sitepoint

JUMP START MYSQL

TIMOTHY BORONCZYK

MASTER THE DATABASE THAT POWERS THE WEB

Summary of Contents

JUMP START MYSQL

BY **TIMOTHY BORONCZYK**

Jump Start MySQL

by Timothy Boronczyk

Product Manager: Simon Mackie
English Editor: Ralph Mason
Technical Editor: Peter Nijssen
Cover Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9924612-8-7 (print)

ISBN 978-0-9941826-3-0 (ebook)
Printed and bound in the United States of America

About Timothy Boronczyk

Timothy Boronczyk is a native of Syracuse, NY, where he works as a senior developer at ShoreGroup, Inc. He's been involved with Web technologies since 1998, has a degree in Software Application Programming, and is a Zend Certified Engineer. In what little spare time he has left, Timothy enjoys hanging out with friends, speaking Esperanto, and sleeping with his feet off the end of the bed. He's easily distracted by shiny objects.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Preface

From "big data" data sets in an enterprise data center to hand-scribbled shopping lists, data is everywhere. Corporations collect as much of it as they can and analyze it to formulate new business strategies. Scientists study data looking for answers that can save lives, improve our environment, and explain our place in the universe. Even the average person maintains a fair amount of data, from ledgers detailing one's spending habits to phone numbers in a cellphone's address book. Storing and organizing all of this this data has become so easy that we often take for granted many of the database concepts and algorithms that make these things possible.

This book is an introduction to the basic concepts of working with a **Relational Database Management System** (RDBMS)—specifically, the popular, open source RDBMS MySQL. Like other installments in SitePoint's *Jump Start* series, it aims to give you a head start in your understanding of the chosen technology. You'll learn the basics quickly, in a friendly, (hopefully) pain-free way, and have a solid foundation to continue on in your learning.

I'm very grateful to have been given the opportunity to write this book. What separates it from others in the lineup is that it discusses a technology widely used both within *and* outside the world of web development. That's not to say MySQL isn't popular with developers creating web-based applications—quite the contrary! But databases are used in many other areas as well and I've tried to capture this in my selection of topics.

What is a Database?

Although we tend to associate the word **database** with the digital world of computers, the term simply refers to any organized collection of data. A database can therefore be digital/electronic or physical. The filing cabinet full of financial records that sits in the corner of your home office is a physical database. The cookbooks on your bookshelf, with their dog-eared pages and extra recipes clipped from magazines tucked inside, can also be viewed as a physical database.

In the digital world, databases are classified by how they organize and store their data. Some common types of digital databases are:

- **Flat file databases** — these store data sequentially, often in plain text files. They are easy to create and to add data to but they also have several drawbacks. Flat file databases are slow to search, may contain redundant data, and can easily become corrupted. An example of this type of database is the text file created by a solitaire game to store users' high scores.

- **Hierarchical database**s — these organize data in parent/child relationships. They are highly organized and searching is efficient, but hierarchical databases are difficult to navigate when you're not familiar with their relationships. Maintaining data relationships over time can be difficult as well. The Windows Registry is an example of a hierarchical database.

- **Key-value/document-oriented databases** — these store free-form data indexed by a key or hash value. They typically scale across wide network topologies very well but share many of the problems with flat file databases. They often contain redundant data, do not maintain relationships, and searching them can be slow. Redis and CouchDB are popular "NoSQL" database systems that manage these types of databases.

- **Relational databases** — these organize data in rows and tables, much like a printed price list or bus schedule can be organized as a table. Relational databases can support indexing large amounts of data for quick retrieval, but the relationships between tables can become very complex.

Sitting above most modern digital databases is a **database server**, an application designed specifically for managing databases, and which is responsible for marshaling access to the underlying data. We never work directly with a database in such systems. Instead, we send requests to add, update, remove, or fetch the desired data to the server. The server performs the requested actions on our behalf and forwards the results on to us. The book you're reading right now focuses on MySQL, a database server that manages relational databases.

