation, such as solutions to the problem and granting rights to create users to certain accounts.

### **Creating the account without using MySQL Administrator**

You can also create the account without having to use MySQL Administrator by issuing the following SQL command (adapt the values for password and database schema accordingly):

```
grant ALL PRIVILEGES on sample.* to 'sample_install'@'%' identified by 'In5t4ll-Pw';
```

As mentioned before, this allows for scripting as well as defining the account without creating the database beforehand. For example, you could use the MySQL command to execute this script.

### **Permitting management of user rights**

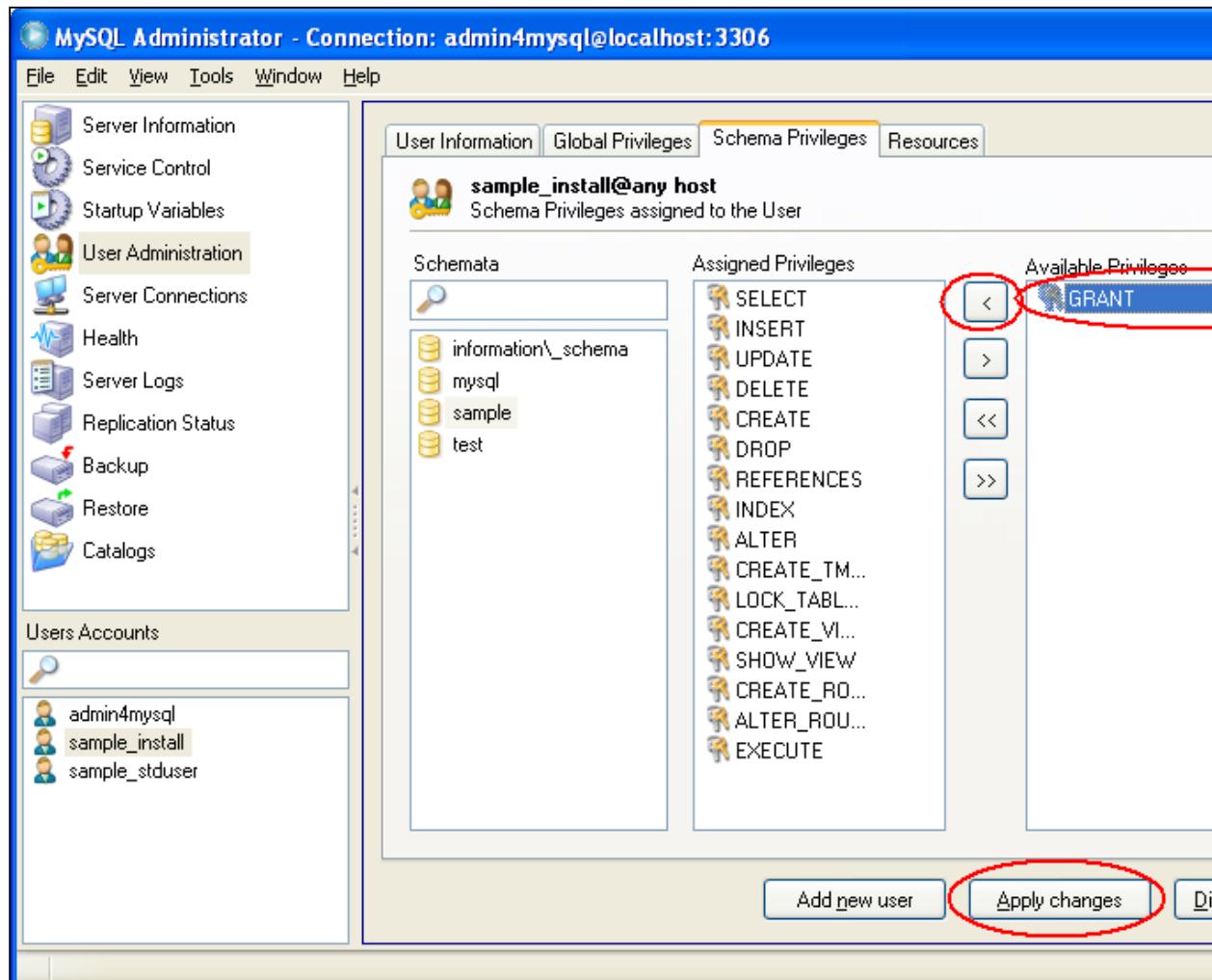
Some applications feature highly sophisticated installation routines, which try to perform many tasks automatically. In some cases, they also try to define MySQL user rights to make sure they have the correct set of privileges. If this is the case, you might need to permit privilege management for the installation user.

As another example, you might want to delegate not only the responsibility of managing a database, but also the (sometimes tedious) task of maintaining the privileges of the corresponding user accounts.

In both situations, you should consider assigning the GRANT privilege to the installation user. This right allows granting those rights to other users (or revoking them from them) without granted yourself. As a result, the GRANT privilege will not allow the installation user to change his or her rights.

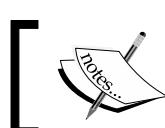
## MySQL User Management

To assign the GRANT privilege using MySQL Administrator, select the *sample\_install* user in the **User Administration** view, switch to the **Schema Privileges** tab, select the sample schema, and assign the privilege by selecting it and pressing the < button, followed by the **Apply Changes** button.



Alternatively, you could issue the following SQL command:

```
GRANT ALL PRIVILEGES ON sample.* TO 'sample_install'@'%' IDENTIFIED BY '1n5t4ll-Pw' WITH GRANT OPTION;
```



Please note that the GRANT privilege itself does not allow you to create users.

For being able to create users, an administration user has to have write access to database. Unfortunately, this allows for manipulation of *all* user accounts regardless scope; so a user with *mysql* write access is not restricted to managing one separately any more. Because of this, we recommend leaving the creation or deletion of the account to the main administrative user. The installation user is then able to assign the specified privileges.

---

Example: Let the `admin4mysql` account create a user with:

```
GRANT USAGE ON *.* to 'john_doe'@'%' IDENTIFIED BY 'Confidentia
```

The installation user `sample_install` (if he or she has the GRANT privilege) can then grant specific rights to the user at his or her own discretion:

```
GRANT SELECT ON sample.* to 'john_doe'@'%';
```

## See also

- ▶ *Defining an alternative user for administrative tasks*
- ▶ *Avoiding plain text passwords in administrative scripts*
- ▶ *Creating a basic user*

# Creating a read-only account

In the previous recipes, we presented how to define users for different roles: global administration, setting up a specific database, and basic access. Another typical role is the guest user, which typically is limited to read-only operations. Credentials for this account can be passed on to different people without risking accidental or deliberate manipulation. In this recipe, we will show you how to define such a read-only user.

## Getting ready

Think of a catchy username, password, and the name of the database schema for which this user will have read access. In addition, this schema should already exist. You will find the SQL statement alternative from the *There's more...* section of this recipe otherwise.

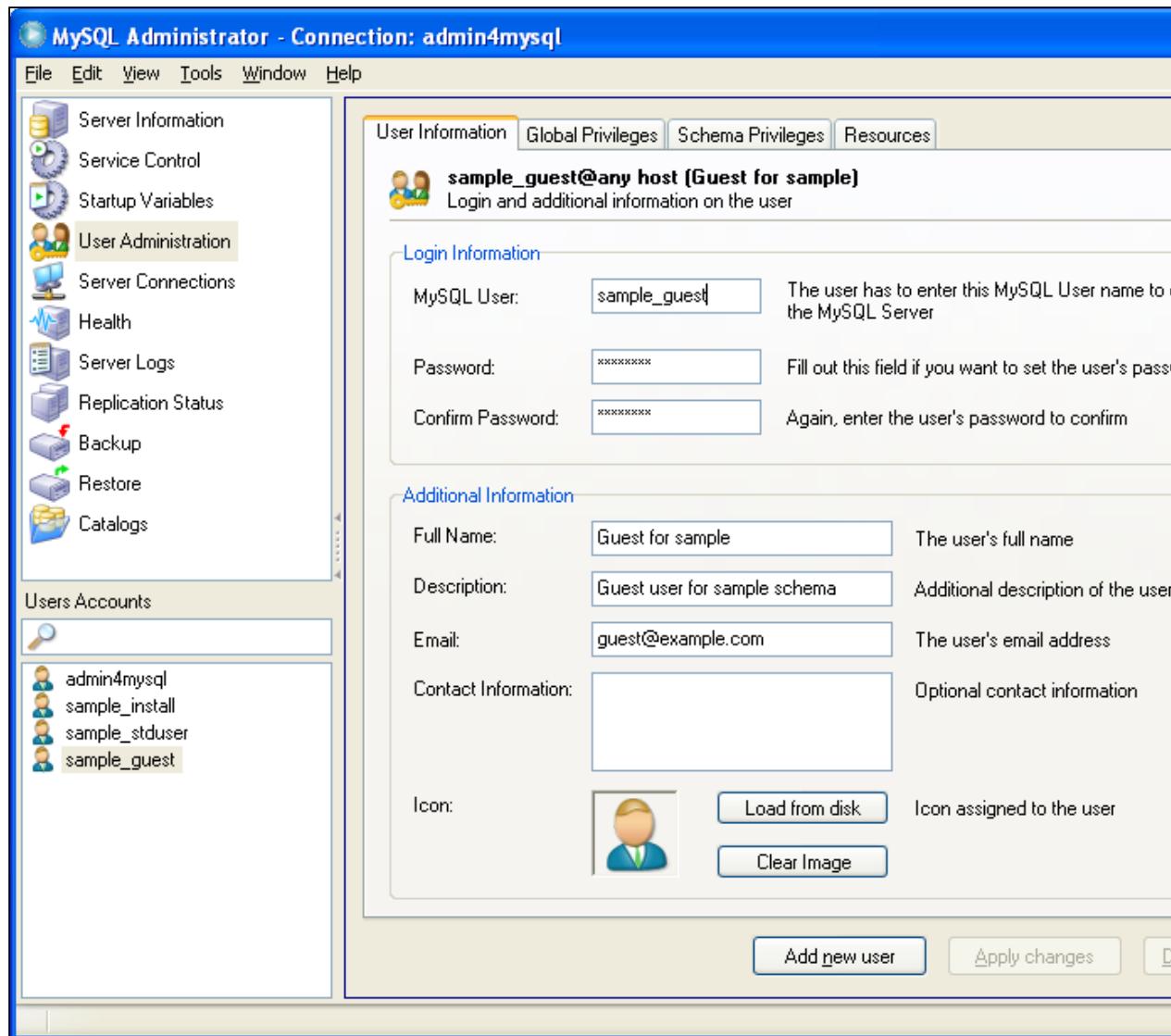
We will assume `sample_guest` as the username, `sample` as the database, and `R3ad0nly` as the password.

## How to do it...

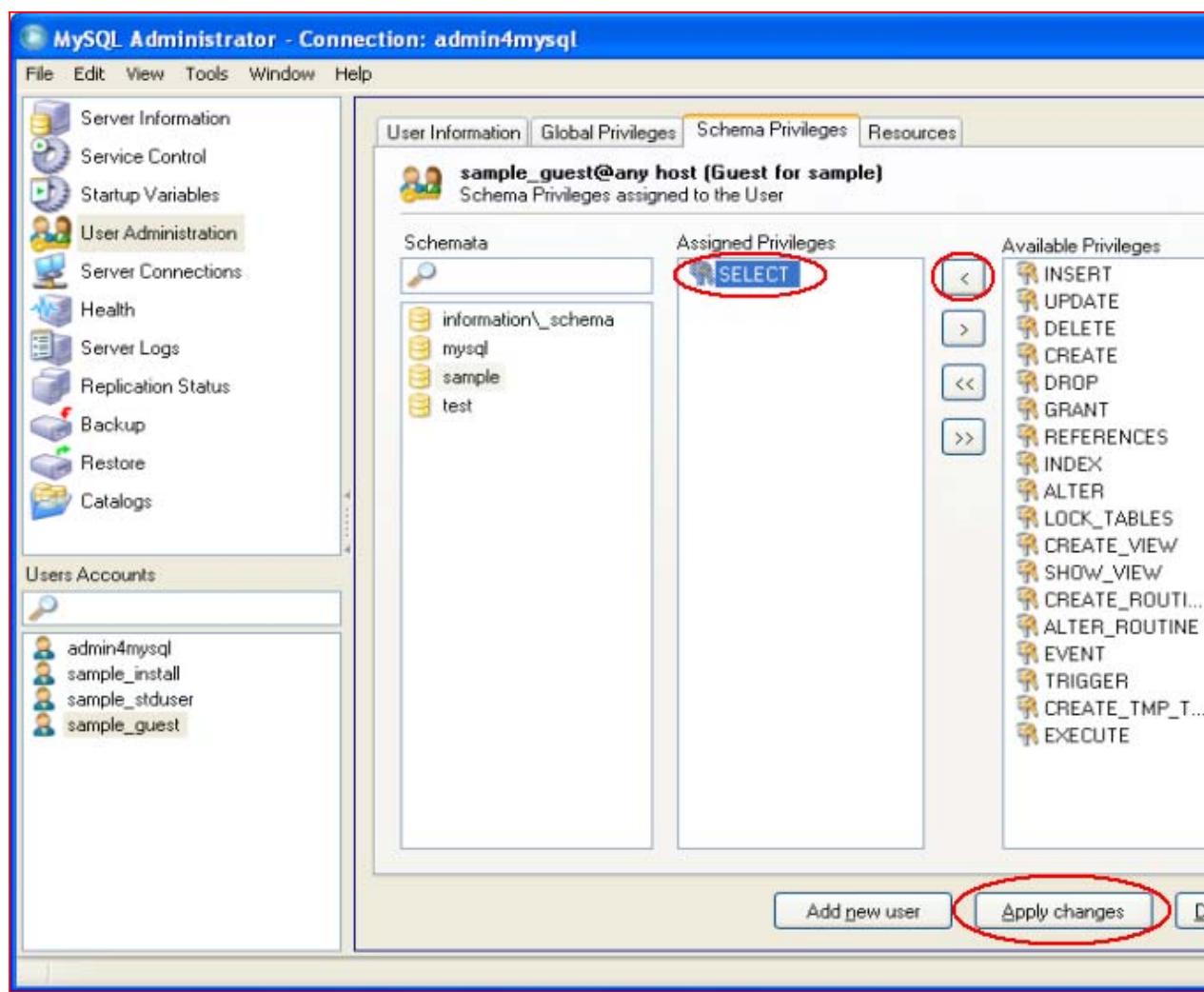
1. Start MySQL Administrator. Connect to your database server using the administrative account (`admin4mysql`).
2. Select the entry **User Administration** either from the list on the left or from the **View** menu.
3. Click on the **Add new user** button.

## MySQL User Management

4. Enter the basic user information (username, password, contact information) by a click on the **Apply changes** button.



5. Select the tab **Schema Privileges** and select the schema *sample* from the list on the left. Click on the **SELECT** privilege from the **Available Privileges** on the right, and press the < button to assign this privilege. Choose the **Apply changes** button afterwards.



## How it works...

Steps 1 through 4 create the user without any privileges; step 5 assigns the **SELECT** privilege to the *sample* database to the user.

With these settings, this user is not able to make any changes to the database. Nevertheless, we strongly recommend not communicating the username and the password for the account too laxly. In many cases, a database contains valuable or sensitive information. Credentials of a guest user account are common knowledge (or too easy to guess, infamous username 'guest' with exactly the same password), your data will be an easy target for any possible intruder.

## MySQL User Management

### There's more...

In this section, we will discuss some advanced aspects of the user creation, such as solutions to the problem and granting rights to create users to certain accounts.

### **Creating the account without using MySQL Administrator**

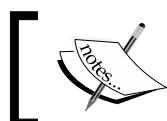
To create the guest account without using MySQL Administrator, execute the following SQL command:

```
GRANT SELECT ON sample.* TO 'sample_guest'@'%' IDENTIFIED BY  
'R34d-Only';
```

This command will also work if the sample database was not created in advance.

### **Allowing stored procedure calls**

Some database designers try to encapsulate complex statements in stored routines or procedures). Some of these routines could also be very helpful for a guest user, she can resort to predefined logic, and does not have to try to construct complex code by him- or herself. For this reason, it might make sense to also assign the EXECUTE privilege to the guest user. But please note that assigning this privilege might have an unexpected side effect:



A guest user with EXECUTE rights can perform changes to the database.

If a stored routine performs changes to the database, like an UPDATE or a DELETE, it will execute flawlessly even if the user who calls this routine does not have any other privilege but EXECUTE. This does not typically pose a serious risk because the guest user is not allowed to define new routines on his or her own. But as soon as there are stored procedures that perform changes to the database, the EXECUTE privilege should be granted with care.

### See also

- ▶ *Defining an alternative user for administrative tasks*
- ▶ *Avoiding plain text passwords in administrative scripts*

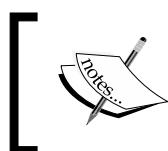
## Defining a specific user for backup

Even though many people have a tendency to ignore the possibility of unpleasant events, one of your duties as a database administrator is to take reasonable precautions to minimize the negative effects of a disaster. In short, it is your job to perform backups.

There are some strategies on how to best back up your database and there are also tools on the market that promise to help you do so. Basically, the different backup types can be divided into two groups: hot and cold backup. While cold backups can simply be copying and saving files, hot backups are not as easy to perform.

There are some tools available that promise file-based hot backups. However, if you need to resort to the MySQL tools, you will typically have to do a database dump (for example using *mysqldump*) to back up your data during normal operations (if all your tables are MyISAM tables, you could use the *mysqlhotcopy* tool instead).

To perform a database dump, you will need a user to connect to your database. We recommend defining a user that is specifically suited for this task. This allows you to identify connections used for backups and you can be sure that these connections are not able to change any data. The following recipe will show you how to create such an account.



Please note that this user is suited for backup purposes only. The recommended task should be performed by an administrator user because you typically need full access to the database for this.

### Getting ready

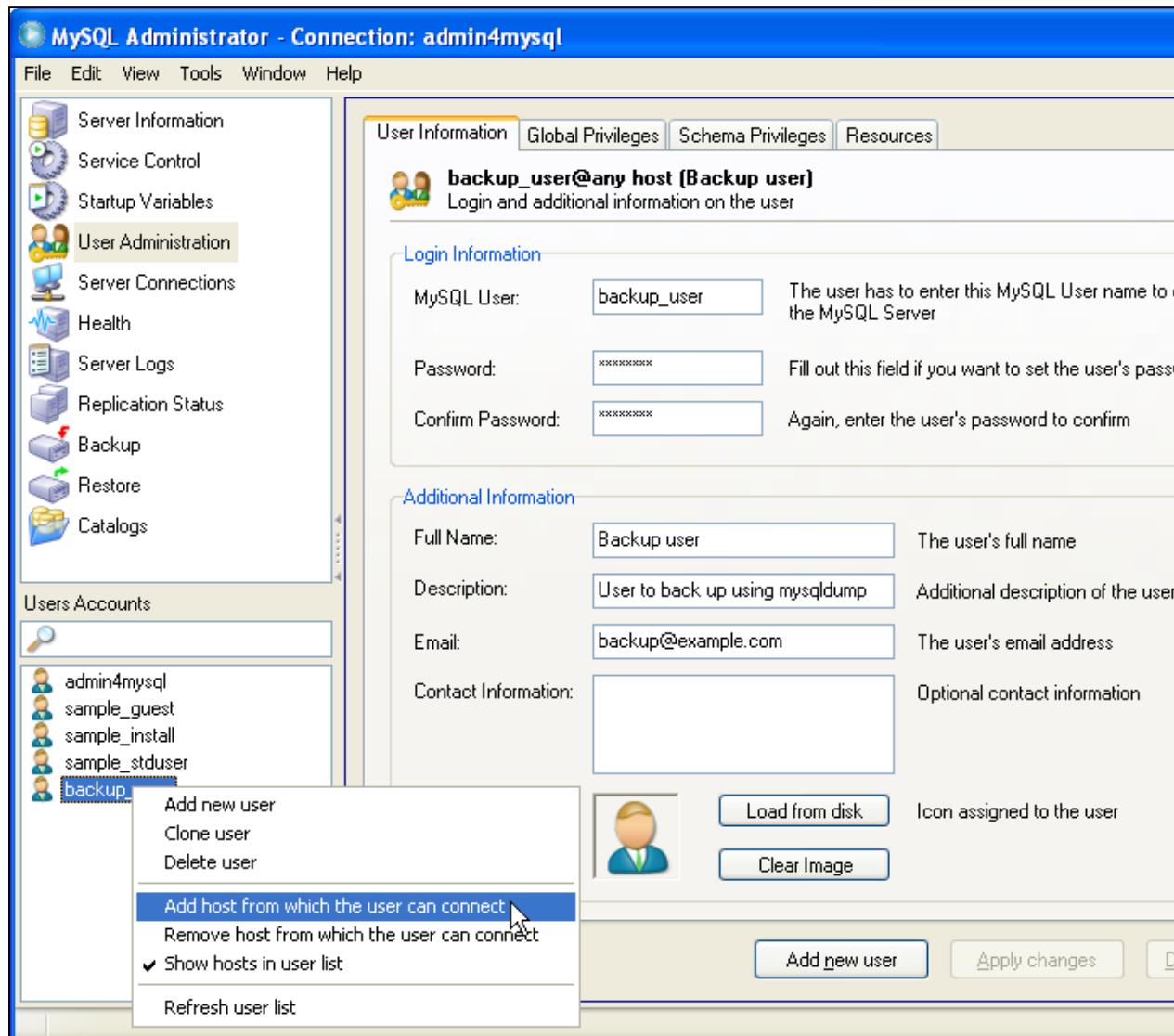
Again, you are going to need a username and a password. We will use *backup\_user* as the username and *B4ckM3Up!* as the password.

### How to do it...

1. Start MySQL Administrator. Connect to your database server using the *admin4mysql* account.
2. Select the entry **User Administration** either from the list on the left or from the **View** menu.
3. Click on the **Add new user** button.
4. Enter the basic user information (username, password, contact information) and click on the **Apply changes** button.

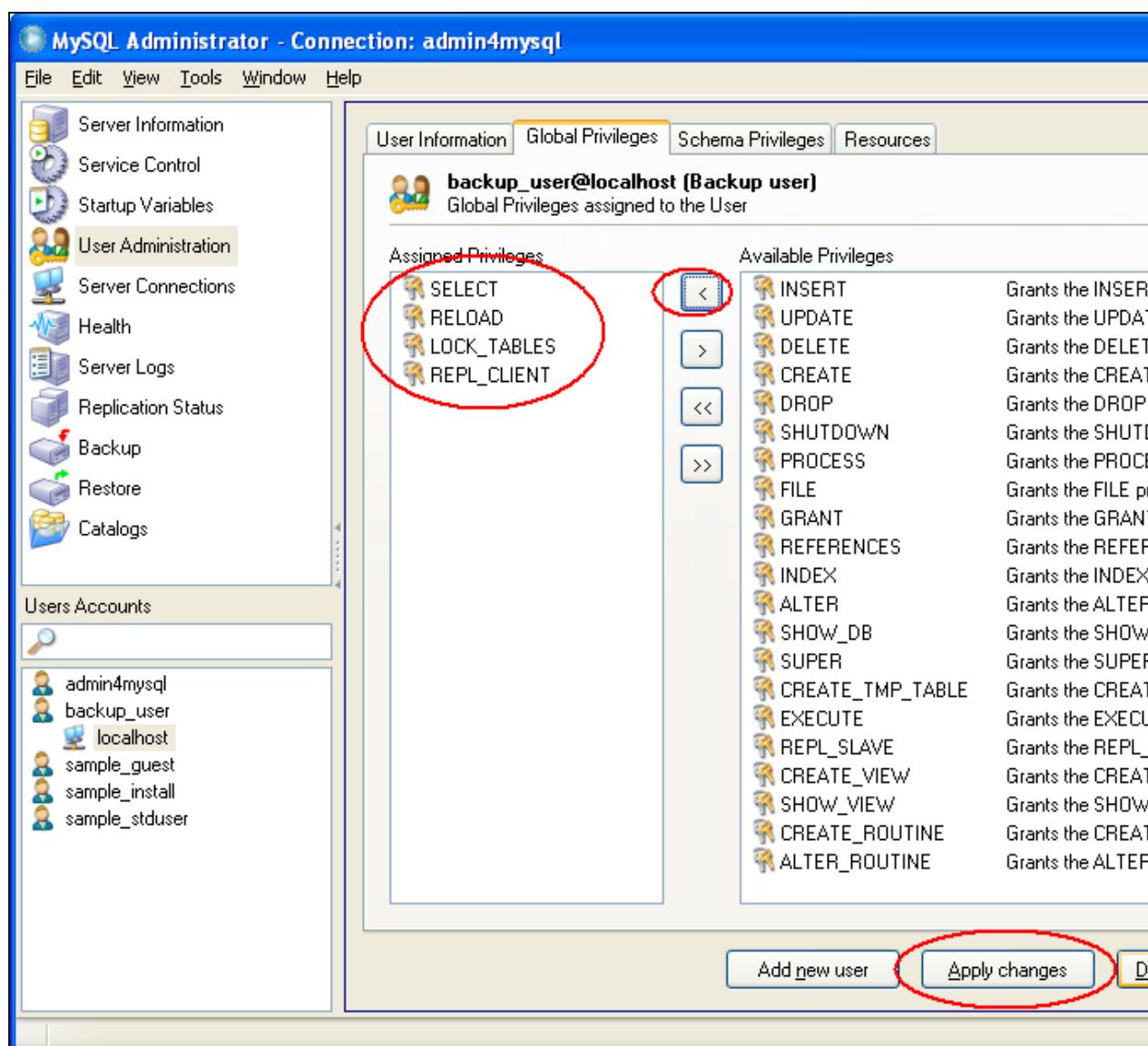
## MySQL User Management

5. Right-click on the new user *backup\_usr* (in the user list on the lower left) and select the option **Add host from which the user can connect**.



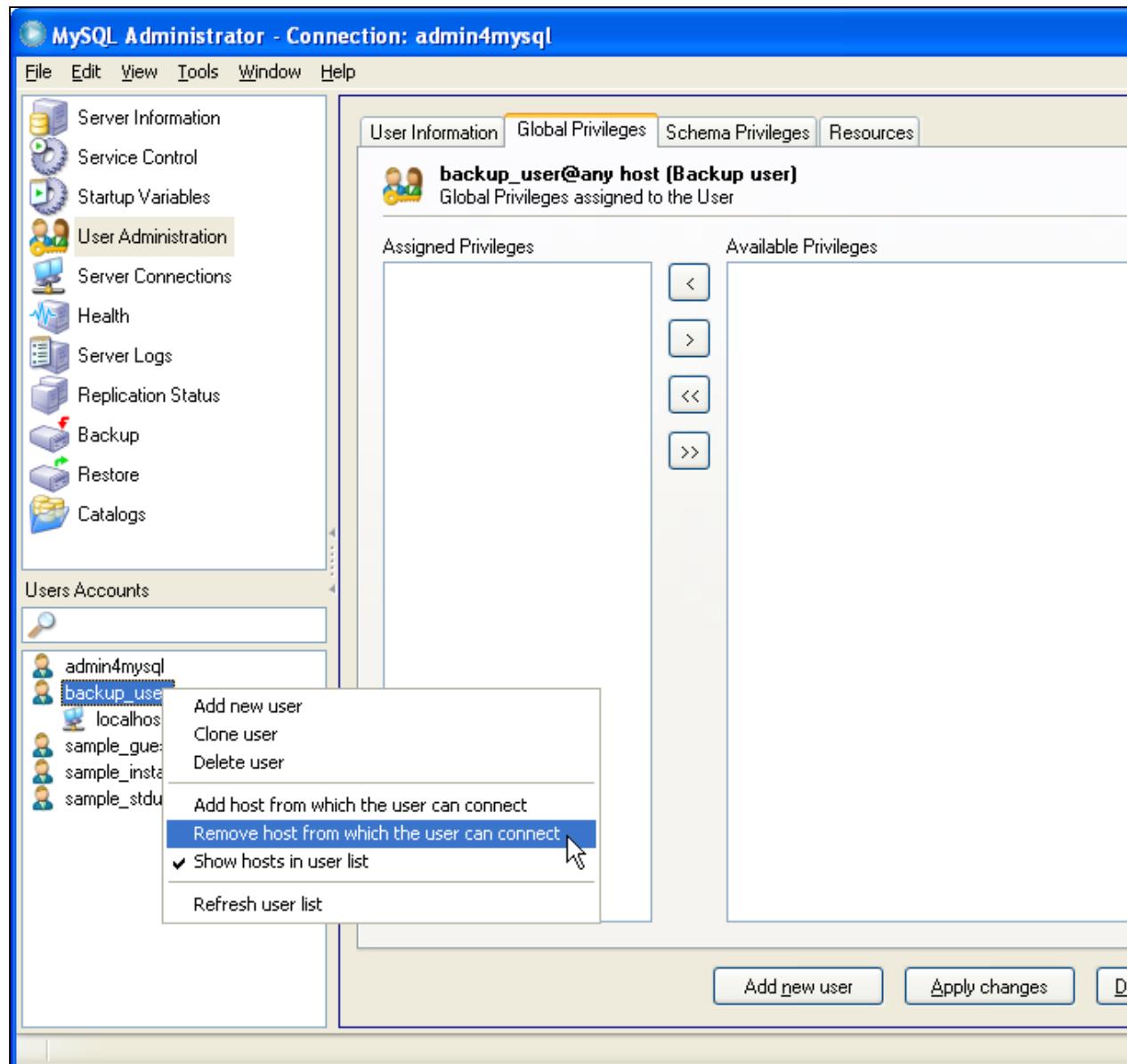
6. In the following form, enter the host from which you are going to perform your backups (typically localhost).

7. Select the tab **Global Privileges**. Choose the **SELECT, RELOAD, LOCK\_TABLES, REPL\_CLIENT** privileges from the **Available Privileges** list on the right, press the **<** button for each of them, and click on the **Apply changes** button.



## MySQL User Management

8. Right-click on the *backup\_usr* entry on the user list and select **Remove host from which the user can connect** from the context menu.



9. Confirm the message box indicating **The any-host (%) entry has been deleted** and click the **Apply changes** button.

### How it works...

Let's take a look at the steps of this recipe. By following steps 1 through 4, a new user *backup\_usr* is created. At this point, this user does not have any rights, but could still connect from any host. Steps 5 and 6 define a specific host (*localhost*) from which this user will be allowed to log on. Step 7 finally assigns the rights that are necessary to perform backups using the *mysqldump* command.

With steps 8 and 9, the user is no longer able to connect from any host other than localhost. This makes an attack harder because a possible intruder would first have to log in to the host itself before he or she can access the database.

The first (and most important) privilege that was assigned is the **SELECT** privilege, used to read the data; otherwise we would not be able to write it to the dump file. As this user is allowed to read all data from all databases, you should not forward the user credentials to this user.

The second privilege is **LOCK\_TABLES**. It is needed because *mysqldump* locks the tables before dumping their data. This privilege is not required when the *mysqldump* options **--single-transaction** or **--master-data** are used, but it is recommended to use it on its own just in case a dump without one of these options has to be done.

Finally, the privileges **RELOAD** and **REPLICATION\_CLIENT** are required for the *mysqldump* option **--master-data**. This option is used frequently, if you have to recover data from a master server that has multiple slaves. If the slaves are configured as replication clients; assigning these privileges allows you to use them as well.

## There's more...

MySQL Administrator is not always the proper tool for user definition (for example, in a command-line environment). An alternative route to constitute an account for backup tasks is the use of the following SQL command:

```
GRANT SELECT, LOCK TABLES, RELOAD, REPLICATION CLIENT ON *.*  
    TO 'backup_user'@'localhost' IDENTIFIED BY 'B4ckM3Up!';
```

## See also

- ▶ *Defining an alternative user for administrative tasks*
- ▶ *Configuring MySQL Administrator to display global privileges and hosts*
- ▶ *Avoiding plain text passwords in administrative scripts*

## Defining a specific user for replication

In some cases, the MySQL replication mechanism is a very helpful feature, for example, to horizontally scale your read loads or to provide redundancy for improved robustness. If you plan to use this feature, you have to define a user on the replication master for this. This recipe will show how to create this user.

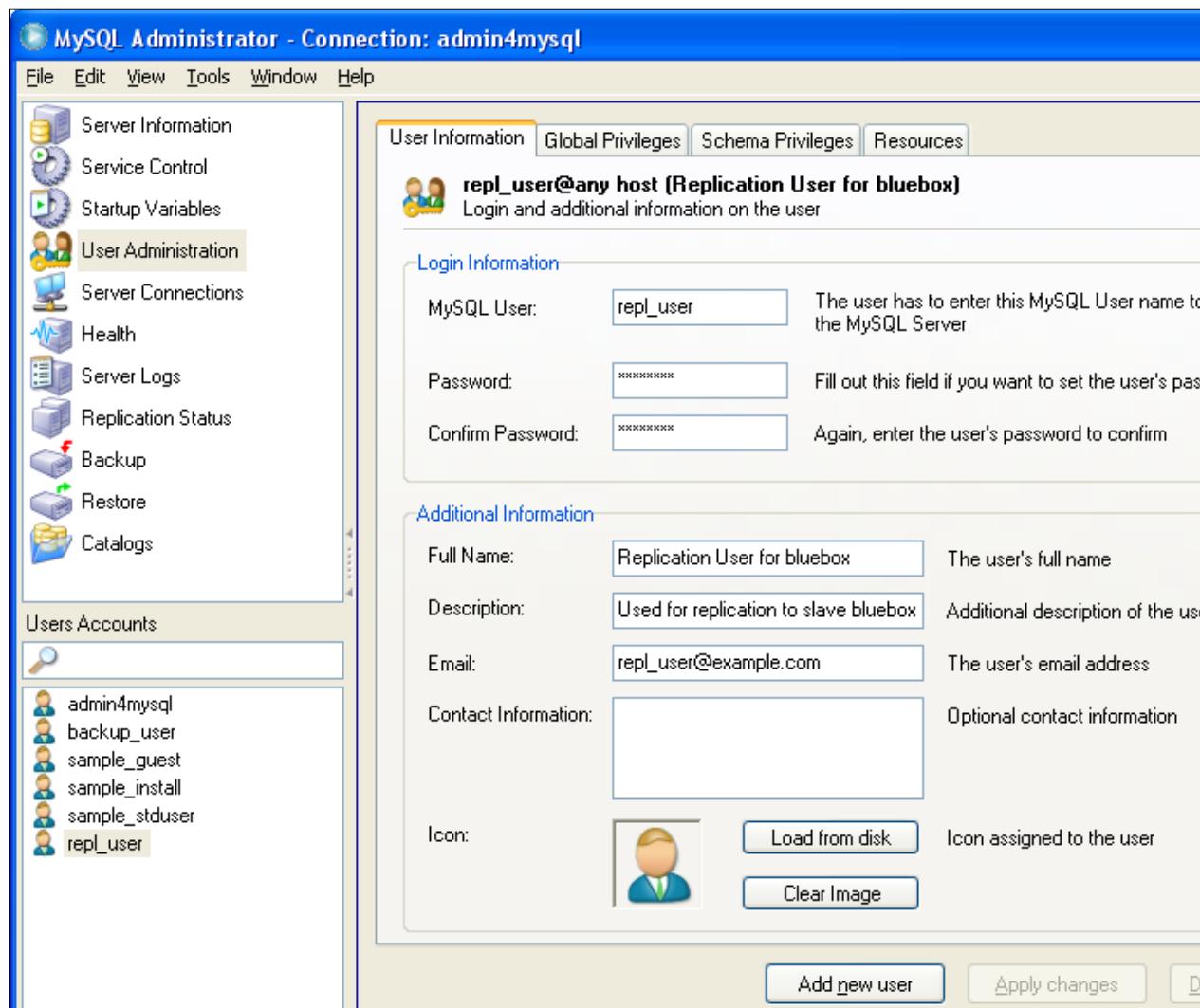
## MySQL User Management

### Getting ready

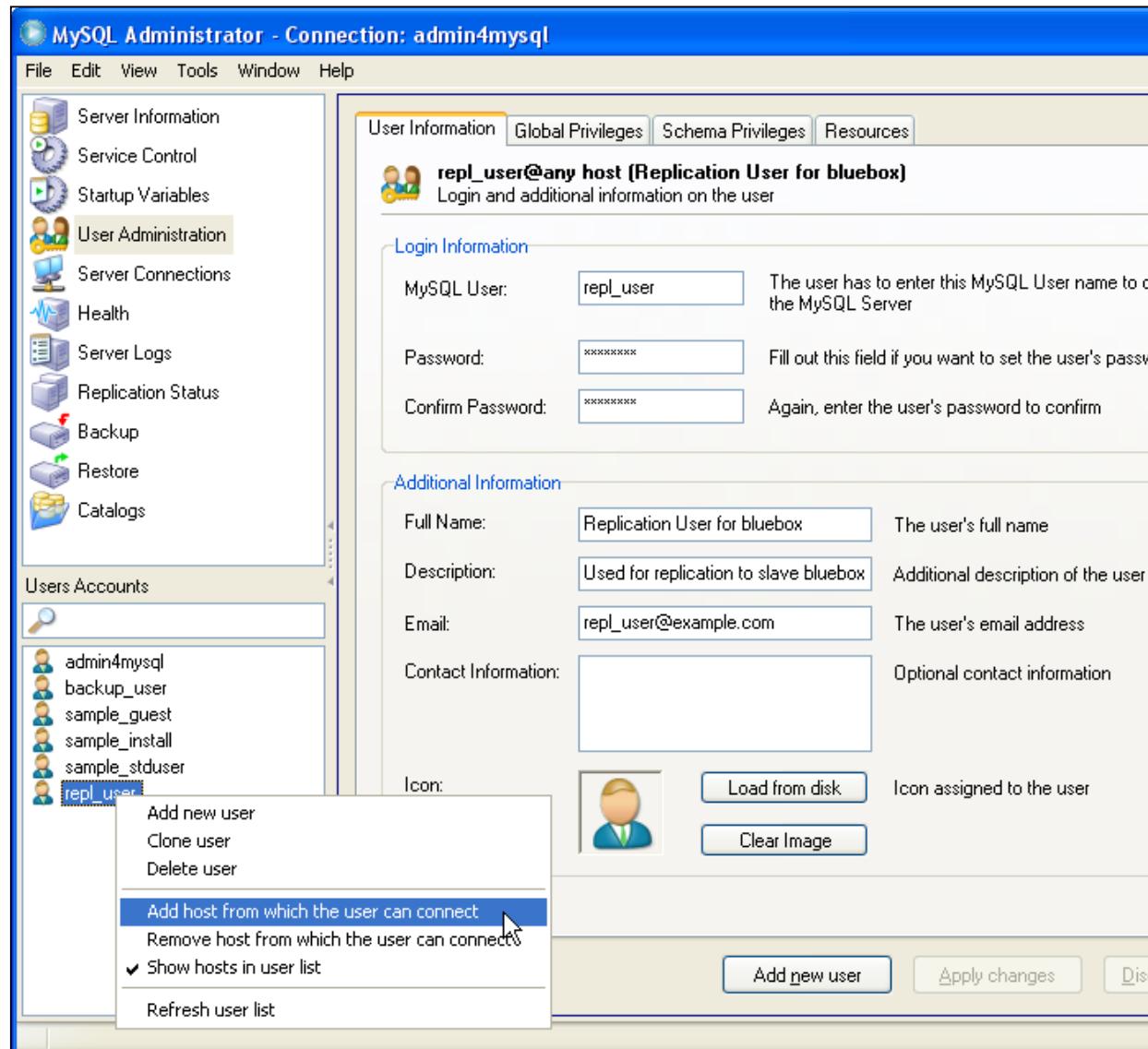
To step through the recipe, you will need a username, a password, and the host on which the replication slave will be located. We will use `repl_user` as the username with `Pw_4_R` as the corresponding password and we will assume that the replication slave will be located on host `bluebox`.

### How to do it...

1. Start MySQL Administrator. Connect to your database server using the `admin4mysql` account.
2. Select the entry **User Administration** either from the list on the left or from the **View** menu.
3. Click on the **Add new user** button.
4. Enter the basic user information (username `repl_user`, password `Pw_4_R`), followed by a click on the **Apply changes** button.

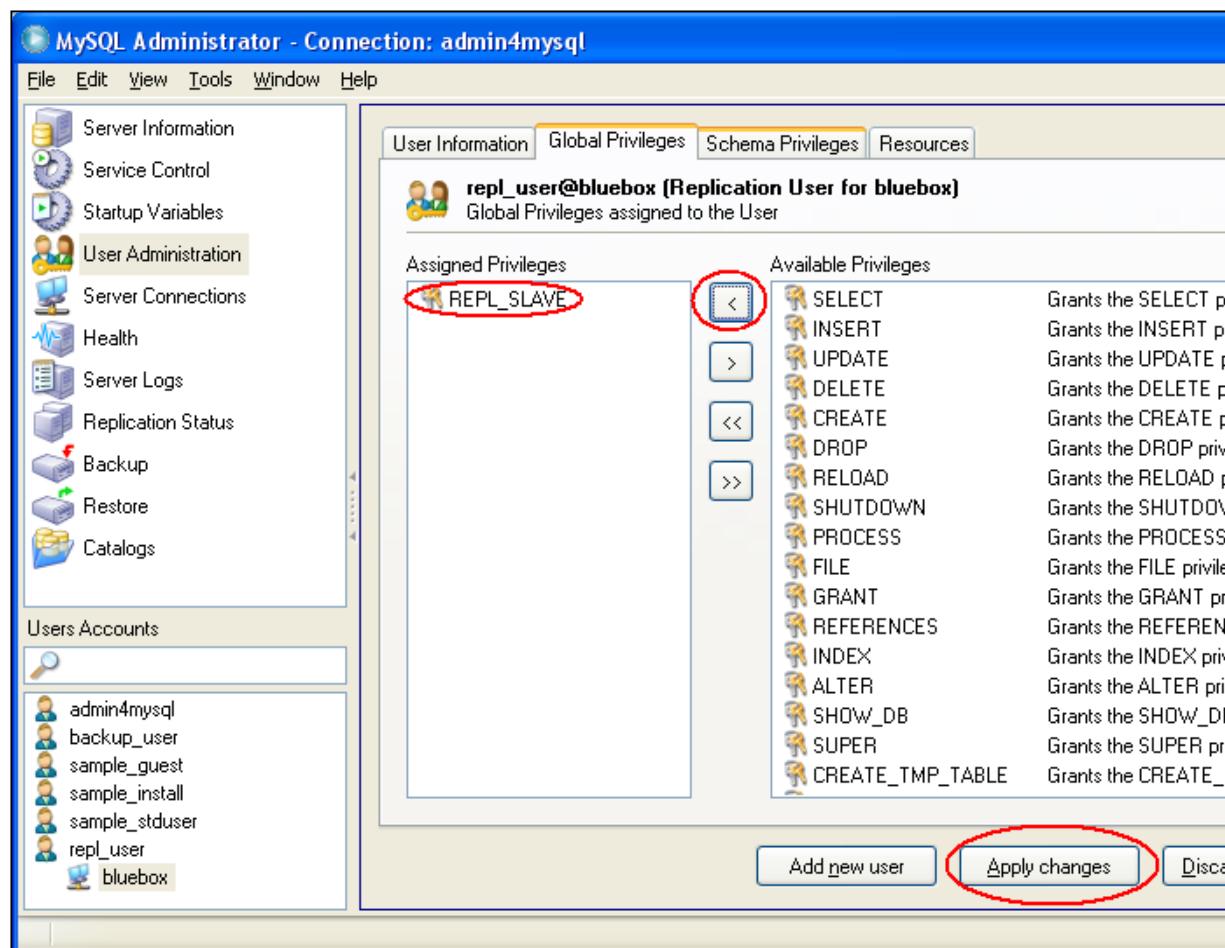


5. Right-click on the new user *repl\_user* (in the user list on the lower left) and the option **Add host from which the user can connect:**

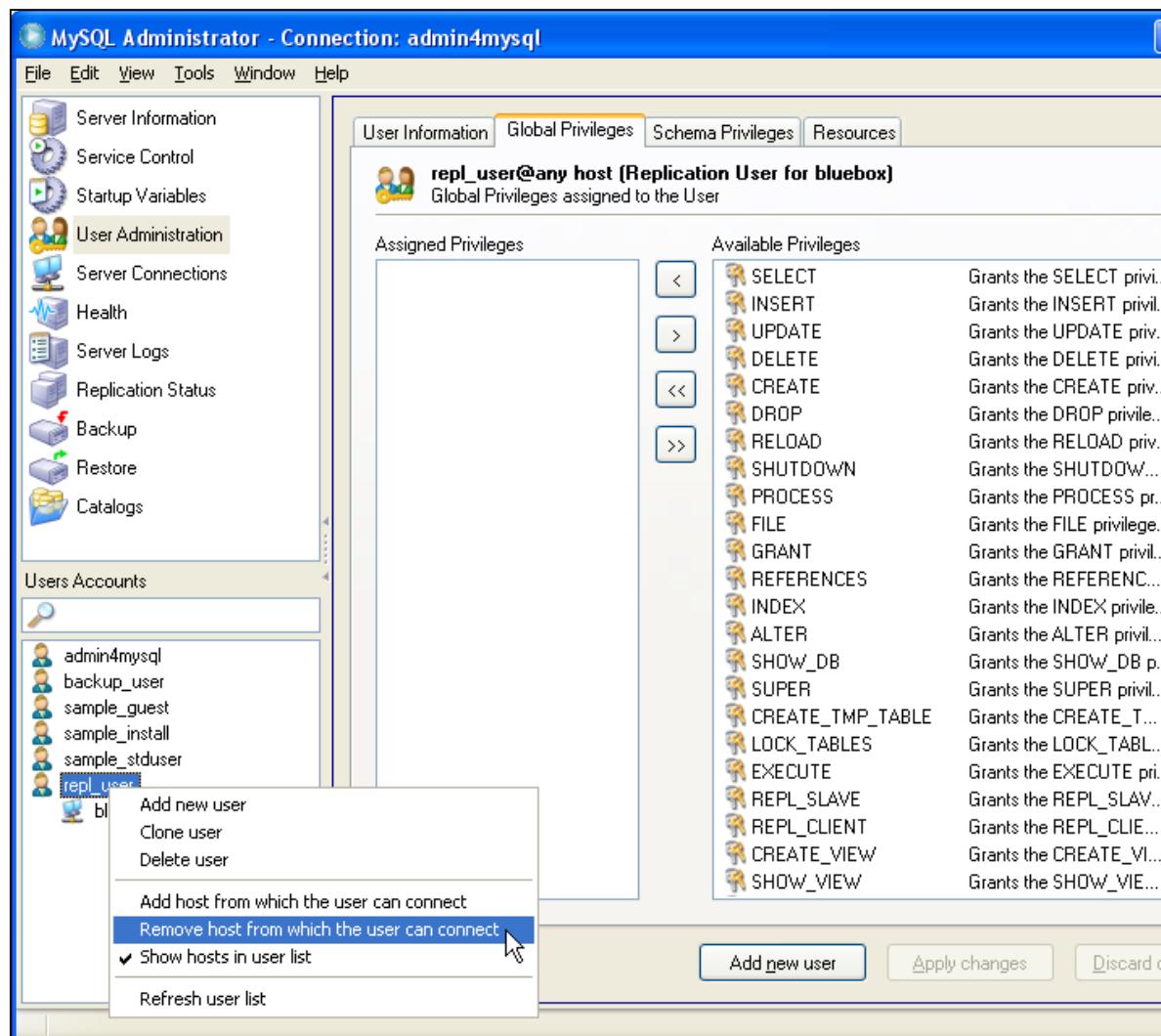


## MySQL User Management

6. In the following form, enter the host from which you are going to perform your backups (here: bluebox).
7. Select the tab **Global Privileges**. Choose the **REPL\_SLAVE** privilege from **Available Privileges** list on the right, press the < button, and subsequently on the **Apply changes** button.



8. Right-click on the *backup\_usr* entry on the user list and select **Remove host from which the user can connect** from the context menu.



9. Confirm the message box indicating **The any-host (%) entry has been deleted** and click the **Apply changes** button.

## How it works...

Let's inspect the steps of the above recipe. In steps 1 through 4, we defined a new user named *repl\_user*, which at this point has no privileges at all, but could connect to the server from any host. With steps 5 and 6, we define the specific host from which the user will be allowed to log on (in this case this is the host that runs the replication slave). Step 7 assigns the privilege necessary for the replication slave to the new account.

With steps 8 and 9, the account *repl\_user* is changed in such a way that it can no longer be used from any host (actually, the account *repl\_user@%* is removed, while a second account *repl\_user@bluebox* stays intact). A possible attacker would have to log in to the replication slave host to use the account.

## MySQL User Management

### There's more...

To create the replication account without resorting to a GUI tool, you can alternatively use the following SQL command:

```
GRANT REPLICATION SLAVE ON *.* TO 'repl_user'@'bluebox' IDENTIFIED BY 'Pw_4_R3pl';
```

### See also

- ▶ *Defining an alternative user for administrative tasks*
- ▶ *Configuring MySQL Administrator to display global privileges and hosts*
- ▶ *Avoiding plain text passwords in administrative scripts*

## Allowing access from specific hosts on MySQL

When creating a user account using MySQL Administrator, the user is by default enabled to log in from any host. If the account will be used on specific clients only, it is advisable to restrict the account in such a way that login is restricted to these clients. This helps to reduce the chance of successful attacks against your database server because a possible intruder would not only have to get hold of (or guess) the proper credentials, but also has to seize control of one of the registered clients.

The following recipe will guide you through the steps of restricting a user account to log in from a specific host.

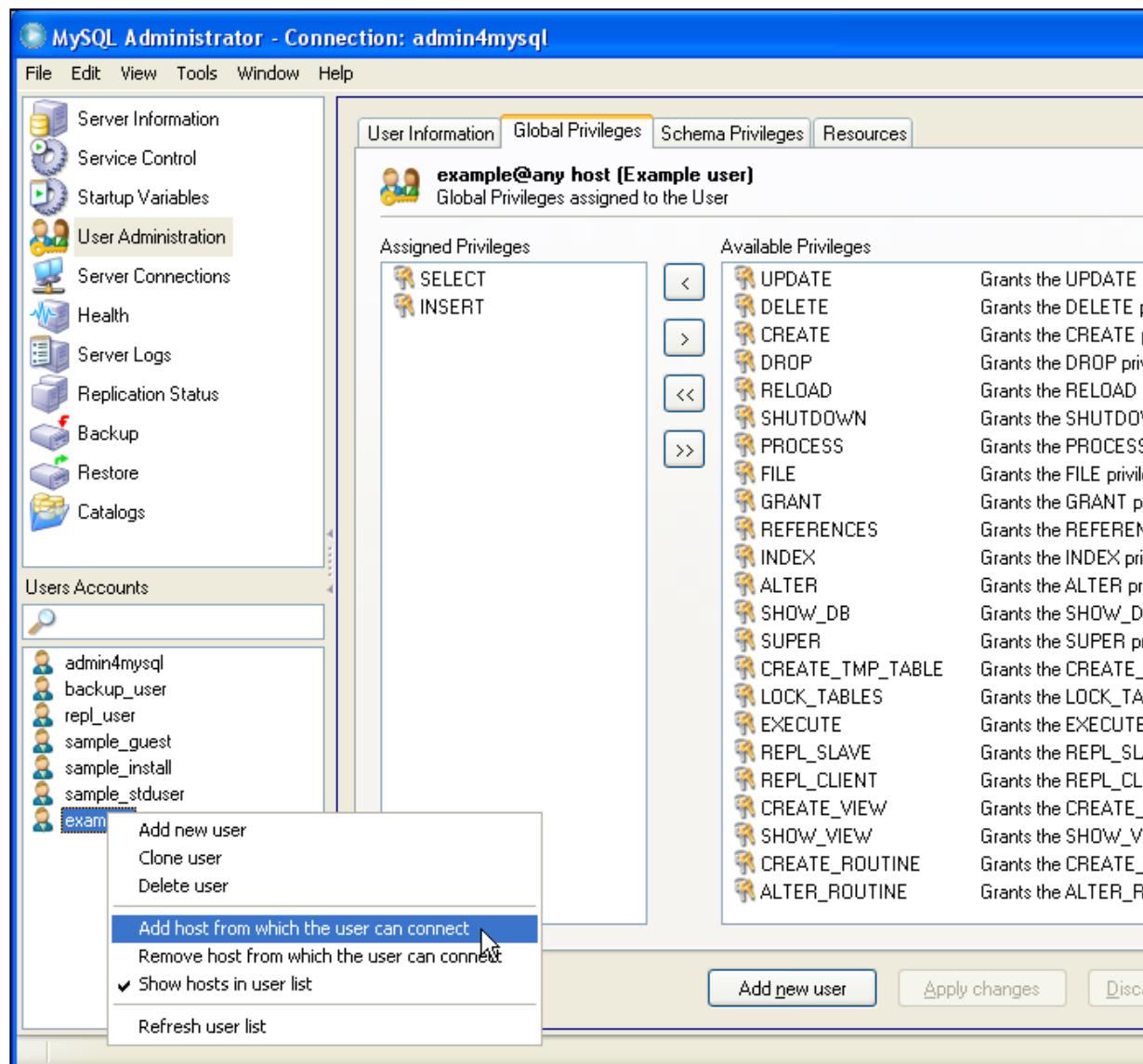
### Getting ready

For the following steps, we assume that an account with username `example` has already been defined and that this account is enabled for login from any host. We furthermore assume that this account should be changed in such a way that it can only be used to log in from the host `client1.mycompany.com`.

### How to do it...

1. Start MySQL Administrator. Connect to your database server using the `admin4mysql` account.
2. Select the entry **User Administration** either from the list on the left or from the **View** menu.

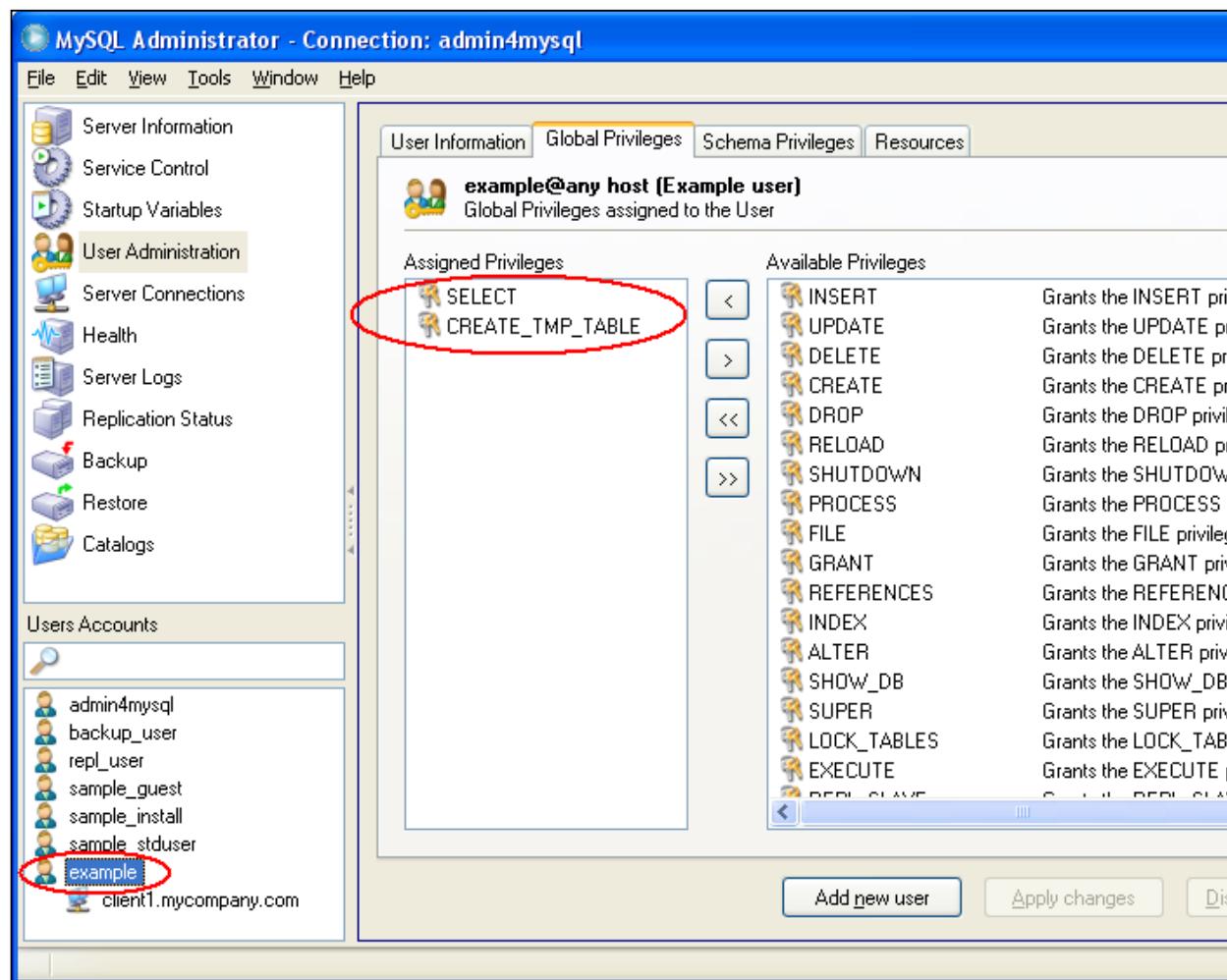
3. Right-click on the user account you want to change in the **Users Accounts** left and choose the option **Add host from which the user can connect**.



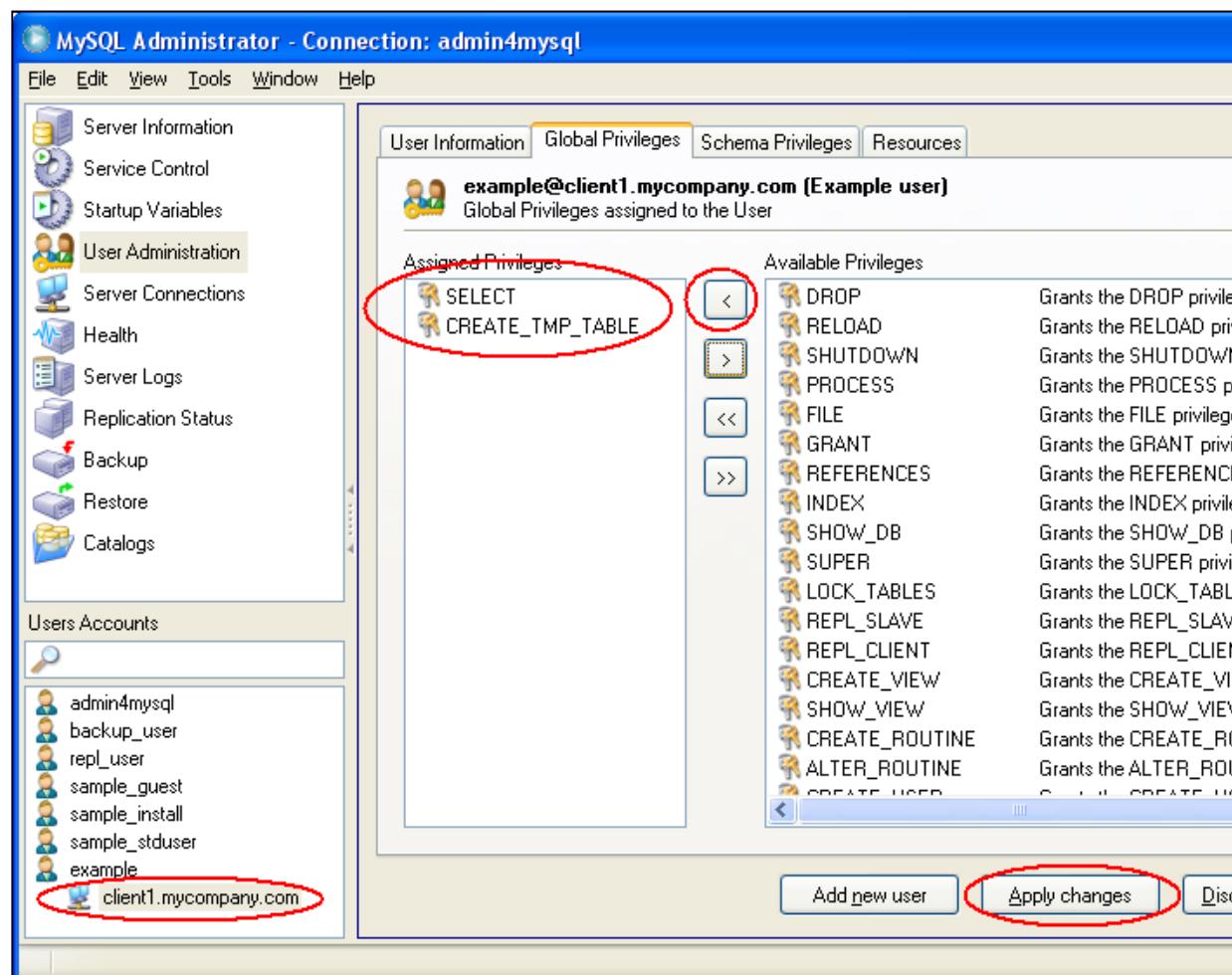
4. In the following form, enter the host from which the user will be able to connect (here: client1.mycompany.com).  
 5. Save the changes by clicking on the **Apply changes** button.

## MySQL User Management

- Select the tab **Global Privileges**. Left-click on the user entry **example** on the lower left. Note the privileges that are listed in the **Assigned Privileges** example screen, we assume that global privileges **SELECT** and **CREATE TABLE** are assigned to the user).



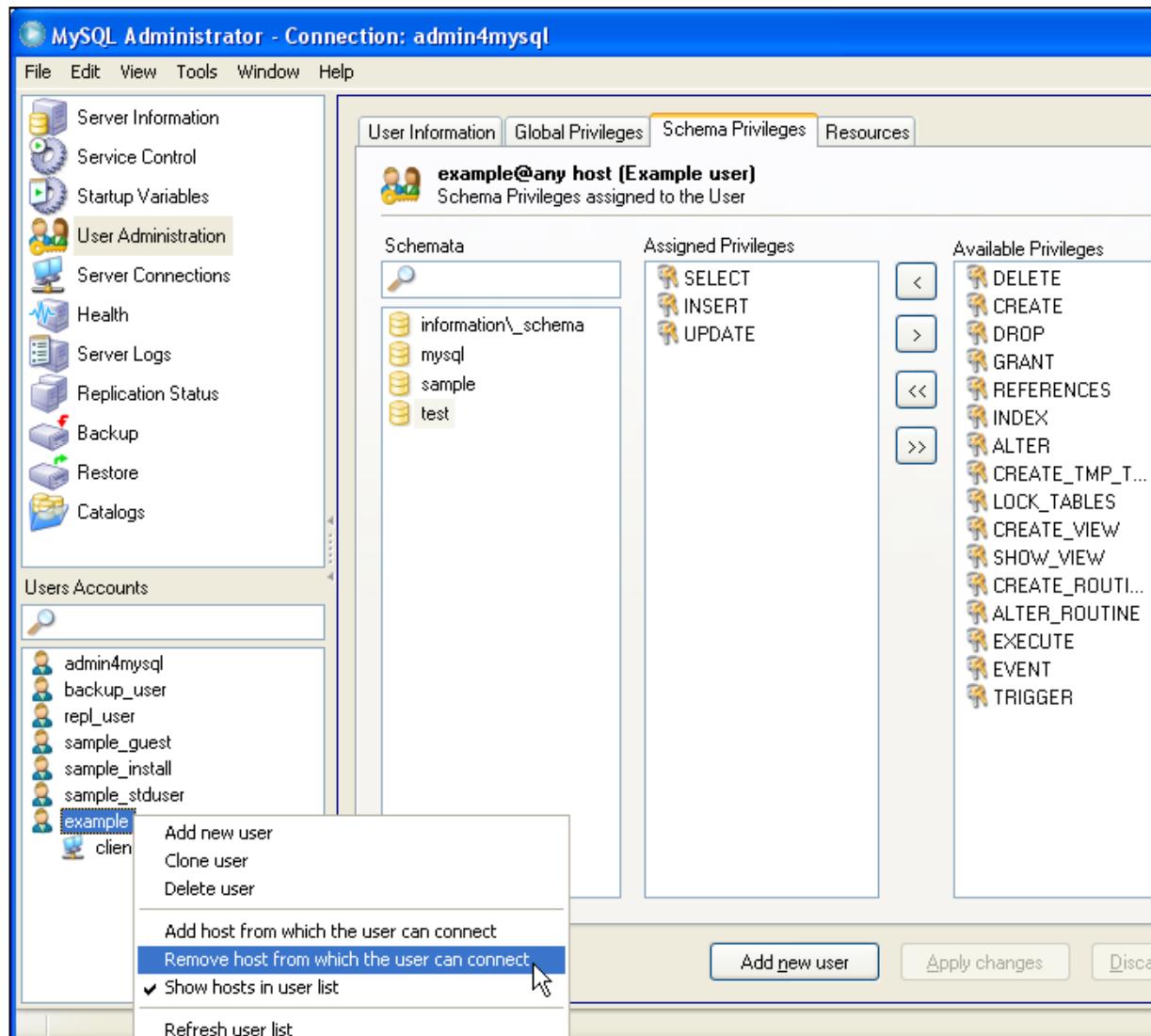
- Switch back to the new host by clicking on the host name `client1.mycompany.com` on the left. Assign the same privileges noted in the previous step (in our example: **SELECT** and **CREATE TEMPORARY TABLE**) by selecting them from the available privileges on the right and selecting the < button, followed by the **Apply changes** button.



8. Choose the tab **Schema Privileges** and select the first schema from the Schemata list.
9. Select the entry **example** from the **Users Accounts** list on the lower left. Now privileges that are assigned for the selected schema.
10. Switch to the new user by selecting the host name **client1.mycompany.com** on the left. Assign the privileges noted in the previous step (by selecting them from the list on the right and clicking the < button), then choose **Apply changes**.
11. Select the next schema and repeat steps 9 and 10 for every entry in the Schemata list.

## MySQL User Management

12. Right-click on the **example** entry on the user list and select **Remove host the user can connect** from the context menu.



13. Confirm the message box indicating **The any-host (%) entry has been deleted** and click the **Apply changes** button.

### How it works...

Let's have a look at what we did throughout this recipe.

In steps 1 through 5, we added the new host to which the account should be restricted more precise, we created a new account `example@client1.mycompany.com`.

Steps 6 and 7 are performed to copy the existing global privileges, while steps 8 through 10 do the same for the schema-level privileges. With these operations, the user will be granted the same permissions when logging in from the newly-defined specific host (this request does not take into account any schema object privileges).

---

Within steps 12 and 13, the permission to log in from any host is withdrawn by deleting the account (technically speaking, the user `example@%` is deleted).

With these steps, we created a copy of the existing `example` account, which is restricted to a specific host, and disabled the possibility to log in from any other machine.

### There's more...

The preceding recipe can be used interactively only and it describes definition of single hosts only. Next, we will show you how to create a user by using a script and how to introduce multiple hosts.

### **Creating the account without using MySQL Administrator**

If you want to restrict access of an account to a client machine, but you do not want to use MySQL administrator for this, the following steps lead to identical results:

1. Connect to your database server as an administrative user (`admin4mysql`) using your favorite SQL client (for example, MySQL Query Browser).
2. Find out about the current privileges by issuing the following command:

```
mysql> show grants for 'example'@'%';
```

3. The result of the above query will be a list of one or more SQL statements. To change these statements by replacing the username '`example'@'%'` by the new one '`example'@'client1.mycompany.com'`.

4. Execute each of the modified statements from the previous step (you will have to add a semicolon at the end of each statement).

5. To deny login from any host, execute the following command:

```
mysql> drop user 'example'@'%';
```

### **Allowing access from a group of hosts**

The above recipe is suited for accounts that are restricted to log in from a specific client machine. In some cases, however, you do not want to allow access from any host, but rather from a group of hosts. It is also possible to restrict connections to a single client machine either.

As long as the number of clients is reasonably low, it is possible to individually register them and grant them respective machines. For this, the mechanism outlined in the above recipe can be used. In this case, you will have to adapt steps 6 and 9 by selecting the appropriate template user and you will have to skip steps 12 and 13.

As soon as the number of clients exceeds a certain limit, it is no longer feasible to list all of them separately. For these cases, the wildcard feature of MySQL might be a valid alternative for you.

## MySQL User Management

---

MySQL allows the use of wildcard operators when defining host names of user accounts. For example, if you define a MySQL account with host client1.mycompany.com through client56.mycompany.com, assume you have a set of client machines client01.mycompany.com through client56.mycompany.com. These clients should be able to access the database, but you don't want to allow access from any other machine. In this case, you would restrict the access by defining the host as client% .mycompany.com (note the % character). The percent sign is a wildcard that can be replaced by any character string. With this definition, login from client57.mycompany.com will be successful, but a connection attempt from a machine other than client57.mycompany.com will be refused.

Please note that the use of wildcards does not necessarily allow for the exact restriction that you might want to define. In the above example, there are some unwanted machines that would still be able to access the database: a machine named clientfake.mycompany.com would match the host definition as well. To minimize the wrong matches, you could use the host client\_\_ .mycompany.com instead (with \_ being a placeholder for exactly one additional character), but a machine clientXY.mycompany.com would still be able to establish a connection.

The host definition for MySQL user accounts can also be used to provide IP addresses instead of machine names. If you are able to assign similar IP addresses to all machines that should be able to access your host, you can provide a host name like 192.168.1.% . This value would enable access for all clients with an IP address from 192.168.1.0 through 192.168.1.255.

The use of the single character wildcard is possible with IP addresses as well. A host name of 192.168.1.1\_\_ will allow login from clients with IP addresses 192.168.1.100 through 192.168.1.199.

An alternative to wildcards is to group possible clients in a common subnet. You can make use of a notation that allows defining IP ranges using the base address of the subnet combined with the corresponding netmask. If you set the host of a MySQL account to <subnet prefix>/<subnet mask>, connections from any IP address within the range will be accepted by MySQL.

If, for example, you want to allow login from four client machines whose IP addresses fall from 192.168.1.100 to 192.168.1.103, you could set up an account with the host 192.168.1.100/255.255.255.252, which will enable login for just the range you intend.

This behavior is somewhat contradictory to the MySQL manual, which states:

*"The netmask can only be used to tell the server to use 8, 16, 24, or 32 bits for an IP address." (See <http://dev.mysql.com/doc/refman/5.1/en/account-names.html>.)*

---

If this statement were true, the only possible subnet masks would be 255.0.0.0, 255.255.255.0, and 255.255.255.255 (in the latter case the subnet mask would be refused for the sake of brevity). However, experience shows that other subnet masks work well, but you have to make sure that the IP address given before the subnet mask is the subnet prefix. Any address from within the subnet would not work, as all connections would be refused.

To clarify that, we will inspect the example from the MySQL manual, which tries to prove that different netmasks do not work. The manual states that a host value 192.168.0.1/255.255.255.240 (28 bits) will not work, which is basically correct. The reason for that is **not** that handling of a 28 bit netmask (255.255.255.240) is unsupported, but that the IP address 192.168.0.1 is an address from the subnet and not the subnet prefix itself. The correct subnet prefix would be 192.168.0.0, and with a subnet mask of 192.168.0.0/255.255.255.240, every client from the subnet—with an IP address from 192.168.0.0 to 192.168.0.15—would be accepted for connections.

Generally, if you define the host with the `<subnet_prefix>/<subnet_mask>` notation, any client address is tested against this value according to the following rule (the `&` symbol stands for the bitwise AND operation).

If the following equation is true, the client address is accepted: `<client_address> & <subnet mask> = <subnet prefix>`.

## See also

- ▶ *Defining an alternative user for administrative tasks*
- ▶ *Configuring MySQL Administrator to display global privileges and hosts*
- ▶ *Avoiding plain text passwords in administrative scripts*

## Regaining access to your database in case of lost account information

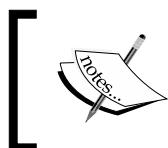
Of course, you want to protect your database against attacks in every possible way. But from time to time you might find yourself in a situation in which you have to act as your own savior—yourself—for example, if you forgot the user credentials for the administration user or if you accidentally deleted the root user account without having created an equivalent user beforehand. But do not worry. We will show you a way to regain control of your database without losing any data (not even your existing user accounts).

## MySQL User Management

### Getting ready

This recipe involves steps to edit the MySQL configuration files and to restart your server. Therefore, you will need access to the host your database runs on. You also have the rights to start and stop the MySQL instance at your own discretion. Furthermore, you know the location of the MySQL configuration file (typically `my.ini`) and you should have the rights to change this file.

This recipe allows you to both create a new user and to change the password of a user that already exists. In our example, we will change the password of the user `admin4mysql` to `As,ysp4M` and we will create a new user `root` with `r00t_pw` as the password.



Before executing the following instructions, you should make sure that no other users or processes can access your database server.

### How to do it...

1. Open a text editor and create a file with the following content:

```
SET PASSWORD FOR 'admin4mysql'@'localhost' =
    PASSWORD ('As,ysp4M') ;
GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' IDENTIFIED
    BY 'r00t_pw' WITH GRANT OPTION;
```



Make sure that the `SET` command and the `GRANT` command are each written on a single line.

2. Save the file in a location of your choice (for example: `C:\temp\mysql-init.sql`).
3. Open the MySQL configuration file (for example: `C:\Program Files\MySQL\MySQL Server 5.1\my.ini`) in a text editor. Find the line that reads `[mysqld]` and add the following line below (if a line starting with `init-file=` already exists, change it accordingly):

```
init-file="C:/temp/mysql-init.sql"
```

4. Use the location of the file you saved in step 2. Please note the use of simple slashes instead of backslashes.
5. Save the MySQL configuration file.
6. Restart your running MySQL instance (for example, by restarting the Windows service `net stop MySQL & net start MySQL`).
7. Connect the MySQL server to the file you just modified and run the following command:

## How it works...

The file created in steps 1 and 2 contains the SQL commands by which the password for the user `admin4mysql` is reset and a user `root` is created. In the case of lost account information for the administrative user, these commands can't be executed as usual (for example via MySQL client). We change the MySQL configuration in steps 3 and 4 in order to have these commands executed on the next MySQL start despite the lost credentials. Step 5 is a restart of the MySQL server, which causes the commands to be executed. Step 6 removes the configuration changes of steps 3 and 4, so the command file won't be repeatedly executed every subsequent MySQL start.

The content of the initialization file as listed in step 1 has to be adapted according to specific needs. As you might have guessed already, this mechanism is also suitable for your actual database content, not only user information. It allows for basically everything imaginable to your database, so you should make sure that access to the MySQL database file is restricted to authorized persons. The `init-file` option might be used otherwise to tamper with your data, create user accounts for future unauthorized access, or render the system inoperable.

## There's more...

A different alternative to reset passwords is to start your MySQL database server using the option `--skip-grant-tables`. With this setting, any login will be successful and any privilege will be granted—regardless of the user credentials specified. In fact, this option is often referenced as the preferred way to change your user accounts without the need for entering credentials. Nevertheless, we strongly recommend not using this approach, as there are several limitations and drawbacks.

While the `--skip-grant-tables` approach enables you to reset a forgotten password, it does not allow creating a new user account right away, as any attempt to grant rights via the `GRANT` statement will at first be refused with reference to the disabled grant tables:

```
MySQL> GRANT ALL PRIVILEGES ON *.* TO test@'%';
ERROR 1290 (HY000): The MySQL server is running with the --skip-grant-tables option so it cannot execute this statement
```

## MySQL User Management

---

You have to issue a `FLUSH PRIVILEGES` statement first in order to have MySQL accept a `GRANT` command. While this is a way to create a new user account with full rights, it is a bit cumbersome.

More importantly, use of the `--skip-grant-tables` is a major security issue because as long as the database runs with this option enabled, anybody can connect to the database without having to provide any credentials at all, which enables free access for all potential intruders. While this might not be critical for your personal development database, it is absolutely intolerable for production use or other databases with sensitive data. To prevent unauthorized access while the grant tables are disabled, it is strongly recommended to accompany this configuration with the option `--skip-networking` (on Windows machines, you have to additionally provide one of `--shared-memory` or `--named-pipe` options). Unfortunately, this actually deactivates access for most regular clients as well, which is equivalent to a service interruption until the database is relaunched in the regular configuration.

In conclusion, the `--skip-grant-tables` option makes it necessary to restart the database twice (as opposed to once with the `--init-file` variant) and there will be a service interruption between the two restarts due to the network cut-off that is needed for security reasons (while the `--init-file` approach allows for continued access for the moment of the first restart). This is why we strongly recommend using the method described in this recipe.

## Avoiding plain text passwords in administrative scripts

In many of the previous recipes, we also showed how to define user permissions using `CREATE USER` commands. As such statements can easily be executed in scripts, these are well suited to produce scripts for automated definition of MySQL accounts. While it is generally a good idea to reduce the manual tasks in database administration, one should keep in mind that such scripts contain information that is extremely useful for possible attackers. One can define users, their specific rights, the hosts from which access is granted, and type in the corresponding passwords. In this way such user definition scripts often contain the information necessary to access your database, which makes them extremely sensitive. The risk that somebody coincidentally trips over such a script should not be underestimated, as an open door may tempt a saint.

The most critical portion of such scripts is of course the **passwords**. In this recipe, we will show how to create user definition statements without the need to give a plain text password.

## Getting ready

To step through this recipe, you will need a running MySQL database server and a script that defines the accounts (using plain text passwords). No other prerequisites are required.

## How to do it...

1. Connect to your database server with the MySQL command-line client.
  2. For a plain text password in your script, give the following command and note the encoded result:
- ```
SELECT PASSWORD ('<Your plain text password>');
```
3. Replace the IDENTIFIED BY '<Your plain text password>' portion of your script by IDENTIFIED BY PASSWORD '<Encoded result>'.
  4. Repeat steps 2 and 3 for all passwords in your script.

## How it works...

MySQL needs to store its passwords to be able to verify user credentials. This information is not stored in plain text, but as a so-called hash value. A **hash value** is a value that represents a password, but which is not reversible, so it is basically impossible to find the actual password for a given hash value.

To check whether a given password is correct, MySQL calculates the hash value for the password and compares it with the stored value. If these values are identical, the password is verified.

When creating a user account using the IDENTIFIED BY '<Your plain text password>' syntax, MySQL calculates the hash code and stores the resulting value in the user data. The IDENTIFIED BY PASSWORD '<hash value>' syntax stores the password immediately. In both cases, the effects of the script are identical, but with the second method no plain text password is accidentally exposed.

## Example of creating a user in a script without a plain text password

Let us assume the following script:

```
GRANT ALL PRIVILEGES ON *.* TO 'admin4mysql'@'localhost' IDENTIFIED BY 'As,ysp4M' WITH GRANT OPTION;
```

## MySQL User Management

Execute the following command on the MySQL command line:

```
mysql> SELECT PASSWORD('As,ysp4M');
```

```
Command Prompt - mysql -u sample_guest -p
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 22
Server version: 5.1.31-community-log MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SELECT PASSWORD('As,ysp4M');
+-----+
| PASSWORD('As,ysp4M')          |
+-----+
| *46FFD1D6944482DFCCD3B31AC500199AFDE515F7 |
+-----+
1 row in set (0.08 sec)

mysql> _
```

Replace the password in the script:

```
GRANT ALL PRIVILEGES ON *.* TO 'admin4mysql'@'localhost' IDENTIFIED BY password '*46FFD1D6944482DFCCD3B31AC500199AFDE515F7' WITH GRANT OPTION;
```

### There's more...

The SET PASSWORD command is another common place for use of plain text passwords. To prevent use of plain text passwords altogether, you should replace all SET PASSWORD [...] = PASSWORD ('<plain text password>') expressions in your scripts by their corresponding variant using hash values directly (without using the PASSWORD () function). For example, instead of

```
SET PASSWORD [...] = 'Encoded password'.
```

# Managing Schemas

In this chapter, we will cover:

- ▶ Adding new columns at specific positions
- ▶ Defining a Primary key for a table containing (non-unique) data
- ▶ Allowing individual `INSERT` statements with "0" values in auto-incrementing columns
- ▶ Globally allowing `INSERT` statements with "0" values in auto-incrementing columns
- ▶ Choosing a suitable storage engine
- ▶ Improving the performance of `ALTER TABLE` for InnoDB
- ▶ Using a stored procedure to conditionally add columns or indexes
- ▶ Improving query performance for InnoDB tables with BLOB columns
- ▶ Identifying differences between two schemas
- ▶ Comparing schema revisions using hash values

## Introduction

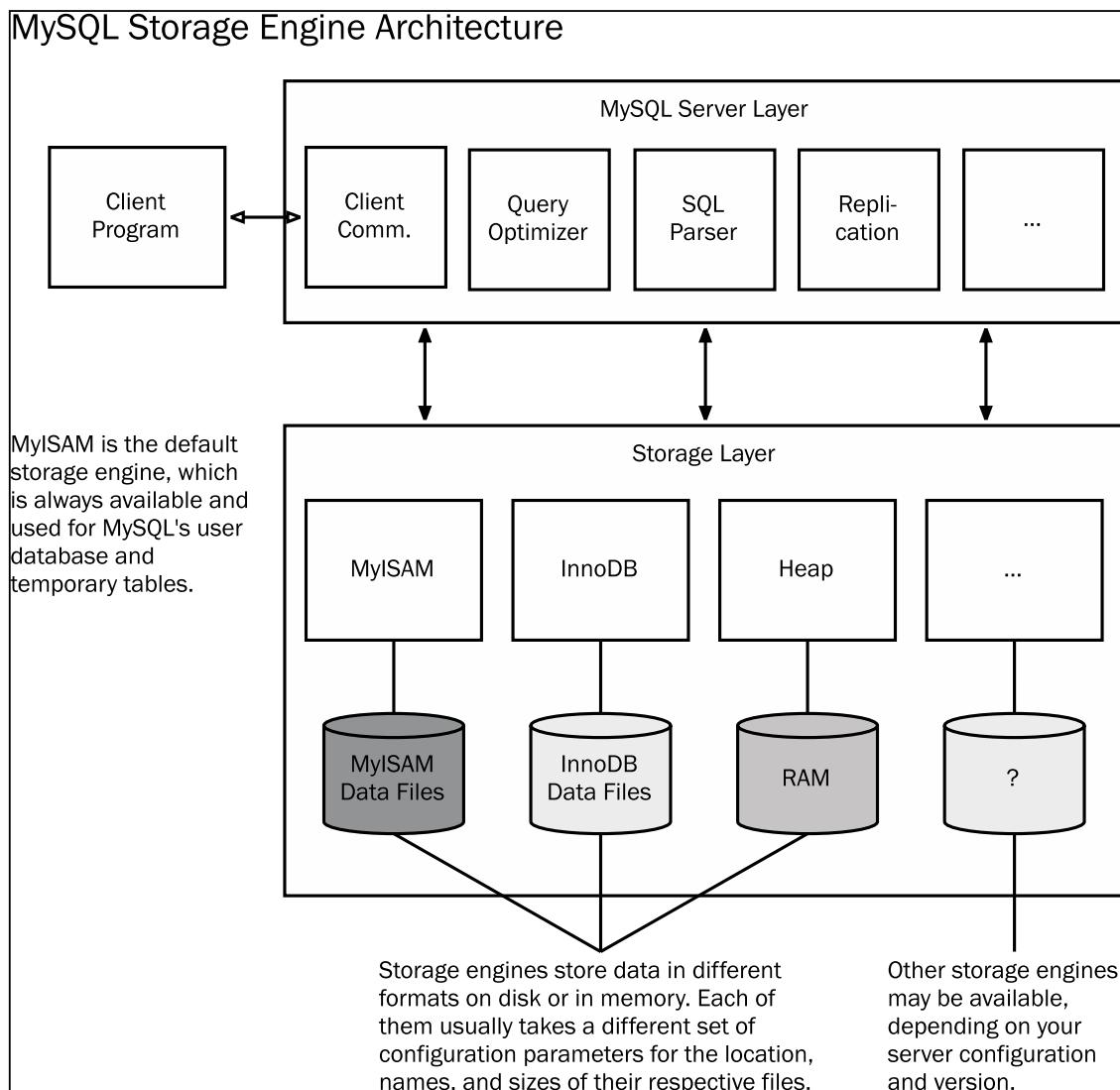
When you first install a database server, you obviously do so because you want to store data and later access information reliably and quickly. A major concern in this regard—apart from the server's hardware and operating system—is the logical layout of the database. This means that you decide on the structure of the tables, which will in the end be what any data access application will use as its primary level of abstraction.

This chapter is not about data types, table naming, or other topics revolving around the creation of tables. It is about decisions that are often best made in cooperation with application developers. Of course, the application developers will have their own ideas about how to set up the database schema that you should take into account as a database administrator (both their and your lives will be easier in the long run).

## Managing Schemas

Instead in this chapter, you will find advice on the specialties and differences that MySQL different from other **Relational Database Management Systems (RDBMS)** Oracle, Microsoft SQL Server, or IBM's DB/2. Moreover, we will have a look at some tasks that will come up time and again, either before you set up the tables or after when your database is already up and serving requests from applications.

MySQL is different from most other RDBMS in that it does allow you to choose from a variety of storage engines. A storage engine is a part of the database system that physically stores your data on a disk, in contrast to, for example, checking the syntax of a statement or executing a function like `DATE()`. With MySQL you get a choice as to which of those implementations (each with its individual strengths and weaknesses) is to be used on a per-table basis. The following picture shows the general architecture of the MySQL server above the individual storage engine implementations:



---

Using this approach, one can choose the optimum storage strategy for each table. There are third parties too that produce (open and closed source) storage engines that specialize in niches like PBXT for media storage. Other databases offer all the features of MySQL's default (and usually only) storage format: from transaction support to high performance, full-text indexing and fine-grained locking to special data types.

MySQL makes the administrator choose from a range of options, none of which supports all the features just named together. So in the end, having the option to pick from the available storage engines is sometimes more of a burden than an opportunity.

The MySQL online manual contains a table contrasting the available storage engines and their respective feature sets in its Chapter 13 at <http://dev.mysql.com/doc/refman/5.7/en/storage-engines.html>.

We will not go into the details of each and every engine available because from a learning perspective—and this is what counts in our opinion—realistically only two of them are worth a detailed discussion: MyISAM and InnoDB.

Apart from that, you might find the Blackhole engine useful for special replication tasks, and you should also take a look at the *Federated* and *Merge* engines, both of which, though with limits, allow multiple tables to be treated as one, locally and over the network.

## Adding new columns at specific positions

One of the regular tasks a database administrator has to perform is to modify the structure of existing tables, especially adding new columns to accommodate the need to store new attributes for the records stored in a table.

While in general the order of columns is not relevant for MySQL itself—or any well-known application accessing columns by name rather than their position in the table—it is often desirable to have control over the order the columns appear in a table.

There are several reasons to precisely control the column order: from a general desire to keep your schema tidy, over the general benefit of a table displaying its columns in a sensible order when doing a `SELECT *`, to other external constraints that you cannot influence.

In this recipe, we will show you how to modify an existing table and add one or more columns at specific positions.

*Managing Schemas***Getting ready**

To follow along, you will need access to MySQL server with a user account that has sufficient privileges to alter table definitions. For example, we will first create a table with a few columns and then add a few more columns at predefined positions. We assume the test database is available and your user account to have the right to create and modify tables in it.

**How to do it...**

1. Connect to the database using the command-line mysql client
2. Make the test database active

```
mysql> USE test;
```

3. Create the initial schema

```
mysql> CREATE TABLE person (
    firstname VARCHAR(30),
    lastname VARCHAR(30),
    birthday DATE,
    person_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (person_id)
);
```

4. Insert some data

```
mysql> INSERT INTO person (firstname, lastname, birthday)
VALUES ('Martin','van Buren','1782-12-05'),
       ('Thomas','Wilson','1856-12-28'),
       ('William','Clinton','1946-08-19');
```

5. Select the full contents

```
mysql> SELECT * FROM person;
```

| firstname | lastname  | birthday   | person_id |
|-----------|-----------|------------|-----------|
| Martin    | van Buren | 1782-12-05 | 1         |
| Thomas    | Wilson    | 1856-12-28 | 2         |
| William   | Clinton   | 1946-08-19 | 3         |

6. Add a salutation column and a middle\_initial column. The former is created before firstname, the latter between firstname and lastname.

```
mysql> ALTER TABLE person
          ADD COLUMN salutation VARCHAR(10) FIRST,
          ADD COLUMN middle_initial CHAR(1) AFTER firstname
Update some data:
mysql> UPDATE person SET salutation='Mr.';
mysql> UPDATE middle_initial='W' WHERE person_id=2;
mysql> UPDATE middle_initial='J' WHERE person_id=3;
```

7. Select the full table contents:

```
mysql> SELECT * FROM person;
```

| salutation | firstname | middle_initial | lastname  | birthday   | person_id |
|------------|-----------|----------------|-----------|------------|-----------|
| Mr.        | Martin    | NULL           | van Buren | 1782-12-05 |           |
| Mr.        | Thomas    | W              | Wilson    | 1856-12-28 |           |
| Mr.        | William   | J              | Clinton   | 1946-08-19 |           |

## How it works...

The `ALTER TABLE ADD COLUMN` command allows for parameters specifying the position of a newly added column. To make a new column the first after the modification is made, use the `FIRST` option. For all other positions, you specify `AFTER` which existing column is to be placed. In our example, we added the `salutation` column at the beginning of the table, while the `middle_initial` column was put right after `firstname`.

Doing a simple `SELECT *` displays the presidents' names and salutations nicely now because of the column order. Had we just added the columns without specifying `AFTER FIRST`, they would have been appended to the end, behind the `person_id` column. As they are semantically identical, the output would not have been as readable.

## There's more...

For a full syntax description of the `ALTER TABLE ADD COLUMN` command, please refer to section 12.7.1 of the online MySQL manual at <http://dev.mysql.com/doc/refman/5.1/en/alter-table.html>.

## Managing Schemas

### See also

- ▶ Comparing schema revisions using hash values

## Defining a primary key for a table containing (non-unique) data

When you need to add a Primary key to a table that did not have one previously, you may find yourself confronted with a problem that is not immediately obvious, but can be rather tricky.

A typical example for this problem occurring is with persistence frameworks that require you to have a Primary key on all tables you want it to manage. If the tables were defined independently and before you knew of this requirement, you might not have defined an explicit Primary key, for example, on the dependent one in a relationship between them.

In this recipe, we will show you a way to simply add a Primary key to a table that already contains data and is even taking part in a Foreign key relationship.

### Getting ready

To follow this example, you will need MySQL server and an account with privileges to create and modify tables and their contents. For the next section, we assume that you have these rights in the `test` schema.

### How to do it...

1. Connect to the MySQL server and make `test` the default database.
2. Create the following tables:

```
CREATE TABLE parent (
    parent_id bigint(20) NOT NULL,
    somevalue varchar(20) default NULL,
    PRIMARY KEY (parent_id)
) ENGINE=InnoDB;
```

```
CREATE TABLE child (
    x_parent_id bigint(20) default NULL,
    value bigint(10) default NULL,
    KEY fk_parent_id (x_parent_id),
    CONSTRAINT child_fk_1 FOREIGN KEY (x_parent_id) REFERENCES
    parent (parent_id)
```

---

There is a 1:0..\* relationship between parent and child. Clearly, it is possible (and intended) that there can be several children with references to the same parent, even if they have equal values.

3. Insert some sample data to demonstrate.

```
mysql> INSERT INTO parent
      VALUES (1,'Parent No. 1'),
              (2,'Parent No. 2'),
              (3,'Parent No. 3');

mysql> INSERT INTO child
      VALUES (1,10),(1,12),(1,15),
              (2,25),(2,26),(2,26),
              (3,31),(3,31),(3,31);
```

4. Trying to add a Primary key to the child table like this will fail. This step simply provides a demonstration. You can try it out, if you want, but you can skip it just as easily.

```
mysql> ALTER TABLE child
      ADD COLUMN child_id BIGINT(20) NOT NULL FIRST,
      ADD PRIMARY KEY(child_id);
ERROR 1062 (23000): Duplicate entry '0' for key 1
```

The reason for this failure is described in the following *How it works...* section.

5. Instead, use these statements to add the primary key.

```
mysql> ALTER TABLE child
      ADD COLUMN child_id BIGINT(20)
      AUTO_INCREMENT NOT NULL FIRST,
      ADD PRIMARY KEY(child_id);
Query OK, 9 rows affected (0.06 sec)
Records: 9  Duplicates: 0  Warnings: 0
```

```
mysql> ALTER TABLE child
      MODIFY COLUMN child_id BIGINT(20) NOT NULL;
Query OK, 9 rows affected (0.08 sec)
Records: 9  Duplicates: 0  Warnings: 0
```



Notice that you cannot combine these two statements into a single `ALTERTABLE` command.

6. When inserting new data, make sure you provide Primary

## Managing Schemas

### How it works...

At first the statement presented first in the steps above would seem sensible to add a key column to the table:

```
mysql> ALTER TABLE child
      -> ADD COLUMN child_id BIGINT(20) NOT NULL FIRST,
      -> ADD PRIMARY KEY(child_id);
ERROR 1062 (23000): Duplicate entry '0' for key 1
```

The error message is a little hard to grasp at first, but becomes perfectly clear when we look more thoroughly about what MySQL tries to do here. First, let's have a look at the same table:

```
mysql> select * from child;
```

| x_parent_id | value |
|-------------|-------|
| 1           | 0     |
| 1           | 0     |
| 1           | 0     |
| 2           | 0     |
| 2           | 0     |
| 2           | 0     |
| 3           | 0     |
| 3           | 0     |
| 3           | 0     |

9 rows in set (0.00 sec)

Trying to add a column in MySQL will fill it with either NULL or with the data type's default value—in this case 0 for a BIGINT column. Declaring it as a Primary key at the same time then ought to fail because all 9 rows would get the same default value, violating the requirement of uniqueness for a key column.

So to get around this, we need some initial distinct values for the column. Later on, in your application, you will have to provide unique key values for new records. Often the persistence layer can provide generated keys once it has been told about the Primary key column. In the initial round, the auto-increment feature comes in handy:

```
mysql> ALTER TABLE child
      -> ADD COLUMN child_id BIGINT(20) AUTO_INCREMENT NOT NULL
      -> ADD PRIMARY KEY(child_id);
Query OK, 9 rows affected (0.06 sec)
Records: 9  Duplicates: 0  Warnings: 0
```

```
mysql> select * from child;
+-----+-----+-----+
| child_id | x_parent_id | value |
+-----+-----+-----+
1	1	10
2	1	12
3	1	15
4	2	25
5	2	26
6	2	26
7	3	31
8	3	31
9	3	31
+-----+-----+-----+
9 rows in set (0.00 sec)
```

This takes care of providing a unique value for each record's newly created Primary column. Because we do not need the auto-incrementing anymore, we can remove

```
mysql> ALTER TABLE child MODIFY COLUMN child_id BIGINT(20)
Query OK, 9 rows affected (0.08 sec)
Records: 9  Duplicates: 0  Warnings: 0
```

The values of the `child_id` column are retained; the final column definition just ignores the temporary `AUTO_INCREMENT` option.

Unfortunately, this cannot be combined into a single `ALTER TABLE` statement because the parser first checks if the statement's parts are all fine before beginning execution. At the time the statement is checked, the latter `MODIFY COLUMN` segment is not valid because at that time the `child_id` column does not exist yet.

### There's more...

The above example only works for numeric key columns because the `AUTO_INCREMENT` feature can only be used for those. If you need a different data type for key, you can of course modify the second `ALTER TABLE` statement to change the column's type instead of just dropping the `AUTO_INCREMENT` option. MySQL will then try to convert the value that it inserted automatically to the newly defined data type—for example, `VARCHAR`:

```
mysql> ALTER TABLE child
        MODIFY COLUMN child_id VARCHAR(20) NOT NULL;
mysql> INSERT INTO child VALUES ('foo', 3, 31);
```

After that you are free to use `UPDATE` statements to modify the key values to your

*Managing Schemas*

## Allowing individual INSERT statements "0" values in auto-incrementing columns

Auto-incrementing columns have many uses; primarily, they are used to automatically generate Primary key values for new records inserted into tables.

The usual behavior for MySQL is to assign the next free number from the auto-increment sequence to a record you insert that has either NULL or 0 as the value for any such column.

However, sometimes it may be necessary to insert an actual 0 value without assigning it via automatic replacement.

In this recipe, we will show you how to do so for individual INSERT statements.

### Getting ready

You will need a MySQL user account that can insert data.

In the example that will follow, we will demonstrate how to execute a single INSERT statement with a 0 (zero) column value, even though the table definition calls for the column to be auto-incremented. We will assume a table called `enumerator` to be present in the test database with the following structure:

```
CREATE TABLE enumerator (
    id INT NOT NULL AUTO_INCREMENT,
    textvalue VARCHAR(30),
    PRIMARY KEY (id)
) ENGINE=InnoDB;
```

### How to do it...

1. Connect to the MySQL server and make test the default schema:
2. Save the current value of the `SQL_MODE` variable for later and set the `NO_AUTO_VALUE_ON_ZERO` option:
 

```
mysql> SET SESSION @OLDMODE=@@SQL_MODE;
mysql> SET SESSION SQL_MODE=CONCAT(@OLDMODE,
   , 'NO_AUTO_VALUE_ON_ZERO');
```

 Make sure you do not miss the comma before `NO_AUTO_VALUE_ON_ZERO`.
3. Insert a record with a 0 value for the `id` column:
 

```
mysql> INSERT INTO enumerator VALUES (0, 'Zero');
```

4.

Read back the data:

```
mysql> SELECT * FROM enumerator;
```

```
File Edit View Terminal Tabs Help
mysql> SELECT * FROM enumerator;
+----+-----+
| id | textvalue |
+----+-----+
| 0  | Zero      |
+----+
1 row in set (0.03 sec)
```

5.

Reset the SQL\_MODE variable to its previous value:

```
mysql> SET SESSION SQL_MODE=@OLDMODE;
```

## How it works...

By default, MySQL will interpret 0 or NULL values for columns marked as AUTO\_INCREMENT as a sign to issue the next free number from its internal auto-increment counter and substitute it for the 0 or NULL value in the INSERT received. This makes it generally safe for applications to ensure conflict-free key values. However, for special requirements, this behavior can be modified to allow 0 as a regular value for individual sessions. This is what we did by adding the NO\_AUTO\_VALUE\_ON\_ZERO option to the SQL\_MODE system variable for the current session. Once the record has been inserted, we restore the variable to its old value.

## Globally allowing INSERT statements with "0" values in auto-incrementing columns

In this recipe, we will show you how to set MySQL's default behavior to globally allow 0 values in columns defined as auto-incrementing for a whole server and all connected clients.

Historically, MySQL created a new automatic value for any insertion with a 0 or NULL value in a column set up as auto-incrementing. This can lead to unexpected behavior if the table you are going to store contains actual zeroes—these will silently be converted to non-zero values from the auto-increment sequence. See the *Appendix* for a more thorough description and demonstration of this effect.

## Managing Schemas

The MySQL online manual's description of the `NO_AUTO_VALUE_ON_ZERO` option includes a paragraph on how this behavior could even lead to changed data when restoring from backup dumps (just the opposite of what a backup is supposed to accomplish). For this reason, current `mysqldump` versions (starting from MySQL 4.1.1) make sure that it is turned on automatically when restoring by including it in the dump file.

To avoid accidentally falling into the same trap, in this recipe we will globally enable the `NO_AUTO_VALUE_ON_ZERO` for all databases of the MySQL server instance.

### Getting ready

You will need an operating system account with privileges to change the MySQL server configuration file. To activate the new setting, you will have to restart the server after the configuration file has been edited.

### How to do it...

1. Open the MySQL configuration file with a text editor.
2. In the `[mysqld]` section find any existing `SQL_MODE` setting. If it's not present, add it, otherwise append to any existing settings, separated by a comma like here:

```
[mysqld]
...
sql-mode=STRICT_ALL_TABLES,NO_AUTO_VALUE_ON_ZERO
```
3. Save the file.
4. Restart the MySQL server.

### How it works...

The per-session variable you set in the recipe mentioned earlier can be configured as the default by setting the value in the config file. This is what this recipe has shown you. As an example of how the setting works, see the recipes listed in the See also section. Of course, you will not need the `SET SQL_MODE` statements contained therein, as you can set the same behavior up as the default.

### See also

- ▶ *Allowing individual INSERT statements with "0" values in auto-incrementing columns*
- ▶ *Understanding auto-increment values (Appendix)*

# Choosing a suitable storage engine

When creating tables, either by mistake or on purpose, you may use a storage engine that does not prove to be the right choice later. Fortunately, MySQL allows you to change the storage engine type of existing tables with a single statement. Fittingly, MySQL extended the syntax of the `ALTER TABLE` command to allow you to specify the storage engine as well as another property. The operation is non-destructive, which means existing data is preserved in the table.

In this recipe, we will show you how to move a table from MyISAM to InnoDB. Of course, there are many more storage engine types available, but these two are the most widely used and therefore most relevant.

## Getting ready

To follow along, you will need a MySQL user account with access to the test database and the rights to modify table structures (the `ALTER` privilege). We will be using a table definition like this:

```
CREATE TABLE person (
    salutation char(10) DEFAULT NULL,
    firstname varchar(30) DEFAULT NULL,
    middle_initial char(1) DEFAULT NULL,
    lastname varchar(30) DEFAULT NULL,
    birthday date DEFAULT NULL,
    person_id int(10) unsigned NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (person_id)
) ENGINE=MyISAM;
```

If it does not exist yet in your test database, execute the above `CREATE TABLE` statement before moving on.

## How to do it...

1. Connect to the database and make test the default database.
2. Change the table storage engine.

```
mysql> ALTER TABLE person ENGINE=InnoDB;
Query OK, 3 rows affected (0.24 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

## Managing Schemas

### How it works...

The `ALTER TABLE` statement tells MySQL to create a new InnoDB table with the same definition as the source table and copy all the contents over there. Once done, the old table will be removed and the new one put in its place. This operation does preserve the data in the original table.

However, as changing the storage engine effectively means storing the data in a different format on disk, your server will have to deal with potentially lots of I/O, directly proportional to the current size of the table. In this example, the table was very small (only three rows), so everything went quickly.



Large tables, however, will take much longer because not only will all data have to be read, but also written simultaneously.

During that operation, MySQL will allow read operations, but any writes (`UPDATE`, `DELETE`, and `INSERT`) will be delayed until the conversion is complete. Only then will they execute against the new table. This will cause trouble for most applications not expecting their operations to take that long.

This is why you will want to delay changing the storage engine to a low-traffic time period or better yet, an offline maintenance window!

Moreover, your disks must provide enough space for the tables in both the old and new formats at least temporarily because MySQL will only remove the old table once the new one has been successfully created and all the data has been transferred.

Be careful here that different storage engines have different disk space requirements for the same data. InnoDB tables tend to be larger than their MyISAM counterparts.

### There's more...

The example outlined in this recipe was a very simple one—converting from MyISAM to InnoDB. There is usually no problem because InnoDB basically offers the same features as MyISAM and more. However, you will not be able to convert MyISAM tables that have full-text indexes because those are not supported by InnoDB. If you try to alter the engine type on such a table, you will get an error message:

```
mysql> ALTER TABLE forum_posts ENGINE=InnoDB;
ERROR 1214 (HY000) : The used table type doesn't support FULLTEXT
indexes
```

Converting tables from InnoDB to MyISAM is usually less common, but still possible as there are no Foreign key relationships in place. This applies to both sides of any relationship. If, for example, you had two InnoDB tables called `parent` and `child`, child records were set up to refer to their parent row in that table, trying to change engine on any of these would fail:

```
mysql> ALTER TABLE parent ENGINE=MyISAM;
ERROR 1217 (23000): Cannot delete or update a parent row: a foreign key constraint fails

mysql> ALTER TABLE child ENGINE=MyISAM;
ERROR 1217 (23000): Cannot delete or update a parent row: a foreign key constraint fails
```

Before you could change the storage engine for those, you would have to first drop the key constraints on them.

Apart from MyISAM and InnoDB, there are many more storage engines available. For information on their respective capabilities and other properties please refer to Chapter 10, “Storage Engines,” and the MySQL online manual at <http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html>, and to the `ALTER TABLE` documentation in section 12.1.7 at <http://dev.mysql.com/doc/refman/5.1/en/alter-table.html>.

## Keeping a watch on silent engine substitution

Special attention should be paid to the fact that MySQL might silently ignore your request and even modify the storage engine type to a default one, if the one you specified is not available on your server. The exact behavior depends on your MySQL server version and the value of the `SQL_MODE` configuration variable:

|                                                        | <b>MySQL 5.1.11 and older</b>                             | <b>MySQL 5.1.12</b>                            |
|--------------------------------------------------------|-----------------------------------------------------------|------------------------------------------------|
| <code>NO_ENGINE_SUBSTITUTION</code> specified          | Gives error                                               | Gives error                                    |
| <code>NO_ENGINE_SUBSTITUTION</code> disabled (missing) | Uses default storage engine and issues warning, not error | Ignores requests and issues warning, not error |

Please note that this table does not apply to `CREATE TABLE`, but only to `ALTER TABLE`. For more information on this topic, see the MySQL online manual, section 5.1.8 on SQL Modes at [http://dev.mysql.com/doc/refman/5.1/en/server-sql-mode.html#sqlmode\\_no\\_engine\\_substitution](http://dev.mysql.com/doc/refman/5.1/en/server-sql-mode.html#sqlmode_no_engine_substitution).



We recommend always enabling the `NO_ENGINE_SUBSTITUTION` option in your MySQL configuration file (`my.cnf`) to prevent silent engine substitution.

*Managing Schemas*

---

## Improving the performance of ALTER TABLE for InnoDB

This recipe will show you a way of limiting the unavoidable impact of InnoDB table changes to the necessary minimum.

Unfortunately, there is no way of circumventing that, as `ALTER TABLE` on InnoDB is a time-consuming and I/O-intensive operation, but planning ahead can make a huge difference as to how severe the impact on your systems will be.

### Getting ready

While this recipe applies to any InnoDB table, the actual benefits of the following recommendations are best seen with a large table. For the example below, we will use the `employees` sample database's `salaries` table, which contains slightly less than one million records and is about 100 MB in size. This sample database can be downloaded from the MySQL website at <http://dev.mysql.com/doc/employee/en/employee.html>.

The table is defined like this:

```
CREATE TABLE salaries (
    emp_no int(11) NOT NULL,
    salary int(11) NOT NULL,
    from_date date NOT NULL,
    to_date date NOT NULL,
    PRIMARY KEY (emp_no,from_date),
    CONSTRAINT salaries_ibfk_1 FOREIGN KEY (emp_no) REFERENCES
    employees (emp_no) ON DELETE CASCADE
) ENGINE=InnoDB
```

We will add an index to the `from_date` column and also add a column called `remainin`

You will need a user account with sufficient rights to change the table definition.

## How to do it...

1. Connect to the database server using the administrative user and make the default database.

2. Check on the current size of the table using the following statement (output)

```
mysql> SHOW TABLE STATUS LIKE 'salaries' \G
***** 1. row ****
      Name: salaries
      Engine: InnoDB
      Rows: 2844513
      Data_length: 100270080
      Data_free: 1352663040
1 row in set (0.07 sec)
```

As you can see, the table is approximately 95 MB big, with about 1,290 MB of table space. Assuming that no other operations will fill up this space during TABLE, we still need to consider the larger table size caused by the addition. In this example, we should be fine. If the table space is auto-extending and there is enough free space on the volume it is stored on, you are good to go as well; if not, you should go and make sure the table space gets extended before you proceed.

3. Issue the following ALTER TABLE statement that adds the new column **in one step**:

```
mysql> ALTER TABLE salaries
      ADD COLUMN remark VARCHAR(32) DEFAULT NULL
      AFTER to_date,
      ADD INDEX IDX_FROMDATE(from_date);
```

- 4.

This will take

## How it works...

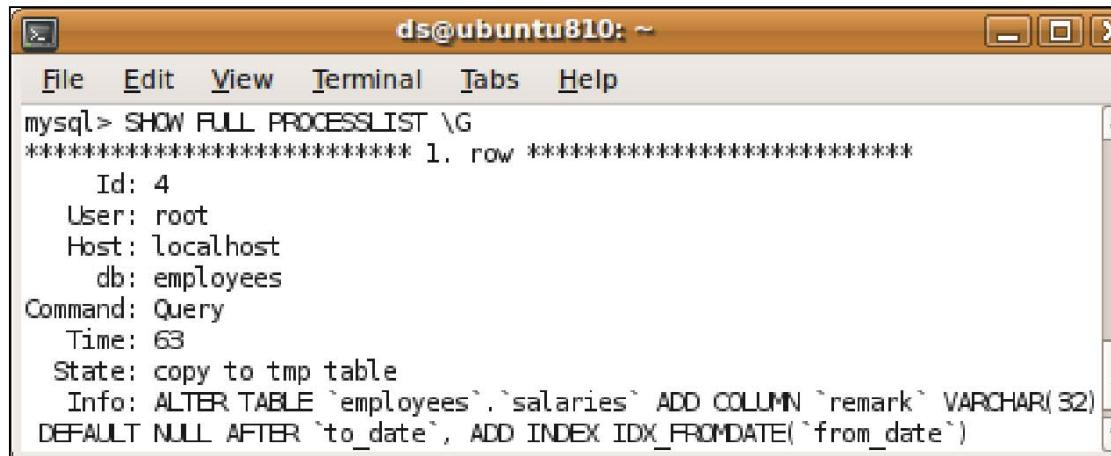
When modifying an InnoDB table using the ALTER TABLE command, in most MySQL installations (see the *There's more...* section for information about exceptions) it will perform the modifications by first creating a new table with the modified definition, then copying data from the original table, and finally swapping the newly-created, modified one for the old one.

As tables grow, this can take a long time, creating lots of I/O activity for the copy process. Making matters worse, this process requires much disk space because temporarily you have both the original table and the copy present on disk, filling up the remaining space thereby potentially influencing other applications. But even just inside MySQL itself, you can run into problems because your tablespace might not have enough free space left.

## Managing Schemas

Using a second connection, you can actually see the temporary table being used to run the following command while the `ALTERTABLE` is still running:

```
mysql> SHOW FULL PROCESSLIST \G
```



The screenshot shows a terminal window titled "ds@ubuntu810: ~". The window contains the following text:

```
File Edit View Terminal Tabs Help
mysql> SHOW FULL PROCESSLIST \G
***** 1. row *****
Id: 4
User: root
Host: localhost
db: employees
Command: Query
Time: 63
State: copy to tmp table
Info: ALTER TABLE `employees`.`salaries` ADD COLUMN `remark` VARCHAR(32)
      DEFAULT NULL AFTER `to_date`, ADD INDEX IDX_FFROMDATE(`from_date`)
```

Though you cannot prevent InnoDB from copying the tables for modification, you can minimize the influence of this procedure by trying to make all necessary modifications at once. That means instead of, say, first adding a new column with one `ALTER TABLE` command and then adding the new index with a second one, you combine both in a single statement.

The longer composite statement itself may not be as readable as individual ones (though you can work around that with proper indentation and formatting), but it will only require a single copy pass instead of one for each modification.

There are limits to this (for example, you cannot drop a constraint at the beginning of a statement and then add another one with the same name later), but in general, this is the way to go as soon as the table in question contains more than just a couple of rows.

The example statement above took 1 minute and 34 seconds on my iMac. The same operation with an unmodified MySQL server configuration took 3 minutes and 2 seconds when using separate `ALTER TABLE` statements for the index and the new column individually.

### There's more...

Starting with MySQL 5.1, storage engines can be loaded into the MySQL server as plugins. Although InnoDB comes bundled with MySQL, there is a separate plugin version that was developed independently of the main server. At the time of writing this book, MySQL 5.4 was in beta and contained some of the newer features that were only available in the plugin release, available from <http://www.innodb.com>. As the MySQL server plugin model is currently in a state of change, it is hard to predict what the most recent version will be at the time you are reading this and what it will be called.

However, apart from other improvements, the main feature worth mentioning is called **index creation**. It allows creating and dropping indexes without needing to copy all its data. This can alleviate the pain of schema modifications significantly. However, MySQL bug #33650 (<http://bugs.mysql.com/bug.php?id=33650>), this feature cannot be used if any of the indexed columns are configured to use utf8 as their text encoding. We suggest you monitor this bug if you want to try out the plugin.

## Using a stored procedure to conditionally add columns or indexes

Several of MySQL's schema-related commands allow for an **IF EXISTS** clause, which is a very useful addition to standard SQL syntax because it allows for more robust automated schema handling. When you need to do unattended schema manipulations, for example, to re-create a table when you do not know for sure whether it exists on the target system, simply do a

```
DROP TABLE IF EXISTS tablename;
CREATE TABLE tablename (...)
```

Without the **IF EXISTS** clause, the **DROP TABLE** statement would fail if the table was present when executing the script and abort the execution immediately, in effect erasing the table without the new table.

Unfortunately, there are some cases where the **IF EXISTS** clause would come in handy. For example, the **ALTER TABLE** command is not supported by MySQL. One such case is the addition of new indexes to an existing table. In this recipe, we will present a way to work around this limitation and write portable scripts to modify a table unattended.

Note that this is primarily useful for the automatic execution of updates to databases that are not under your immediate control, for example, as part of a software update installation. For manual modifications of a schema, it is usually way less work to have a look at the current table structure first to determine if there is anything to do in the first place.

### Getting ready...

In the following example, we will be modifying two tables in the `test` database schema. The two tables will be identical, except for one having an index `IDX_B` already, while the other does not. To follow along, you will need a user account with sufficient privileges to first create the tables and then modify them. This is done via a stored procedure that you must have the privilege to create as well.

## Managing Schemas

---

These are the table definitions:

```
CREATE TABLE TableA (
    col_A int(11) DEFAULT 1,
    col_B varchar(40) DEFAULT NULL
) ENGINE=InnoDB;

CREATE TABLE TableB (
    col_A int(11) DEFAULT 1,
    col_B varchar(40) DEFAULT NULL,
    KEY IDX_B (col_B)
) ENGINE=InnoDB;
```

### How to do it...

1. Connect to the MySQL server using an administrative user.
2. Create a stored procedure with the following sequence of instructions. The steps will be explained in the *How it works...* section:

```
mysql> DELIMITER $$

mysql> DROP PROCEDURE IF EXISTS sp_AddIndex $$

mysql> CREATE PROCEDURE sp_AddIndex
        (tblName VARCHAR(64), ndxName VARCHAR(64),
         colName VARCHAR(64))

BEGIN
    DECLARE IndexColumnCount INT;
    DECLARE SQLStatement VARCHAR(256);
    SELECT COUNT(index_name) INTO IndexColumnCount
    FROM information_schema.statistics
    WHERE table_schema = database()
    AND table_name = tblName
    AND index_name = ndxName;
    IF IndexColumnCount = 0 THEN
        SET SQLStatement = CONCAT('ALTER TABLE ',tblName,
        ' INDEX ',ndxName,' (',colName,')');
        SET @SQLStmt = SQLStatement;
        PREPARE s FROM @SQLStmt;
        EXECUTE s;
        DEALLOCATE PREPARE s;
    END IF;
END $$
```

- 
3. Call the procedure, providing the table name, index name, and the column(s) to be indexed:

```
mysql> CALL sp_AddIndex('TableA','IDX_B','col_B');
```

4. Clean up, removing the procedure and resetting the statement delimiter:

```
mysql> DROP PROCEDURE IF EXISTS sp_AddIndex;
```

```
mysql> DELIMITER ;
```

## How it works...

While MySQL does not offer a direct way of executing the statement only if the target table already exists, a stored procedure can be used to perform this check. The first statement used to change the default statement delimiter from ; to something different—two dollar signs \$\$ in this case. This allows the definition of the actual procedure contents in a more modular form, using the default semicolons to separate the routine's commands.

The first application of this new delimiter is a statement to drop any procedure with the same name as that we are about to use. It ends with \$\$, the temporary replacement for the usual semicolon.

After that, the actual procedure is declared. It is called `sp_AddIndex` (`sp_` being a common prefix for stored procedures, but this is not strictly necessary) and takes three parameters to work with. The first one is the table name to operate on, the second one is used to pass the name of the target index, and the last one the name of the column(s) that are supposed to span.

It then queries the `INFORMATION_SCHEMA` to find out if there already is an index with the given name on the given table in the current schema. The result of the `COUNT()` function is zero if there is no index matching the given criteria. This is then checked in the `IF` condition guarding the building of a suitable `ALTER TABLE ... ADD INDEX` statement, which is then executed. If an index already existed, the `IndexColumnCount` result will be greater than zero, hence skipping the `ALTER TABLE` statement.

Finally, the procedure is called for `TableA` and `col_B` with an index name of `IDX_B`. Once the procedure is completed, it is dropped again and the default delimiter is restored to the semicolon.

## Managing Schemas

### There's more...

Of course, you need not drop the procedure if you plan to use it more regularly. However, this example was taken from an automated script we use to update large numbers of schemas at a time to time. Therefore, we usually create and drop such maintenance procedures.

At the time of writing this book, there was a bug in MySQL version 5.1.30 that lead to a wrong result of the COUNT() query inside the procedure on Mac OS X and Windows. After reporting the bug to MySQL support and some e-mailing back and forth, MySQL Bug #46771 (<http://bugs.mysql.com/bug.php?id=46771>) was put into their bug database.

While I recommend you to go and check on the state of things when you read this, the main point here is that the INFORMATION\_SCHEMA pseudo-database does not answer queries correctly unless you enter table names all in lower case! Usually, you would not notice on Windows or Linux because they have case-insensitive file systems, while most Unix file systems do.

To work around this bug, make sure you only use lower case table names if you manage your MySQL server on one of those operating systems, unless you find that your server version has been fixed.

## Improving query performance for InnoDB tables with BLOB columns

MySQL and InnoDB behave in a hardly predictable way when querying tables that contain BLOB columns. This might catch you off guard and cause slow performance where it does not have to be. This is caused by a bug (or a missing feature, depending on your perspective) in MySQL.

In this recipe, we will demonstrate how to reproduce this bug, and in the *How it works...* section explain how to work around it. Of course, you should first check the above MySQL bug and see whether it has been fixed at the time you are reading this in the MySQL server version you are using. However, as this bug has been open since 2004 and has been prioritized as a feature request, I do not think chances are especially good that this has happened.

### Getting ready

To follow along, you will need a MySQL user account with sufficient privileges to create tables in the test schema and insert data into them. For the BLOB contents, a file will be needed to read from. In the example, we will use /tmp/blobdemo, but the content is not important. You could also use any other file at your disposal.

## How to do it...

1. Connect to the database and make the test database the default.
2. Create a table with the following command:

```
mysql> CREATE TABLE blobtest (
    intA INT(10) NOT NULL AUTO_INCREMENT,
    intB INT(10) DEFAULT NULL,
    contents BLOB,
    PRIMARY KEY (intA)
) ENGINE=InnoDB;
```

3. Load the file contents into it. Make sure to adapt the path name if you want to use a different file:

```
mysql> INSERT INTO blobtest (intB, contents)
VALUES(100, LOAD_FILE("/tmp/blobfile"));
```

4. Execute the following command 10 times to produce some data:

```
mysql> INSERT INTO blobtest (intB, contents)
SELECT intB, contents FROM blobtest;
```

Each time you repeat this command, the number of records in the blobtest table will double. To make the effect more obvious, you could do more than 10 repeats, but beware that each one of them will take quite a lot longer to execute.

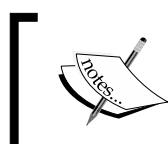
5. Execute a query that asks for all the columns, including the contents column. The result set does not return any rows.

```
mysql> SELECT * FROM blobtest WHERE intB=0;
Empty set (2.64 sec)
```

The WHERE clause does not mention the BLOB type contents column.

6. Execute a query that explicitly excludes the BLOB type column.

```
mysql> SELECT intA, intB FROM blobtest WHERE intB=0;
Empty set (0.01 sec)
```



Notice that this query was completed significantly faster than the one above, even though they both queried the same condition on the same table and did not have to return any rows.

## Managing Schemas

### How it works...

Back in 2004 we filed MySQL Bug #7074 (<http://bugs.mysql.com/bug.php?id=7074>) which reports this unexpected behavior. Heikki Tuuri, the inventor of InnoDB, conjectured that MySQL first reads all the columns you specify in a `SELECT` statement from the data store, before applying the `WHERE` condition to them. This is usually a good idea in case the criteria match, you already have all the data at hand. However, for BLOBs this can quickly become overly expensive in terms of execution time because lots of data has to be read from disk.

In the previous example (which is the same as in the bug report), we intentionally ran queries that do not match any rows—all the values of `intB` have been set to the constant value 100—to show the effect very clearly.

Of course, this is only an issue if InnoDB cannot use an index to check for the `WHERE` conditions. If it finds a suitable one, no BLOBs will be read in the earlier example for all the queries, no matter which columns are requested in the `SELECT`. However, depending on the execution plan the optimizer generates, you might still end up with a situation like this if the circumstances are right. This is what actually happened to us—which is how we found out about the problem in the first place—and was tricky to diagnose.

So in the end there are two strategies to prevent this from hitting you:

- ▶ Make sure all queries against the table in question are covered by an index.
- ▶ Do not use `SELECT*` or a complete list of columns in your queries and instead just pull up the key values of the rows you want to process. Then go and retrieve the other columns with another query, leveraging Primary key lookups.

While having good indexes in place is always a thing to strive for, we also strongly recommend only reading the columns you are actually going to use in any query. Especially, using the `*` wildcard in your programs can make them susceptible to errors when you have to make modifications to the table structure—column ordering being a good candidate—in the future. Apart from making your software more robust, it can also help reduce network load between the client and the server.

### There's more...

If you do not want to use a real file for the BLOB columns contents, you can easily fill them with completely random contents from the Linux or Mac OS X command line:

```
$ cd /tmp
$ dd if=/dev/urandom of=blobfile bs=4096 count=16
16+0 records in
```

---

What this does is read from the pseudo-device called `/dev/urandom` that simply provides a continuous stream of (almost) random bytes (the `if` parameter is short for *input file*) and writes 16 blocks of 4,096 bytes (`bs` means *block size*) to the *output file* (`of` for *output file*) called `blobfile`.

By varying the `bs` and `count` parameters, you can control the size of your output file.

## Identifying differences between two schemas

Especially when preparing software upgrades, it is important to know the differences between a previous release's database schema and the current, probably modified development version. There are several tools available to create a report on the differences between schema definitions, trying to make smart guesses as to what went on. Nevertheless, we have found most of them to be of only limited use. This is mostly due to the fact that when an automated tool compares two schema definitions, it cannot know about many aspects that are obvious to the human eye.

Consider a simple column rename, for example. In a previous schema version, a column might have been simply called `name` while in the current release it is to be called `lastname`. A simple comparison of the individual table definitions will not reveal, however, that the two columns have any relationship. Most automated tools will tell you to `DROP COLUMN name` and `ADD COLUMN lastname`. This is, of course, not what you want to do because doing so would lose all the personal data already stored in the table.

In the end, from our experience it is the most primitive, but also the most efficient way to compare two schemas by simply text-diffing their definitions. This is what we will do in this recipe.

### Getting ready

To follow along, you will need privileges to create a structure dump of both the old and new schema definitions. In the next example, we assume you have already dumped the schema definitions to two files called `old.sql` and `new.sql`. You can find these files on the book's website for download. They contain information about three tables called `TableA`, `TableB`, and `TableC` in the `test` schema.

Table definitions for `TableA` and `TableB` are different in several details, while `TableC` is missing completely in `new.sql`.

## Managing Schemas

On Mac OS X and any Linux distribution, the `diff` command is part of the usual operating system installation. Windows users will have to get hold of a copy of a text comparison tool separately. The `diff` command for Windows is part of the Unix Utils (<http://unxutils.sourceforge.net>), while WinMerge is an open-source graphical utility and can be downloaded from <http://winmerge.org>. Many modern text editors and development environments (like Eclipse) do include a text comparison feature as well, and most of them are available cross-platform.

The steps below show a screenshot using WinMerge. The same information can be obtained using the `diff` program. See the *There's more...* section for details.

### How to do it...

1. Put both `old.sql` and `new.sql` in a common directory.
2. Open them both in WinMerge. You will see a screen like this:

```

WinMerge - [old.sql - new.sql]
File Edit View Merge Tools Plugins Window Help
old.sql - new.sql
C:\Users\Daniel Schneller\Desktop\old.sql
CREATE TABLE `test`.`TableA` (
  `A_ID` int(11) NOT NULL AUTO_INCREMENT,
  `col_A` int(11) NOT NULL DEFAULT '1',
  `col_B` varchar(40) DEFAULT 'b.default',
  `col_C` datetime DEFAULT NULL,
  PRIMARY KEY (`A_ID`),
  KEY `VAR_INDEX` (`col_B`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='First'

CREATE TABLE `test`.`TableB` (
  `ID` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` char(30) CHARACTER SET utf8 COLLATE utf8_b
  `street` varchar(50) NOT NULL DEFAULT '',
  `zipcode` char(5) DEFAULT NULL,
  `city` varchar(30) DEFAULT NULL,
  PRIMARY KEY (`ID`),
  KEY `IDX_street` (`street`),
  KEY `IDX_ZIP` (`zipcode`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='Old B'

CREATE TABLE `test`.`TableC` (
  `ID_C` int(11) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`ID_C`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='To be

C:\Users\Daniel Schneller\Desktop\new.sql
CREATE TABLE `test`.`TableA` (
  `A_ID` int(11) NOT NULL AUTO_INCREMENT,
  `col_A_new` int(11) DEFAULT '1',
  `col_B` varchar(40) DEFAULT 'b.default',
  `col_C` datetime NOT NULL DEFAULT '2009-01-01 00:00:00',
  PRIMARY KEY (`A_ID`),
  KEY `VAR_INDEX` (`col_B`(10))
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='Second

CREATE TABLE `test`.`TableB` (
  `ID` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` char(30) CHARACTER SET utf8 COLLATE utf8_b
  `street` varchar(100) CHARACTER SET utf8 COLLATE utf8_b
  `zipcode` char(5) DEFAULT NULL,
  `city` varchar(40) DEFAULT NULL,
  PRIMARY KEY (`ID`) USING BTREE,
  KEY `IDX_street` (`street`),
  KEY `IDX_ZIP` (`zipcode`,`city`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='New B

Ln: 13 Col: 10/44 Ch: 1 | 1252 | Win | Ln: 1 Col: 1/32 Ch: 1/32 | 1252 |
Ready | 6 Differences Found

```

- 
4. Identify the differences between the old and new schema based on this output and formulate the appropriate ALTER TABLE statements.

Table creations and dropped tables need no changes, of course. For this exercise, the following modifications are required:

```
mysql> ALTER TABLE test.TableA
CHANGE COLUMN col_A col_A_new INTEGER DEFAULT 1,
MODIFY COLUMN col_C DATETIME NOT NULL
        DEFAULT '2009-01-01 00:00:00',
DROP INDEX VAR_INDEX,
ADD INDEX VAR_INDEX
        USING BTREE(col_B(10)),
ENGINE = InnoDB;

mysql> ALTER TABLE test.TableB
MODIFY COLUMN street VARCHAR(100) BINARY
        CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
MODIFY COLUMN city VARCHAR(40)
        CHARACTER SET utf8 COLLATE utf8_general_ci
        DEFAULT NULL,
DROP PRIMARY KEY,
ADD PRIMARY KEY USING BTREE(ID),
DROP INDEX IDX_ZIP,
ADD INDEX IDX_ZIP(zipcode, city),
ENGINE = InnoDB
COMMENT = 'New B';
DROP TABLE test.TableC;
```

5. Repeat the schema dump for the altered schema and compare again against the expected schema. You should not see any more differences, if your modifications were correct.

## How it works...

The principle of this recipe is quite simple and very straightforward: create a textual representation of the current (old) schema and the desired (new) schema definition, and compare them.

Based on the result and your knowledge about the context of the changes, you then formulate the appropriate ALTER TABLE statements to migrate your schema from the old to the new state.

Finally, you rerun the comparison to make sure your modification statements were the right thing.

## Managing Schemas

### There's more...

For automation or pure command-line access to servers, the textual output of the can be more suitable. Most versions of `diff` have a wealth of options, but the most one is `-u`, producing a *unified* `diff` formatted output:

```
$ diff -u old.sql new.sql > difference.sql
```

The newly created `difference.sql` file will contain the same information WinMerge visually, but in a standardized textual format. You can find the `difference.sql` book's website, too.

If you are familiar with the `diff` tool, the output should be pretty obvious to you. If look at the following simple example (taken from the recipe steps earlier):

```
--- old.sql      2009-12-15 00:39:40.000000000 +0100
+++ new.sql      2009-12-15 00:39:35.000000000 +0100
```

These first two lines tell you which files were compared with one another. The first prefixed with three minus signs (---), telling you that whenever one of the following starts with a minus, that line was taken from the `old.sql` file.

Similarly, the second line informs you that any subsequent line starting with a plus taken from the `new.sql` file.

After that, the `diff` tool renders lines with different contents from `old.sql` and like this:

```
CREATE TABLE `test`.`TableA` (
  `A_ID` int(11) NOT NULL AUTO_INCREMENT,
-  `col_A` int(11) NOT NULL DEFAULT '1',
+  `col_A_new` int(11) DEFAULT '1',
```

The first two lines serve to give some context that was identical in both files. This allows (and automatic tools reading the `diff` output) to better understand where the differences were found in the input files.

In this case, we can clearly see that the definition of `TableA` was changed. The line with - shows what it looks like in the `old.sql` file, the line directly below, prefixed with + shows what it looks like in the `new.sql` file. So you see that the column was renamed from `col_A` to `col_A_new` at the same time dropping the `NOT NULL` constraint. This change will need to be taken care of with an `ALTER TABLE` statement to migrate `TableA` from the old to the new schema.

With some practice, reading this output will become second nature to you. However, if you do not want to mentally parse this format, we recommend one of the many graphical tools available. If you cannot use these, the `diff` command has a `-y` option that displays

```

Yavin-Mac:Scripts ds$ diff -y old.sql new.sql
CREATE TABLE `test`.`TableA` (
  `A_ID` int(11) NOT NULL AUTO_INCREMENT,
  `col_A` int(11) NOT NULL DEFAULT '1',
  `col_B` varchar(40) DEFAULT 'b.default',
  `col_C` datetime DEFAULT NULL,
  PRIMARY KEY (`A_ID`),
  KEY `VAR_INDEX` (`col_B`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='First table';

CREATE TABLE `test`.`TableB` (
  `ID` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` char(30) CHARACTER SET utf8 COLLATE utf8_bin DEFAULT '',
  `street` varchar(50) NOT NULL DEFAULT '',
  `zipcode` char(5) DEFAULT NULL,
  `city` varchar(30) DEFAULT NULL,
  PRIMARY KEY (`ID`),
  KEY `IDX_street` (`street`),
  KEY `IDX_ZIP` (`zipcode`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='Old B';

CREATE TABLE `test`.`TableC` (
  `ID_C` int(11) NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (`ID_C`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT='To be dropped';
Yavin-Mac:Scripts ds$ 

```

The differences are marked by the characters in the middle between the file contents. For more information, consult your `diff` command's man page.

## See also...

- ▶ *Comparing schema revisions using hash values*

# Comparing schema revisions using hash values

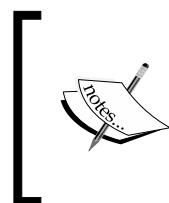
When dealing with software product versioning, you almost invariably will be facing situations where you have to upgrade a database schema from one version to a newer one. The problem now is that even with robust scripts, you cannot be 100% sure that your updates will apply correctly, unless you are perfectly sure they are applicable to a current schema running on a client's computer. This is even more true when the updates are run unattended, which is a scenario we often face when silently upgrading several hundred MySQL instances across several countries.

Applying an unsuitable set of `ALTER TABLE` or even `DROP TABLE` commands to a database might lead to unrecoverable data loss and a broken system. This is something you must avoid at any cost.

A simple way to try to make sure you know what you are dealing with would be to just compare the version of the accompanying application or the contents of some (fictional)

## Managing Schemas

In this recipe, we present a way to calculate checksums for table schemas that enable you to verify beyond doubt if you are dealing with a well-known version that has not been tampered with. This is based on cryptographic checksums that were specifically designed for this purpose. Using an approach like this is superior to, say, a `schema_version` table because it is much harder to easily get out of sync with reality, be it due to someone patching the database or simply forgetting to update the schema modification.



This method will tell you whether a schema conforms to a given well-known schema version you tested your modifications with. It will not tell you what the exact differences are, but rather which of the many possible schema versions the system is currently dealing with.

The major benefit of using this hash-based approach is that you can use it to uniquely identify any given schema version. This is very handy when you are tracking a whole lot of different schema versions because the hash values are very short and relieve you from carrying around many historic versions of schemas you might have. Instead, your upgrade process could automatically pick from a several available update scripts the one that is suitable for upgrading to the latest and recent version the particular schema it identified on the target machine.

### Getting ready

To follow along, you will need operating system and MySQL user accounts with privileges sufficient to run a `mysqldump` command targeted at the schema you would like to calculate a hash value for. In this example, we will be using the `test` database schema that contains two tables `TableA` and `TableB`. The exact definition of those tables is not really important; you can use any other schema as well. However, to try out provoking a hash checksum mismatch, you will have to make at least some minor modification to the schema you pick.

The program used here to calculate the hash values is written in Java. This is not mandatory, of course, the general principle can be implemented in any language you like. We chose Java because it is available for practically any platform and is often installed anyway, especially on Windows, where it is probably more common than Perl, for example.

If you do not have a **Java Runtime Environment (JRE)** installed yet, please download and install it before you proceed. The machine you install it on need not be the MySQL server machine, but can be your workstation instead. However, you will have to be able to connect to the MySQL server via the network in that case, which might be restricted by the MySQL account setup or your company's network firewalls.

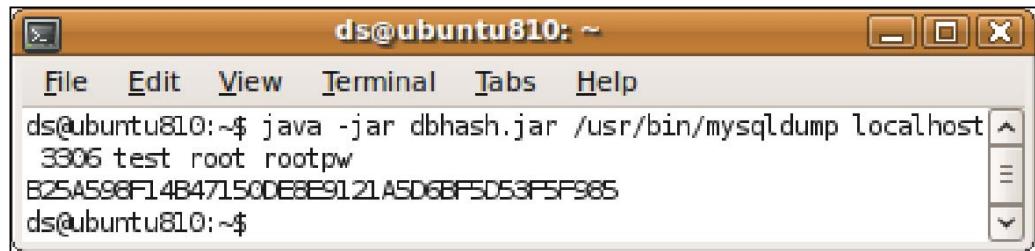
A description of the algorithm used by the program can be found in the *How it works...* section.

## How to do it...

1. Download the dbhash.jar file from the book's website. It contains a pre-compiled, runnable version of the dbhash tool.

2. On a command line, enter the following command, substituting your database, and user names appropriately:

```
$ java -jar dbhash.jar /usr/bin/mysqldump localhost 3306
rootpw
```



3. The first parameter is the path to the mysqldump command. The second parameter follows the MySQL host (localhost) and port (3306), the database schema (test), and then the user name (root) to use and its associated password (rootpw).

4. Write down the resulting output: this is a unique fingerprint of the current database schema.

5. Optionally, connect to the database and make any change to the database schema, for example, add another table:

```
mysql> CREATE TABLE test.TableC (
    ID_C int(11) NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (ID_C)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

6. Rerun the same command:

```
$ java -jar dbhash.jar /usr/bin/mysqldump localhost 3306
rootpw
```

7. Compare the new hash value with the previous one. They will be completely different. This proves beyond doubt that the database schema was modified between the two runs of the hashing tool. If you dropped the table TableC and ran the program a third time, you would get the same hash value as the first run.

8. Before running the hashing tool again with a new schema version of your database, run the hashing tool and record the hash value together with the version number. By doing so, you can build up a collection of supported database schema versions, being sure to recognize them before running the hashing tool again.

## Managing Schemas

### How it works...

Contained in the `dbhash.jar` file is a program that does the following:

- ▶ Connect to the MySQL server by running the `mysqldump` command from location. It will automatically use the correct parameters to retrieve a "structure dump". No database contents will be read.
- ▶ Apply some processing to the retrieved dump. What this primarily does is constraint and index definitions to have them conform to a predictable order. "format normalization" is important because the cryptographic hashing function going to be used will produce different results, even when the only difference between two runs is in the ordering of those. `mysqldump` does not provide a completely predictable order regarding the output of such information. Most whitespace will be removed to make sure no platform-specific line endings influence the result.
- ▶ A SHA-1 hash of the normalized schema definition is calculated and printed. For more information on this algorithm, please refer to <http://en.wikipedia.org/wiki/SHA-1>. The main characteristic of any hash function is that even a slight change in the input will produce completely different and non-reversible results. This means that no two different inputs will ever have the same hash value. Therefore, by comparing the results of two runs against two database schemas, you know if they are perfectly identical or have even the slightest difference.

### There's more...

You can access the program's source code by downloading the `dbhash-src.zip` file from the book's website. You will find a `src` folder inside it with the full source code of the program.

### See also...

- ▶ *Identifying differences between two schemas*

# Good to Kn

In this chapter, we will cover:

- ▶ Avoiding silent replication disruption on full master disk
  - ▶ Maximizing usable memory on 32-bit Windows
  - ▶ Using separate temporary directories for multiple MySQL servers on a single machine, preventing conflicts
  - ▶ Non-availability of InnoDB may escape monitoring
  - ▶ Troubleshooting "Can't start server: Bind on TCP/IP port: No such file or directory"
  - ▶ Choosing character sets
  - ▶ Understanding auto-increment values

## Introduction

This appendix is a collection of several pieces of information we deemed important for a MySQL administrator to know, but which did not really fit the style of a recipe as well. However, many of the issues you will find here took us hours or sometimes days to figure out. We would like to spare you this time by making our experiences available here.

Naturally, you need not read this chapter from beginning to end—just as in the rest of the book, all items should be individually useful and understandable.

Good to Know

## Avoiding silent replication disruption on master disk

While using replication, you might experience corrupted or incomplete binlog files on the master when the disk they get stored on becomes full. In older versions, the file would have started and when the disk became full in the middle of an event being written, this data would be replicated and cause errors on the slaves.

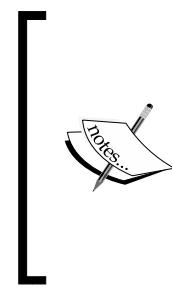
Versions 5.0 and up handle an out-of-space situation more gracefully, as they do not write partial statements to the binlog and try harder to keep the table data and the binlog in sync. However, you still need to be aware that there is no 100-percent sure way of preventing problems on the slaves because under certain circumstances you can end up with a statement having executed in the database, but not recorded in the binlog.

The master server will log this fact into its logfile, so your monitoring system might pick up with lines like:

The binary log <name> is shorter than its expected size.

but still the problem remains.

The MySQL manual has more details about this in chapters 5.2.4 at <http://dev.mysql.com/doc/refman/5.1/en/binary-log.html> and B.1.4.3 on disk-full problems at <http://dev.mysql.com/doc/refman/5.1/en/full-disk.html>.



The bottom line about this, however, is that even with the most conservative settings---`--sync_binlog=1` and `--innodb_support_xa=1`, leading to reduced performance due to more disk syncs—you can still end up with incomplete binlogs, be it on MySQL's part or the operating system's, requiring you to reset the replication and manually re-sync the slaves from a fresh backup.

Considering the performance penalties, the options MySQL offers to limit the risks of corruption of the binlogs and the fact that they do not guarantee problem-free operations anyway, I recommend investing your resources in a reliable system-monitoring solution that will keep you informed about critical conditions regarding disk space and allow you to prevent problems in the first place.

# Maximizing usable memory on 32-bit Windows

MySQL is a cross-platform piece of software, with versions available for all major operating systems and even embedded devices. Many enterprise-level servers nowadays are running MySQL on a variety of hardware and operating system combinations. However, at the time of writing this, there's still a large number of 32-bit systems in active use.

## Limitations of 32-bit systems

One of the major limitations of 32-bit systems is their inherent limit of a maximum address space per process, meaning that no process on such a system can ever access more than 4 GB of memory. However, on typical 32-bit operating systems, there is a lower limit in place. On Windows, this address space is split in half: 2 GB for the application and 2 GB of reserved addresses for kernel use. In this half of the address space, there are areas reserved for all sorts of hardware interfaces including graphics cards and expansion cards—precious addresses that cannot be used to address bytes in RAM chips.

In effect this means that any process—including a MySQL server—can use at most half of combined RAM on a 32-bit Windows system. This is even true for systems running, for example, Windows Server Enterprise Edition, which supports much more physical memory in a machine using some clever trickery. Even though in total they can support many Gigabytes of memory, the per-process limits still apply.

## Impact on MySQL/InnoDB

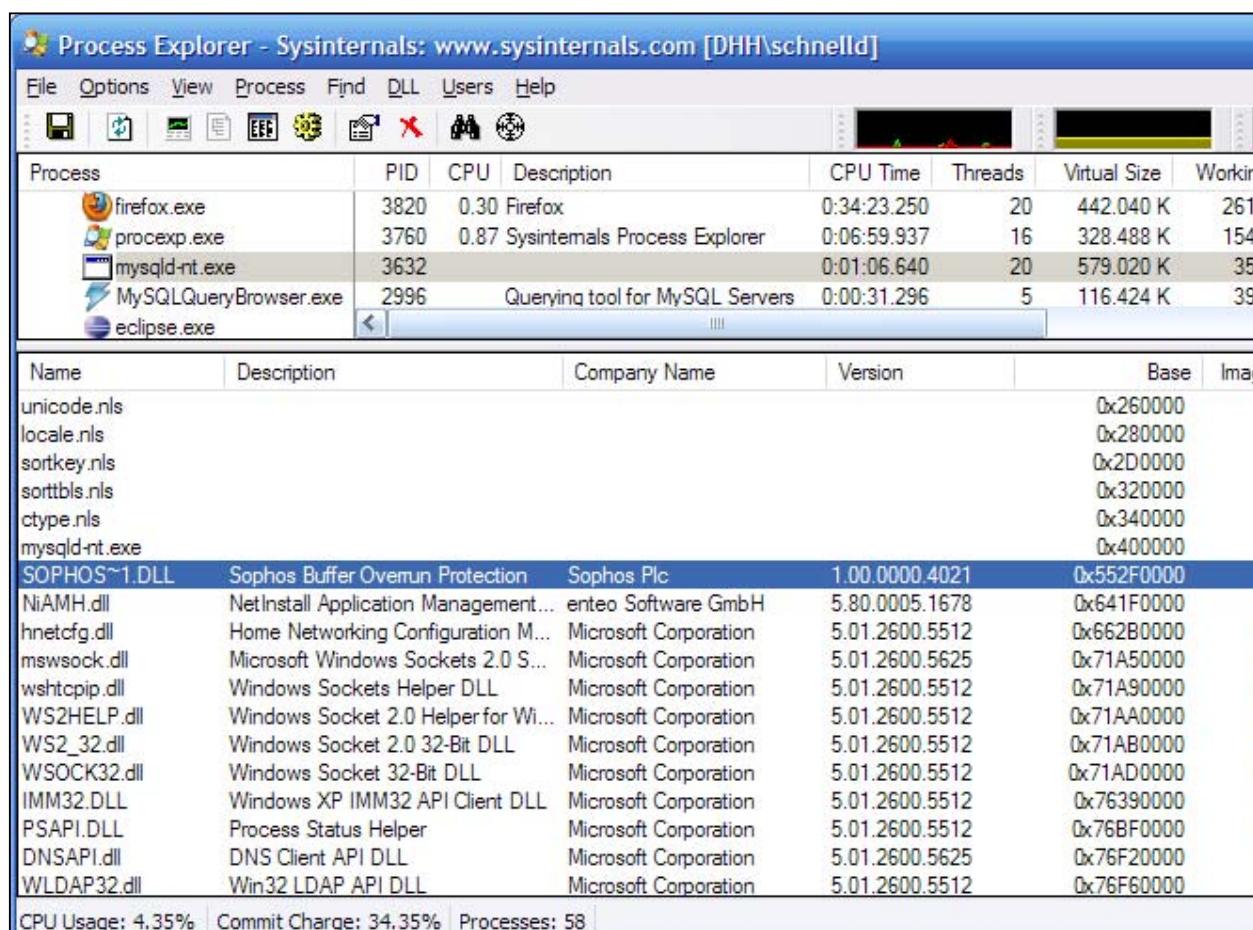
For a typical InnoDB-centric MySQL configuration, you would usually assign most of the available 2 GB per process to the InnoDB buffer pool (configuration setting `innodb_buffer_pool_size`). However, you will often notice that even though the total of all buffer pools is less than this limit, you might still end up with error messages, when trying to start MySQL that look like this:

```
C:\ CMD - Start_MySQL.cmd
091002 16:24:52  InnoDB: Error: cannot allocate 1572880384 bytes of
InnoDB: memory with malloc! Total allocated memory
InnoDB: by InnoDB 12483264 bytes. Operating system errno: 8
InnoDB: Check if you should increase the swap file or
InnoDB: ulimits of your operating system.
InnoDB: On FreeBSD check you have compiled the OS with
InnoDB: a big enough maximum process size.
InnoDB: Note that in most 32-bit computers the process
InnoDB: memory space is limited to 2 GB or 4 GB.
InnoDB: We keep retrying the allocation for 60 seconds...
```

## Good to Know

In this example, we tried a 1,500 MB buffer pool size. This kind of problem is regular by memory fragmentation. InnoDB tries to allocate a large, contiguous area of the address space for the buffer pool here. However, even before this takes place, the operating system has loaded shared libraries (DLLs) into the 2 GB user address space available to the application (0x00000000 - 0x7FFFFFFF) effectively splitting the available address range, so that InnoDB cannot get a large enough contiguous chunk for its buffer pool. Such libraries can be found in anti-virus solutions or management suites that need to hook on to all processes in a system.

The freely available Sysinternals ProcessExplorer tool (downloadable from Microsoft at <http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx>) can help you find out details in situations like this:



The screenshot shows the Process Explorer interface with the title bar "Process Explorer - Sysinternals: www.sysinternals.com [DHH\schnellD]" and the menu bar "File Options View Process Find DLL Users Help". The main window displays a table of processes:

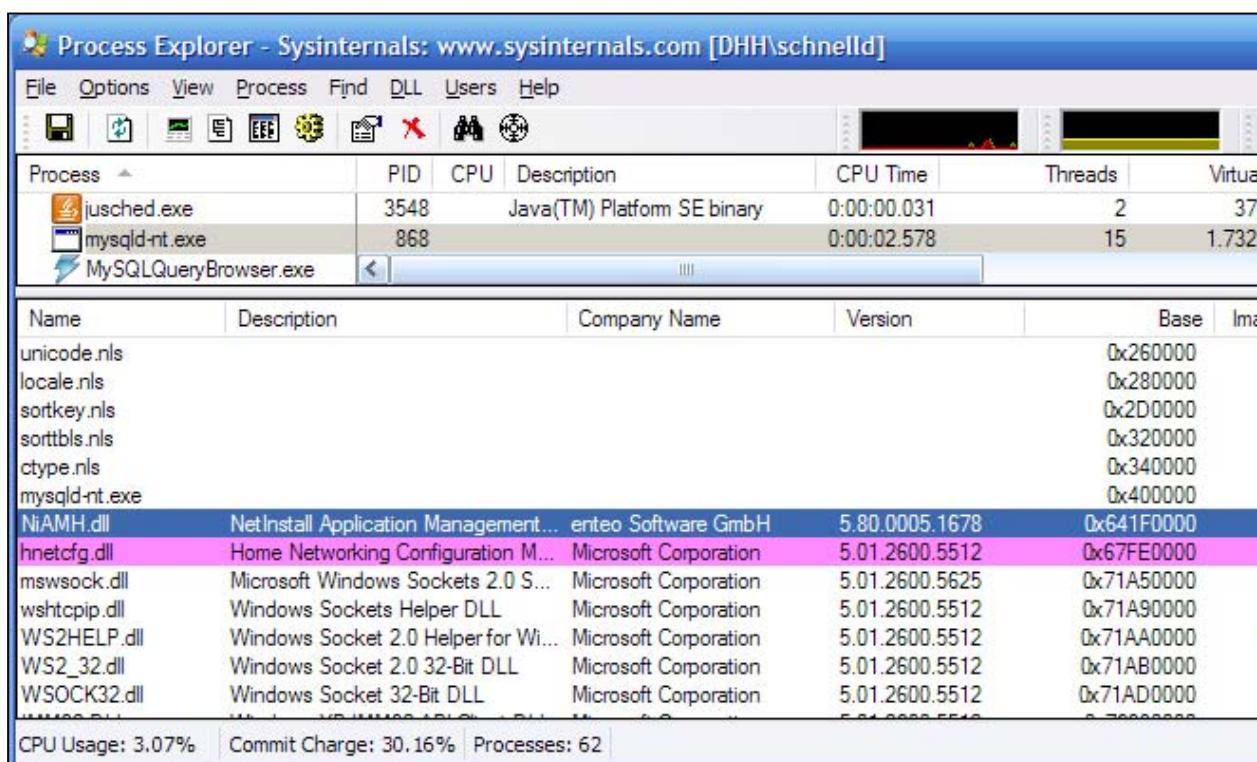
| Process               | PID  | CPU  | Description                     | CPU Time    | Threads | Virtual Size | Working Set |
|-----------------------|------|------|---------------------------------|-------------|---------|--------------|-------------|
| firefox.exe           | 3820 | 0.30 | Firefox                         | 0:34:23.250 | 20      | 442.040 K    | 261.000 K   |
| procexp.exe           | 3760 | 0.87 | Sysinternals Process Explorer   | 0:06:59.937 | 16      | 328.488 K    | 154.000 K   |
| mysqld-nt.exe         | 3632 |      |                                 | 0:01:06.640 | 20      | 579.020 K    | 351.000 K   |
| MySQLQueryBrowser.exe | 2996 |      | Querying tool for MySQL Servers | 0:00:31.296 | 5       | 116.424 K    | 39.000 K    |
| eclipse.exe           |      |      |                                 |             |         |              |             |

The lower pane shows a list of loaded DLLs:

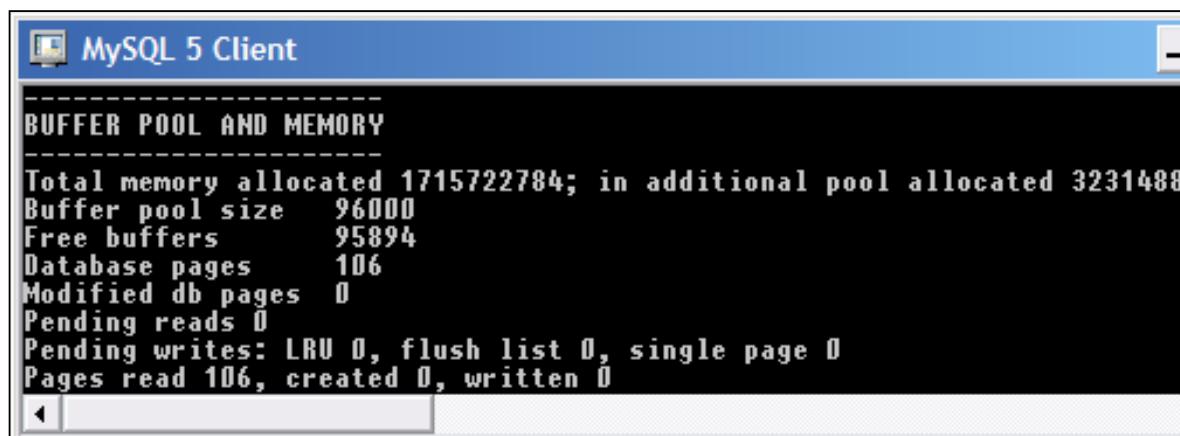
| Name          | Description                          | Company Name          | Version        | Base       | Image |
|---------------|--------------------------------------|-----------------------|----------------|------------|-------|
| unicode.nls   |                                      |                       |                | 0x260000   |       |
| locale.nls    |                                      |                       |                | 0x280000   |       |
| sortkey.nls   |                                      |                       |                | 0x2D0000   |       |
| sorttbls.nls  |                                      |                       |                | 0x320000   |       |
| ctype.nls     |                                      |                       |                | 0x340000   |       |
| mysqld-nt.exe |                                      |                       |                | 0x400000   |       |
| SOPHOST~1.DLL | Sophos Buffer Overrun Protection     | Sophos Plc            | 1.00.0000.4021 | 0x552F0000 |       |
| NiAMH.dll     | NetInstall Application Management... | enteo Software GmbH   | 5.80.0005.1678 | 0x641F0000 |       |
| hnetcfg.dll   | Home Networking Configuration M...   | Microsoft Corporation | 5.01.2600.5512 | 0x662B0000 |       |
| mswsock.dll   | Microsoft Windows Sockets 2.0 S...   | Microsoft Corporation | 5.01.2600.5625 | 0x71A50000 |       |
| wshtcpip.dll  | Windows Sockets Helper DLL           | Microsoft Corporation | 5.01.2600.5512 | 0x71A90000 |       |
| WS2HELP.dll   | Windows Socket 2.0 Helper for Wi...  | Microsoft Corporation | 5.01.2600.5512 | 0x71AA0000 |       |
| WS2_32.dll    | Windows Socket 2.0 32-Bit DLL        | Microsoft Corporation | 5.01.2600.5512 | 0x71AB0000 |       |
| WSOCK32.dll   | Windows Socket 32-Bit DLL            | Microsoft Corporation | 5.01.2600.5512 | 0x71AD0000 |       |
| IMM32.DLL     | Windows XP IMM32 API Client DLL      | Microsoft Corporation | 5.01.2600.5512 | 0x76390000 |       |
| PSAPI.DLL     | Process Status Helper                | Microsoft Corporation | 5.01.2600.5512 | 0x76BF0000 |       |
| DNSAPI.dll    | DNS Client API DLL                   | Microsoft Corporation | 5.01.2600.5625 | 0x76F20000 |       |
| WLDAP32.dll   | Win32 LDAP API DLL                   | Microsoft Corporation | 5.01.2600.5512 | 0x76F60000 |       |

At the bottom, status bars show "CPU Usage: 4.35%", "Commit Charge: 34.35%", and "Processes: 58".

In the previous screenshot, you can see a MySQL process that has its address space fragmented by two DLLs (see the lower pane)—one part of Sophos Antivirus and the other part of the NetInstall software management suite. Both DLLs take up only a small amount of memory for themselves; however, their position in the address space makes them problematic for InnoDB. The Sophos Buffer Overrun Protection Library is loaded at address 0x552F0000. This is only 1,358 MB from the start address of mysqld-nt.exe (0x40000) and prevents a 1,500 MB block from being assigned. If they were located at the far end of the address space, InnoDB could allocate a block of memory large enough to function. Compare this with the picture where the anti-virus software has been removed from the system:



In this setup, InnoDB can start just fine because there is a contiguous range free from 0x400000 to 0x641F0000, sized 1,597 MB. See the following screenshot showing the InnoDB status output section (produced by entering `SHOW ENGINE INNODB STATUS` at a MySQL command-line) telling you about the number of 96,000 available 16 KB pages in the buffer pool. This is exactly the 1,500 MB the configuration variable `innodb buffer pool size` is set to in this server's `my.ini` file.



Usually, we would recommend a switch to more capable 64-bit hardware operating system setups if these memory limits become an issue. However, there might sometimes be external factors preventing that.

## Good to Know

If you hit a problem like this, you have to try to decide whether either you can get rid of the problematic libraries completely (by not using the program they are associated with) or you should contact their respective vendors to find out if they can provide a different version of the same DLL that gets loaded into a more convenient address range.



### **Warning!**

Editing the registry can severely damage your system setup, up to the point of not being able to boot or access it at all; so make sure you know what you are doing. Get in contact with any software vendor whose libraries you disable to get clearance to try this; some libraries might be vital for your system to function!

We strongly recommend making a system backup before you proceed with this.

For testing, you can modify the `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs` registry key to control which DLLs are loaded when a process starts. After making sure this will not affect the stability of your system, remove the DLLs in question from this key and restart the MySQL server. You can now use a larger amount of memory.

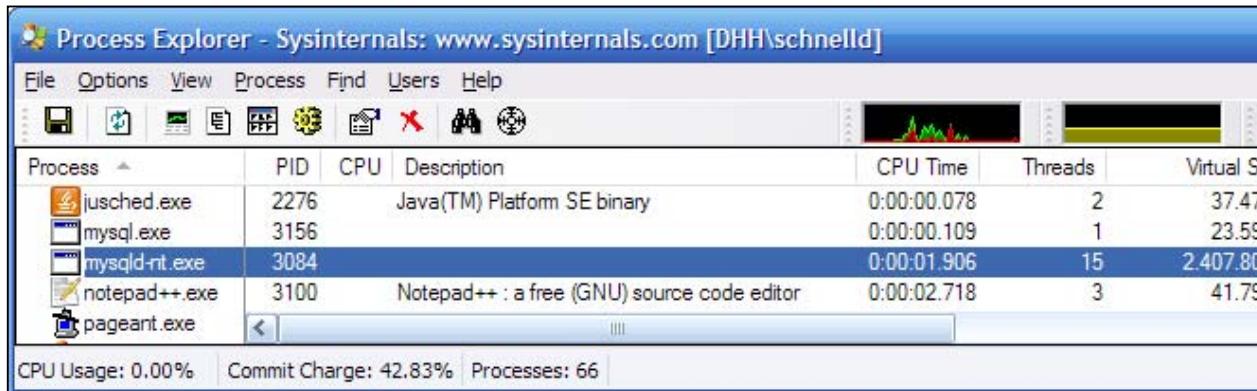
## **Getting even more with the /3GB switch**

To get even more memory available to MySQL on 32-bit systems, you might consider using the `/3GB` boot parameter. This parameter, which can be added to the `boot.ini` file of Windows that you intend to run programs that need more than 2 GB of address space.

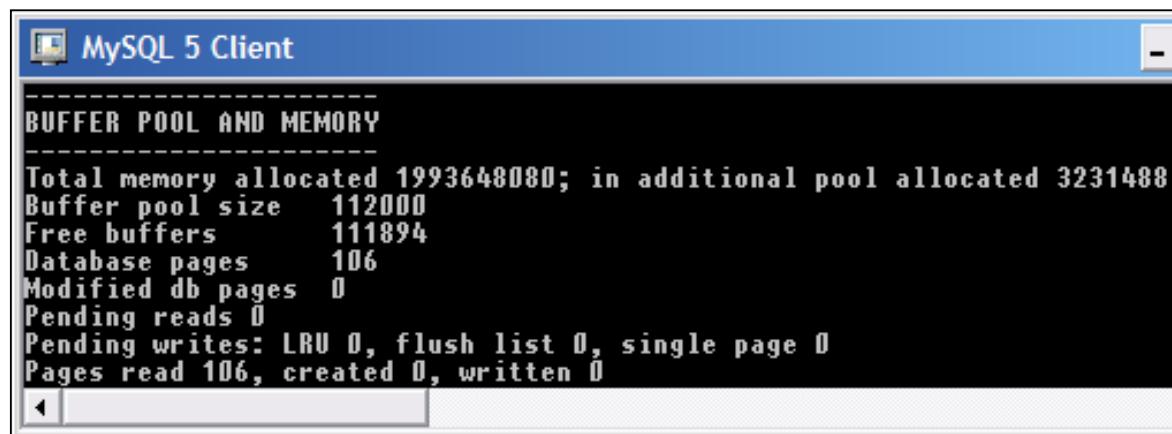
What it will do is present programs that are prepared for it (they need to be compiled with support for this) with a 3 GB address range for application use and reserve only 1 GB for kernel purposes. MySQL versions 5.0.79 and up and 5.1.33 and up are compiled to work from this configuration. However, using it can adversely affect your system in other ways. Refer to the Microsoft website or their support for more information.

Also note that this will not enable you to allocate more than 2 GB to the buffer pool; however, the MySQL server could use the extra space for other buffers.

The following screenshot was taken on a Windows machine that was booted with the `/3GB` option in place and shows a MySQL server process using more than 2 GB:



This is the InnoDB status output showing a 1,750 MB buffer pool on a 32-bit system.



# **Using separate temporary directories for multiple MySQL servers on a single machine, preventing conflicts**

Whenever you run multiple MySQL server instances, which are possible by configuring different sockets or TCP ports, you need to make sure to also configure a different directory for each instance.

This is not clearly documented and often it will work flawlessly, even if they share a temporary directory. If you do not specify a custom directory, the system's temporary will be used.

However, a regular behavior of the MySQL server is to clean up any remaining temporary files it may have created and left behind in a previous run. While this is generally a sensible implementation, the implementation is somewhat lacking. Upon start, each server process deletes its temporary directory whose file names start with #sql—regardless of who created them (files) in the first place (and assuming it has sufficient access rights).

## Good to Know

A problem now arises if one MySQL server is running and currently using a temporary file (which will have a name starting with #sql) while another instance comes up. In this situation, the second daemon will delete the file the first instance is currently using. Depending on the file system and operating system, this will cause issues. For example, if you try to run a SQL statement in the first, already running, server will fail with an:

```
ERROR 6 (HY000) : Error on delete of 'C:\WINDOWS\TEMP\#sql_4a1.MYI'
(Errcode: 2)
```

error because the temporary file has already been deleted by the other process.

The issue is discussed as MySQL Bug #47679 at <http://bugs.mysql.com/bug.php?id=47679>. At the time of writing this book, the only workaround was to configure distinct temporary directories for every MySQL server instance you start, using the `tmpdir` configuration variable in either the configuration file or as an additional command-line parameter.

## Preventing mysqldump from failing with Error 2013

This is a description of how to prevent mysqldump failures that are hard to explain and happen sporadically.

### Diagnosing the symptoms

When taking backups of a database containing large rows—usually with BLOB columns—users might sporadically experience error messages like:

```
mysqldump: Error 2013: Lost connection to MySQL server during
when dumping table `tablename` at row: 1342
```

One of the strange symptoms of this problem is that the row number may vary between runs of the identical statement on an unchanged set of data and that there seems to be something wrong with the records specified in the error message.

We ran into this problem time and again, but very infrequently, over the course of several months. Often restarting the dump would make the problem disappear, only to have it show up again after a seemingly unpredictable number of successful runs.

The problem was finally diagnosed and identified as documented in MySQL Bug #46103 at <http://bugs.mysql.com/bug.php?id=46103>.

## Finding the cause

When `mysqldump` runs, it will connect to the MySQL server using a network connection like any other MySQL client. As such it is subject to the usual settings, especially the network timeouts and the `max_packet_size` setting.

What may now happen with large table rows is that the `net_write_timeout` may be set to a time limit that is too short to transfer a whole data packet of `max_packet_size` length from the server to the client and write it to the disk there. From our experience, this might even happen on a loaded machine when `mysqldump` is connecting via local TCP.

To the MySQL server, this will look as if the client is not responding anymore and it will terminate the connection after `net_write_timeout` seconds, causing the error message shown earlier. As this problem is connected to server and network load factors, the error message can contain varying row numbers, making the problem even more difficult to understand at first glance.

## Preventing the problem

The fix is quite easy—configure the `net_write_timeout` value to a large enough value before running `mysqldump`, making sure that a full data packet can be transferred over the network and its contents be written to the SQL dump file:

```
$ mysql -e "SET GLOBAL net_write_timeout=120;"
```

This will give `mysqldump` two minutes to retrieve and store a single data packet, which will be plenty even for large BLOB columns.

The bug report #46103 is being kept open as a feature request at the time of writing this, so that `mysqldump` will request a long enough timeout automatically. Until this is implemented, you can use the workaround presented here.

## Non-availability of InnoDB may escape monitoring

If the MySQL configuration file specifies parameters that make InnoDB fail to start up, for example, too large memory values for the buffer pool—the server will start up with a particular storage engine, unless you have specified `default-table-type=InnoDB` configuration, too. If InnoDB is your default storage engine as per that parameter, the startup will fail when InnoDB is not available.

### Good to Know

One reason to be wary of this is that, if you are just monitoring if the server has been started and maybe do a simple query on the mysql database (or any other non-InnoDB table), then you might fail to notice that your server is not running correctly in time.

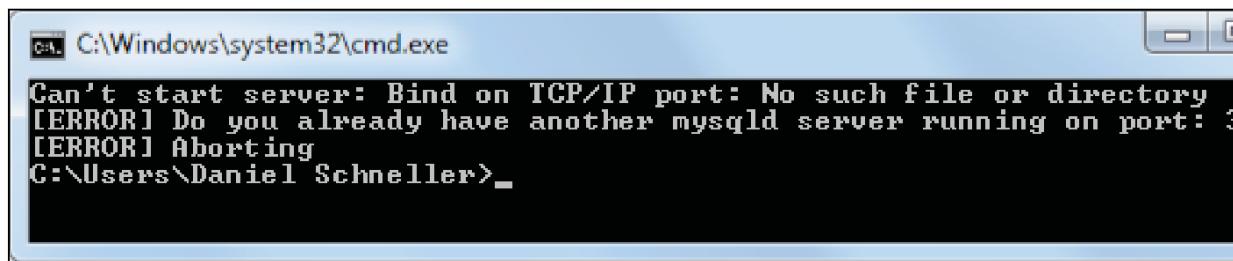
If you are using InnoDB primarily anyway, you should use the default-table-type parameter to make sure your server does not start up at all, if there is a problem with InnoDB.

## Troubleshooting "Can't start server: Bind on TCP/IP port: No such file or directory" errors

When running on Windows, MySQL under certain circumstances issues misleading error messages, which stem from its multi-platform programming. Windows has different ways of communicating underlying errors—for example, from the network subsystem—than Unix-like operating systems.

One such misleading error message is related to the startup process when the server tries to bind the TCP port it is configured to listen on and wait for client connections.

MySQL Bug #33137 (at <http://bugs.mysql.com/bug.php?id=33137>) explains the messages like:



```
C:\Windows\system32\cmd.exe
Can't start server: Bind on TCP/IP port: No such file or directory
[ERROR] Do you already have another mysqld server running on port: 3306?
[ERROR] Aborting
C:\Users\Daniel Schneller>_
```

We saw this error in our systems several times and were misled by the **No such file or directory** part. In fact, the TCP/IP port configured (usually 3306) was indeed already bound by another program at the time the message was created, but not anymore when we got to look at the situation.

Further analysis revealed that there was no user application using that port, but that the operating system itself was assigning this port as a local endpoint to processes that were connected to other network ports themselves. This is a regular mechanism and required for the TCP/IP protocol to work normally. The main problem arises from the fact that the MySQL network port 3306 is in a range called **ephemeral ports**, specifically allocated for this purpose the operating system used it for.

The range of these ports differs between operating systems, even though there is a recommendation by the **(Internet Assigned Numbers Authority)**.

---

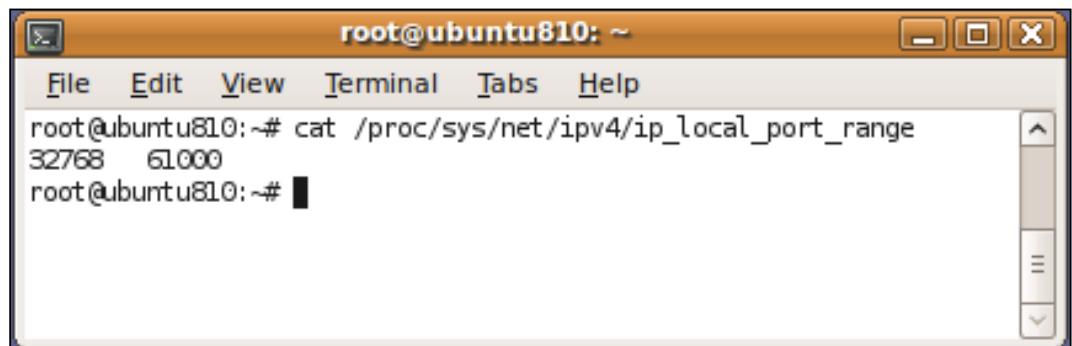
The IANA suggests 49152 to 65535 as "dynamic and/or private ports", according to Wikipedia. The following information is also taken from Wikipedia:

*Many Linux kernels use 32768 to 61000. The file system path /proc/sys/net/ipv4/ip\_local\_port\_range contains the range in use.*

*Microsoft Windows operating systems through Server 2003 use the range 1024 to 5000 as ephemeral ports. Windows Vista and Server 2008 use the IANA range 49152 to 65535.*

*FreeBSD uses the IANA port range since release 4.6.*

This is a screenshot of an Ubuntu Linux default system setup:



```
root@ubuntu810:~# cat /proc/sys/net/ipv4/ip_local_port_range
32768 61000
root@ubuntu810:~#
```

This range does not conflict with the MySQL port, so usually no action is required here.

On Windows versions up to Windows Server 2003, however, you need to take action to prevent conflicts from happening.

On all modern Windows versions—starting with Windows 2000—you can block out ranges (even if they only cover a single port) from the ephemeral ports for applications.

Effectively, they are not reserved for anything specific, but just excluded from the dynamic allocation. To do so, create or edit the following registry value of type REG\_MULTI\_SZ:

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\ReservedPorts

In this value, specify port ranges in the format xxxx-yyyy with xxxx and yyyy being the lowest and highest port of the range to be reserved. To reserve a single port, just use the same values for both as seen below.

There can be multiple ranges, each on its own line. So for MySQL make sure there are 3306-3306 present.

To learn more about this, see Microsoft's Developer Network Site at [http://msdn.microsoft.com/en-us/library/ms737828\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms737828(VS.85).aspx).

Good to Know

---

## Choosing character sets

This recipe is not a step-by-step set of instructions to follow. Consider the information presented here as a list of topics that guide you towards a suitable configuration.

### Text around the world

Whenever you store textual data in a database in CHAR, VARCHAR, or TEXT columns (so virtually always)—you have to think about character sets and collations.

English does not have any special characters apart from the usual Roman letters we use every day. This makes it different from many other languages. For example, the French language includes special characters like â, é, ò, German texts will likely contain things like Ä, Ö, or ß that are easily forgotten when designing computer systems, unless you live in one of these countries. Of course, Chinese, Japanese, Korean, Russian, among others have to be considered, as they are being based on completely different alphabets (if they are alphabetic at all).

Nevertheless, in today's global and networked systems, it has become common for a database to be used by people from around the world, all of them expecting to be able to use their native language with all its subtleties.

### Character sets

Basically, a so-called *character set* is like a table that contains a mapping between each character a given human language makes use of and a numerical representation that the computer uses to store that character internally. This concept is analogous to say that in Morse code in which, instead of mapping a character to a number, it is matched to a sequence of short and long signals.

We will not dive too deeply into the theory of these character sets and their accompanying collations (how to sort and compare characters in any given language), as there is a wealth of information about this available elsewhere, including the MySQL online manual. For now suffice it to say that MySQL has good support for a wide variety of character sets and collations, and allows you to specify which ones to use on a per-server, per-database, per-table, and even per-column basis.

You can find all the details in the online manual's Chapter 9 on *Internationalization and Localization* at <http://dev.mysql.com/doc/refman/5.1/en/internationalization-localization.html>.

### Defaults

Even though you might not be immediately aware of it, any MySQL database schema you create or set-up contains information about the character set to use for the tables and columns it contains. When you do not tell it to do otherwise, it will just silently use the default settings.

---

Instead of a concrete step-by-step walkthrough style recipe, we will merely provide checkpoints to think about when you design a database. As is often the case with topics, there is no one answer that perfectly fits every scenario. However, you will find on what basic aspects to consider when deciding on the character configuration you want to use.

## Multiple levels of configuration

Be aware that due to MySQL's flexibility you do not have to decide on a single character set for all your data, but can go for multiple sets if needed. We do, however, recommend sticking to a single character set if at all possible because matters quickly become very complex and hard to maintain when dealing with **JOINS**, and different client programs, and so on.

For details on this, refer to the MySQL online manual, section 9.1.3 at <http://dev.mysql.com/doc/refman/5.1/en/charset-syntax.html>.

## Getting ready...

To find out what character sets are available to choose from, you can either refer to the MySQL online manual at <http://dev.mysql.com/doc/refman/5.1/en/charset-sets.html> or retrieve a list from your MySQL server. To get it, connect to the server and run this command:

```
mysql> SHOW CHARACTER SET;
```

This will output a table of character sets supported by your server with their names, descriptions, the corresponding default collation, and the number of bytes any single character will use up at most if stored in the database. We will get to this in a minute.

## How to do it...

Even though we said before that there is no quick answer, you may skip reading the rest of this recipe if you are not interested in getting too many details, but just want quick advice that might not be an ideal solution for you, but will work well and keep your options open for the future:



### For the impatient:

Use `utf8` as your default encoding for all your tables.

Following this advice will enable you to store any international text in your database correctly.

If, however, you would like to know more before making a decision, please read on.

Good to Know

---

## Determining required languages

The most important decision you have to make is whether you want (or have) to store contents in more than a single language. If you know for sure that a database is going to exclusively contain English words, matters are going to be rather easy. In this case, just use the default *latin1* character set. Be aware, however, that you might want to use a different collation from the default Swedish one.

For any other single-language content, go through the list of supported character sets if there is one for your language or family of languages. If in doubt, then read on.

## Choosing from Unicode character sets

In case you cannot or do not want to commit yourself to a single language or family of languages, you should probably choose one of the Unicode-based character sets. These are designed to handle many languages and mixed language content well. See below for details on which Unicode character set is most suitable for your needs.

## Deciding on a Unicode character set

MySQL supports two Unicode character sets: *ucs2* (which is the predecessor to *utf8mb4* but still referred to with the older name in much of MySQL's documentation) and *utf8mb4*. They are suitable to store any character that has been defined in Unicode. The main difference between them is the amount of space they require for a single character. With *ucs2*, characters are uniformly stored as a two-byte sequence, whereas with the *utf8mb4* character set, the amount of storage required for a character depends on the individual character.

The UTF-8 encoding was designed to allow for a smooth migration from the common single-byte systems to more sophisticated and internationally usable software. The idea was to continue representing the most often used characters of the Western alphabet as a single byte, just as before, and dynamically use more than one byte per character for more "unusual" (meaning non-ASCII) symbols. A well-defined mapping algorithm was designed to be able to automatically map any not-so-usual character to a two-, three-, or most four-byte-long representation.

A positive and intentional effect of this technique is that an English or German text will require more space in *utf8mb4* (at least not significantly more) than if it were encoded in a single-byte character set like *latin1*. This is because there are only so many Umlauts in German text—none in English whatsoever—meaning that the bulk of the information to be stored with just one byte per character.

Moreover, any existing software program that can handle regular single-byte text in *latin1* will continue to work with UTF-8 encoded information, maybe just displaying non-ASCII characters incorrectly as two or more separate symbols.

The major downside is that you cannot tell the exact amount of space you need to store any UTF-8 encoded text in advance because depending on the characters used,

---

column, 40 bytes of storage space have to be reserved to be certain that there is enough room to store any sequence of 10 characters. This is not 100-percent exact, but good enough for our purposes here. See the MySQL online manual, section 10.5 at <http://dev.mysql.com/doc/refman/5.1/en/storage-requirements.html> for all details on storage requirements for each data type.

With `ucs2`, you have the benefit of being able to tell exactly how much space you will need to store any given text, provided you know how many characters it has. This means that `CHAR(10)` is always 20 bytes defined, as say `CHAR(10)` will use 20 bytes of disk space internally. For the regular Western language-based text this will usually be more than the equivalent `utf8` encoded version would take, but at least you can plan in advance.

A major drawback of `ucs2` (and any double-byte character set in general) is that most existing software products are not ready to process it because they were designed with a single byte per character in mind.

From all the information above, we recommend you use `utf8` in any case where English or any other Western language text will make up the bulk of the contents you are going to store in your database. This will result in the most space-efficient and yet compatible way of storing your information while preserving any international characters.

Only when you know in advance that the bulk of your contents will be in languages such as Arabic or from those, most notably languages from the Middle East and the Far East, should you consider using `ucs2` instead of `utf8` from a space-efficiency standpoint. However, be sure you are aware of the potential implications this has in terms of database client support. If in doubt, `utf8` is the safe choice here as well.

## **Considering conversion needs between server and clients**

The most important benefit of using `ucs2` instead of `utf8` on the MySQL server is that it reduces storage space requirements. However, MySQL for some reason does not support `ucs2` as the default character set for returning data to any client software. What this means is that even though `ucs2` characters might be stored with two bytes in the InnoDB table space or MyISAM data files, the MySQL server will not just send you those two bytes when you ask for that data. Instead, it will convert the data to a different encoding before sending the data across the network. Whatever the reason for this, it entails an additional burden on the server for any data entering or leaving a MySQL database.

To make matters worse from a processing-efficiency point of view, many modern systems internally use `ucs2` or its successor UTF-16 anyway (Java or Windows to name just a few). In theory, those could take the data verbatim from the database server and go on processing it. Instead, MySQL will convert the data from its internal `ucs2` format to, for example, `utf8` before sending it to the client, which in turn will then often convert it right back to `ucs2` before displaying it.

As a consequence, we (again) recommend you use `utf8` for database internal storage. If none of the national character sets fit your need, to save on the conversion effort when sending data back and forth between clients and the server. Only when you really need to do something special

Good to Know

## Understanding auto-increment values

In Chapter 9's *Allowing individual INSERT statements with "0" values in auto-incrementing columns* recipe, the `NO_AUTO_VALUE_ON_ZERO` option to the `SQL_MODE` system variable was used. To fully understand what was happening here, we suggest you to follow this little experiment.

### Getting ready...

Follow the preparations described in *Allowing individual INSERT statements with "0" values in auto-incrementing columns* (Chapter 9). Once you are done, connect to a test database and drop a possibly existing enumerator table (as used in the recipe mentioned above).



Be careful not to harm a production database; do this on a test system!

### How to do it...

1. Create the database schema afresh:

```
mysql> DROP TABLE IF EXISTS enumerator;
mysql> CREATE TABLE enumerator (
    id INT NOT NULL AUTO_INCREMENT,
    textvalue VARCHAR(30),
    PRIMARY KEY (id)
) ENGINE=InnoDB;
```

2. Try to insert and read back some data like this:

```
mysql> INSERT INTO enumerator
VALUES (0, 'Zero'), (1, 'One'),
(2, 'Two'), (3, 'Three');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

3. See if anything was actually inserted:

```
mysql> SELECT * FROM enumerator;
Empty set (0.00 sec)
```

4. Obviously nothing happened, as was to be expected because of the error we got.

- 
5. Try the exact same INSERT statement again to increase confusion:

```
mysql> INSERT INTO enumerator
VALUES (0,'Zero'),(1,'One'),
(2,'Two'),(3,'Three');
Query OK, 4 rows affected (0.00 sec)
```

Records: 4 Duplicates: 0 Warnings: 0

```
ds@ubuntu810: ~
File Edit View Terminal Tabs Help
mysql> SELECT * FROM enumerator;
+----+-----+
| id | textvalue |
+----+-----+
1	One
2	Two
3	Three
5	Zero
+----+-----+
4 rows in set (0.03 sec)
```

## What just happened...

When the table was just created, its auto-increment value was reset to be 1 for the record to be inserted. When we tried to insert the first batch of records shown previously, the (0, 'Zero') record was actually interpreted as a request to assign a new auto-increment value for the `id` column. As this was going to be the first record, MySQL actually tried to insert a (1, 'Zero') record.

However, the second record we tried to insert as part of our statement was (1, 'One'), which conflicted with the `id` value that had just been generated for the first row, making the entire INSERT statement fail.

Even though no records were inserted, MySQL increased the internal counter for the auto-increment value once for each record we tried to insert. Issuing the same INSERT statement again will work after that because the first record (0, 'Zero') is now translated to an auto-incremented value (5, 'Zero'), not creating a conflict for the other values.



This behavior can cause subtle errors if you do not notice what is going on right away because you might be working on data different from what you expect! This is one of the reasons why in general it is considered bad practice to insert your own values for auto-incrementing columns; so strive to avoid it if possible!

Good to Know

## There's more...

You can define the behavior shown here for a single session as the default behavior for the server. See the *Globally allowing INSERT statements with "0" values in auto-increment columns* recipe in Chapter 9 for more details on how to do that.

For more information on the `SQL_MODE` variable and its various settings, refer to the manual, section 5.1.8 Server SQL Modes at

<http://dev.mysql.com/doc/refman/5.1/en/server-sql-mode.html>.

In

## Symbols

- 7-Zip 153**
- 32-bit Windows usable memory**
  - InnoDB impact 337-340
  - limitations 337
  - maximizing 340
  - MySQL impact 337, 338
- \\_\\_ token 120**
- \h token 120**
- \d token 120**
- \u token 120**
- databases option 148**
- decompress option 160**
- hex-blob option 147**
- line-bytes option 165**
- lock-tables option 145**
- master-data option 163**
- single-transaction option 43**

## A

- access**
  - allowing, from group of hosts 295-297
  - allowing, from specific host 290-294
  - regaining 297-300
- alerting mechanism**
  - configuration, improving 208
  - establishing, for remaining InnoDB tablespace 205-207
  - MySQL scheduler, enabling 208
- alternative user**
  - defining, for administrative tasks 263-266
- ALTER privilege 62**
- ALTER TABLE ADD COLUMN command 307**

## ALTER TABLE command

- about 320
- performance, improving 318-320

## ALTER TABLE statement

- using 73

## AppArmor

- turning off 255

## auto-extending InnoDB tablespace

- about 230
- creating, from existing tablespace
- defining 230, 231
- extension steps, controlling 233
- new data file, adding 233
- setting up 231
- size, limiting 233
- working 232

## auto incrementing columns

- individual INSERT statements, allowing values 312
- INSERT statements, globally allowing values 313, 314

## AUTO\_INCREMENT option 311

### auto increment values

- about 350
- working 351

## automatically updated SQL based slaves, setting up

- getting ready 14
- steps 14
- working 15

## automatically updated SQL dump server slaves, setting up

- starting with 10
- steps 11, 12

**automatically updated slaves, setting up**  
 data backup, percona-xtrabackup used 18  
 data file conservation, LVM used 18  
 data file copy, using 15-17  
 getting ready 16  
 working 18

## B

**backup size, SQL dump**  
 command, working 151  
 compression ratio, achieving 152  
 compression utilities, using 153  
 data robustness, considering 152  
 getting ready 151  
 performance factors, considering 152  
 reducing 151  
 tool availability, considering 152

**basic user**

creating 268-271

**binary logging**

about 238  
 configuring 239  
 expire\_logs\_days setting 239  
 log\_bin parameter 239  
 max\_binlog\_size parameter 239  
 PURGE BINARY LOGS command 240  
 requirements 238  
 use 240  
 working 239

**binary logs**

about 153  
 data loss risks 156  
 data, restoring 163  
 disk space, ensuring 156  
 exact location of failure, restoring 165  
 point-in-time recovery, performing 163, 164  
 purging 154, 155  
 purging, in Linux systems 155, 156  
 rotating 154, 155  
 rotating, in Linux systems 155, 156  
 specific database, restoring 164

**binlogging.** See **binary logging**

**blackhole storage engine**

diagrammatic view 39  
 using 62

**BLOB columns**

using, for improving InnoDB table performance 324-326

**bzip2 153**

## C

**CHAR 74**

**character sets**

about 346  
 auto increment values 350  
 choosing 346, 347  
 conversion needs, between servers 349  
 defaults 346  
 global text 346  
 multiple levels of configuration 349  
 required languages, determining 349  
 Unicode character sets, choosing 349  
 Unicode character sets, decoding 349

**columns, finding**

about 220  
 name columns 220  
 numeric data types 222  
 requirements 220  
 with data type 221  
 working 222

**connection settings, sharing**

changes, dealing with 97  
 MySQL GUI Tools, working 96  
 requirements 95  
 steps 95

**CREATE INDEX command 62**

**curl tool**

using 43

**custom prompt, using to distinguish connections**

about 118  
 configuring 120  
 getting started 119  
 steps 119  
 tokens 120  
 working 120

# D

## **data**

DELETE command, using 192  
 deleting, from large tables 188  
 deleting, incrementally from  
     large tables 197-199  
 exporting, to CSV file 168  
 exporting, to custom file 172  
 importing, from CSV file 178  
 importing, from custom file 180  
 inserting, based on existing database content  
     185  
 managing 168  
 new data, inserting 183, 184  
 performance considerations 195  
 some records, retaining 192-195  
 stored procedures, using 174  
 updating 183, 184

## **database name letter case**

about 247  
 adjusting 247, 248  
 requirements 247

## **data, deleting from large tables**

Foreign key constraints, removing 191, 192  
 starting with 188  
 steps 189  
 TRUNCATE TABLE command, using 189  
 working 189, 190

## **data, exporting to CSV file**

error handling, with target file existing 170  
 getting ready 168, 169  
 headers, including 171  
 line breaks, handling 171  
 NULL values, handling 171  
 steps 169  
 working 169, 170

## **data, exporting to custom file format**

about 172  
 starting with 172  
 steps 172  
 working 173

## **data, importing from CSV file**

about 178  
 getting ready 178

working 179

**data, importing from custom file format**  
 about 180  
 getting ready 180  
 steps 180  
 working 181, 182

## **data files, copying**

backing up, LVM snapshots used  
 file-based backup data, restoring  
 file-based backup method, restricted  
 getting ready 141  
 steps 141  
 working 141, 142

**data insertion, based on existing content**  
 content

example 185  
 starting with 185  
 steps 185  
 working 186, 187

## **data management**

about 167, 168  
 data, exporting into CSV file 168  
 data, exporting to custom file format  
 data from large tables, deleting 191  
 data, importing to CSV file 178  
 data, importing to custom file format  
 data, updating 183  
 existing database content data, inserting  
 new data, inserting 183  
 stored procedures, using 174

## **data restoration, dump used**

binlogs, disabling temporarily 168  
 compressed dumps, restoring 168  
 getting ready 159  
 parallel restore, using 161  
 steps 159  
 tables, restoring 161, 162  
 working 159

## **DATE() function 304**

## **dbhash tool 333**

## **default accounts**

disabling 266, 267

## **default pager**

less pager utility, using 117  
 mysql, configuring 117

**diff command 328****domain searches**

- about 83
- getting started 84
- speeding up, steps 84
- working 84, 85

**duplicate indexes**

- disadvantages 86
- Index Analyzer, using 88
- searching 87, 88
- starting with 87
- working 88

**duplicate server IDs**

- about 50
- avoiding, steps 51
- getting ready 50
- symptoms, recognizing 52
- working 51

**E****encrypted MySQL console**

- creating 127
- creating, steps 128
- SSH, using 127
- working 129

**ephemeral ports 344****example, user creating in script without plain text password 301****F****fixed InnoDB tablespace**

- about 226
- checking 202-204
- installing 227
- setting up 227, 228
- working 228, 229

**free InnoDB tablespace, checking**

- requirements 202
- steps 202
- TABLE\_COMMENT column 204
- working 203, 204

**ft\_stopword\_file variable 68****fulltext index, adding**

- about 66
- boolean query mode, using 68
- case sensitivity 68
- dropping 67
- frequent words, ignoring 68
- precautions 67
- recreating 67
- Sphinx 69
- starting with 67
- steps 67
- stopwords 68
- word length 68
- working 67

**G****grep utility 116****gzip 153****H****hash values, schema version**

- using 331-334

**headers, including 171****I****IANA 344****indexes**

- adding, to tables 63
- differences 58
- drawback 58
- features 58
- fulltext index, adding 66
- InnoDB 60
- MySQL Query Browser used, for SQL statement generation 64
- naming 64
- prefix indexes 65
- prefix primary key, using 66
- removing, from table 73
- requirements 62
- storage 58
- working, with MyISAM storage engine 62

**INDEX privilege 62****InnoDB**

ALTER TABLE performance, improving 318  
advantages 61  
index space requirements, estimating 73-76  
primary index (clustered index) 61  
redo log, configuring 240  
secondary indexes 61, 62  
timeout configuration settings 243

**InnoDB data configuration, for storing in separate files**

starting with 234  
steps 234  
working 235

**innodb\_file\_per\_table feature**

using 237

**innodb\_lock\_wait\_timeout option values, settings 245****InnoDB primary key columns**

choosing 80, 81  
clustered indexes 83  
clustered index nature, considering 81  
getting ready 80  
immutability 81  
immutable attributes, identifying 80  
key length 82  
short keys, using 80  
single-column keys 80, 82  
unique attributes, identifying 80  
uniqueness 81  
working 81

**innodb\_rollback\_on\_timeout option value, settings 245****InnoDB tables**

alerting mechanism used 205  
autoextend feature 205  
drawbacks 22  
full text index providing, replication used 22-24

**InnoDB tablespace.** *See* **tablespace****INSERT statements**

allowing individual with 0 values 312  
globally allowing with 0 values 313, 314  
NO\_AUTO\_VALUE\_ON\_ZERO option, enabling

**installation user**

account, creating without using MySQL Administrator 275  
creating 272, 274

**Internet Assigned Numbers Authority****IANA****invalid date value**

storing in DATE, preventing 256-257  
storing in DATETIME, preventing 256-257

**J****Java Runtime Environment.** *See* **JRE 332****K****key length, InnoDB primary key column****L****largest table**

finding, requirements 219  
finding, steps 219  
working 219

**less utility 113****LIKE keyword 66****M****mk-duplicate-key-checker 89****mk-table-checksum tool**

about 48  
downloading 48

**more command**

using 164

**multiple MySQL server**

alternative MySQL Sandbox project 255  
AppArmor, turning off 255  
running parallel, on Linux server 255  
running parallel, on Windows 255  
SELinux, turning off 255

**Multi-Version Concurrency Control****MVCC 188**

**MyISAM storage engine**

- about 59
- drawbacks 60
- PRIMARY KEY index 60

**MySQL**

- binary logging 238
- indexes 57
- replication 7
- timeout configuration settings 243

**MySQL Administrator**

- configuring, for displaying global privileges 261, 262

**MySQL Administrator GUI tool**

- about 106
- additional backup options, exploring 140
- backups, scheduling 139
- custom graphs, adding 107-110
- getting ready 137
- limitations, handling 139
- using, as backup frontend 137, 138
- working 138

**MySQL command line client**

- information, extracting from verbose output 114-116
- pager command 113
- pager command, parameters 114
- using 112
- using, requirements 112
- using, to display query results page-by-page 111-113

**MySQL configuration**

- overview 225

**MySQL data backup**

- about 136
- data files, backing up 141

**mysqld\_multi tool**

- about 251
- working 254

**mysqldump command**

- hex-blob option 147
- about 144, 332

**mysqldump failure, preventing**

- cause, finding 343
- problem, preventing 343
- symptoms, diagnosing 342

**MySQL GUI Tools**

- config file locations 93, 94
- platform differences 93
- stored connection, sorting 98

**MySQL installation as Windows service****custom options used**

- about 248
- requirements 249
- steps 249, 250
- working 250

**MySQL installation, monitoring**

- free InnoDB tablespace, checking 12

**MySQL master server dump files**

- automatically updated slaves, setting 12
- multiple databases, replicating 11
- preparing 10
- working 12, 13

**MySQL server**

- encrypted connection, establishing 260
- encrypted connection, SSH used 260

**MySQL user management**

- about 259-261
- access, allowing from specific host 297
- access, regaining 297
- alternative user, defining 263
- basic user, creating 268
- default accounts, disabling 266
- installation user, creating 272
- MySQL Administrator, configuring 268
- plain text passwords, avoiding 300
- read-only account, creating 277
- specific user for backup, defining 266
- specific user for replication, defining 266

**MySQL variables**

- adapting 213
- changing 213-215
- global setting, displaying 216
- global variable 215
- multiple variables, displaying 216
- requirements 213-215
- session variables 215
- working 215

**N****net\_read\_timeout values, setting 245****network estimation**

- about 25
- compression, enabling 27
- master's network speed, checking 26
- requirements 25
- SLAVE\_COMPRESSED\_PROTOCOL option, using 27

**network limitation**

- alternatives, other than InnoDB 40
- in heavy scenarios, blackhole storage engine used 28-38

**net\_write\_timeout values, setting 246****new columns**

- adding, at specific positions 305-307

**NO\_AUTO\_VALUE\_ON\_ZERO option 313****normalized text search column**

- about 69
- creating 70, 71
- getting ready 70
- working 71, 72

**O****OPTIMIZE TABLE command**

- using 202

**overall table count, assessing**

- requirements 217
- steps 218
- working 218

**P****p7zip 153****plain text passwords**

- avoiding, in scripts 300, 301

**prefix primary keys**

- using 77, 78
- working 78, 79

**primary key**

- defining, for non-unique data table 308-311

**problematic queries**

- about 45
- alternative solution 47, 48

skipping, testing 45

working 47

**PUTTY template connection**

- about 130
- SSH connection, using 131
- using, steps 131, 132
- working 132, 133

**Q****query performance improvement, tables**

BLOB, using 324-326

**R****RBR 8****RDBMS 304****read access, sharing across multiple machines**

- about 19
- connection pooling, working with 20
- getting ready 20
- programming environments, working with 20
- slave addition efficiency, considering 20
- steps 20
- working 21

**read-only account**

- creating 277, 279
- creating, without using MySQL Administrator 280
- stored procedure calls, allowing 280

**redoing 243****redo log, configuring**

- requirements 240
- steps 241, 242

**Relational Database Management Systems 243***See RDBMS***replication**

- about 7, 8, 156
- getting ready 156, 157
- RBR 8
- SBR 8
- statement, executing 157
- using, to perform backups 157, 158
- using, to provide fulltext index for 158

**REPLICATION SLAVE privilege 12****rollforward 243****Row Based Replication.** *See RBR***S****SBR**

- about 8
- architecture 9
- features 8
- filtering 9, 10

**schema differences**

- identifying 329, 330
- identifying, steps 328, 329

**schema management**

- about 303-305
- ALTER TABLE performance, improving 318
- InnoDB tables query performance, improving 324
- INSERT statements, globally allowing with 0 values 313
- MySQL storage engine architecture 304, 305
- new columns, adding at specific positions 305
- primary key, defining for non-unique data table 308
- schema differences, identifying 327
- schema revisions, comparing 331
- storage engine, choosing 315
- stored performance, using 321

**schema revisions, comparing**

- dbhash.jar file programs, activities 334
- hash values, using 331-334

**SELinux**

- turning off 255

**server sync**

- checking, steps 49, 50
- mk-table-checksum tool, using 48
- mk-table-checksum tool, working 49

**SET GLOBAL command 216****SHOW SLAVE HOSTS command 52****SHOW TABLE STATUS command 89****silent replication disruption**

- avoiding, on full master disk 336

**SLAVE\_COMPRESSED\_PROTOCOL option****slave configuration**

- getting ready 53
- report-host setting 54
- report-password setting 55
- report-port setting 55
- report-user setting 55
- setting up, to update master 53

**slave I/O load estimation**

- about 25
- individual slaves' network speed, requirements 25
- SLAVE\_COMPRESSED\_PROTOCOL
- using 27
- working 26

**slaves, setting up via network streams**

- configuration, setting up 44
- curl tool, using 43
- data directory, compressing 43
- getting ready 41
- master.info file, understanding 41
- mysqldump tool, using 43
- START SLAVE statement, issuing steps 41
- temporary daemon 42
- temporary server, shutting down 42
- working 41

**source command 159****specific user**

- defining, for backup 281-285
- defining, for replication 285-289

**Sphinx 24****split tool 165****SQL configuration**

- for current session only 258

**SQL dump**

- backup size, reducing 151
- creating, of all databases 144
- creating, of specific tables 150
- creating, of specific databases 144

**SQL dump creation, of all databases**

- binary log position, including 147
- consistency, preventing 146
- consistent dumps, performing 146
- getting ready 144
- InnoDB table dumps, creating 144

- performing impacts, reducing 147
- steps 144
- working 144, 145
- SQL dump creation, of specific databases**
  - advantages 148
  - automated backups, side effects 149
  - getting ready 148
  - steps 148
  - working 148
  - work, parallelizing 149
- SQL dump creation, of specific tables**
  - getting ready 150
  - steps 150
  - working 150
- Statement Based Replication.** *See SBR*
- storage engine**
  - choosing 315, 316
  - silent engine substitution 317
- stored connection, MySQL GUI Tools**
  - automatically created, working 105
  - creating, automatically 102-104
  - sorting 101
  - sorting, requirements 98
  - sorting, steps 99
  - working 100, 101
- stored procedures**
  - defining 174
  - starting with 175
  - using, for column addition 321-323
  - using, for index addition 321-323
  - using, to export 175, 176
  - working 176
- T**
- table letter case**
  - about 247
  - adjusting 247, 248
  - requirements 247
- table reference**
  - about 223
  - finding 223, 224
  - requirements 223
  - working 224
- tablespace**
  - about 226
  - size, decreasing 236
- tablespace requirement, estimating**
  - about 209
  - requirements 209, 210
  - sample values, choosing 209
  - steps 210, 211
  - working 212
- temporary directories**
  - using, for multiple MySQL servers
- TEXT 74**
- timeout configuration settings**
  - applying, requirements 243
  - configuring 244
  - innodb\_lock\_wait\_timeout option
    - tings 245
  - innodb\_rollback\_on\_timeout option
    - settings 244
  - interactive\_timeout values, setting
  - net\_read\_timeout values, setting
  - net\_write\_timeout values, setting
  - wait\_timeout values, setting 245
  - working 244
- tools**
  - MySQL command line client 92
  - MySQL GUI Tools 92
  - official tools 92
- troubleshooting errors 344, 345**
- TRUNCATE TABLE command**
  - using 189
- U**
- UNLOCK TABLES command 143**
- user rights management**
  - permitting 275, 276
- V**
- VARCHAR 74**
- W**
- wait\_timeout values, setting 245**
- X**