Since the mid 1980s, **Structured Query Language** (SQL) has been the standard language used to communicate with relational database management systems. SQL consists of statements for adding, retrieving, and managing data, creating and maintaining tables, and even managing databases. Statements can be divided into categories or "sub-languages" based on their purpose: those pertaining to data storage and retrieval make up the Data Manipulation Language (DML), those for

table and database management make up the Data Definition Language (DDL), and those that grant or revoke access to the database make up the Data Control Language (DCL). It's good to know about these if they come up in conversation at your next database administrator cocktail party, but I don't make such fine distinctions here. I'll refer to DML, DDL, and DCL statements all collectively as SQL.

From Codd to MySQL, a Brief History

Early databases organized their data into tree or graph structures and accessing the data required a programmer to write code to directly traverse these structures. This was a fragile approach and it was risky to add or update data, or to change the data's organization. Edgar Codd challenged this approach in 1970 in his paper *A Relational Model of Data for Large Shared Data Banks*. He argued that a superior approach would be to organize data into tables and to treat it independently from relationship, ordering, and indexing information. This was an intriguing concept at the time and engineers at IBM's San Jose Research Laboratory began work on System R, a project to prove the validity of Codd's theories.

The System R project produced the first implementation of SQL and proved that the relational concepts championed by Codd were sound. When Larry Ellison heard about the research going into the System R prototype, he was so impressed that he incorporated Codd's ideas and the SQL language into his own database server, Oracle. Incidentally, Ellison beat IBM to market in 1979 and Oracle became the first commercially available relational database management system.

Meanwhile, computer science professors at the University of California, Berkeley, had also taken an interest in Codd's paper. The university obtained funding from the National Science Foundation and the research divisions of the United States Air Force and the United States Army and set a rotating team of students—led by Michael Stonebraker—to work on University INGRES. INGRES explored many of Codd's relational ideas, but also implemented its own query language called QUEL. As students graduated and went on to work at other software companies, commercial INGRES-inspired systems and clones appeared, most notably Sybase (later licensed to Microsoft and rebranded as Microsoft SQL Server). INGRES itself was commercialized and quickly became a market leader.

INGRES' position of dominance started to decline 1985 when public sentiment shifted in favor of SQL over QUEL. SQL was accepted as a standard by both the

American National Standards Institute and the International Organization for Standards by 1987, and the decade came to a close with Oracle and SQL on top.

In 1993, David Hughes was developing a network-monitoring application that stored data in a Postgres (a successor of INGRES) managed database. For portability, he also wanted to provide an SQL interface to the data so he wrote a QUEL-to-SQL translator which he named miniSQL. As work continued on his monitoring app, Hughes grew frustrated by Postgres' hardware requirements and decided to evolve miniSQL into his own light-weight database management system. miniSQL favored a small resource footprint over complete adherence to the SQL standards, implementing only the most important subset of the standards. Hughes distributed his system for a fraction of the cost that current commercial offerings were licensed at and miniSQL went on to become the first low-cost, SQL-based relational database system. The stage was now set for MySQL.

At that same time, Monty Widenius was developing web-based applications for the still-burgeoning Internet using UNIREG, his own home-grown database server. Widenius found that accessing UNIREG to generate dynamic pages was too resource intensive and began to look for an alternative. miniSQL piqued his interest, as it had grown very popular due to its pricing strategy—especially among shared hosting providers—but it didn't implement some of the features Widenius' applications needed. He ended up rewriting UNIREG for better performance, but also took the opportunity to reimplement its API to be compatible with miniSQL's. This would allow him to take take advantage of the many third-party utilities that had sprung up for miniSQL. Widenius renamed his server MySQL and a friend convinced him to release it publicly.

MySQL was made available under the GNU General Public License, and Widenius and his friends, David Axmark and Allan Larsson, founded MySQL AB in 1995 to shepherd the development of MySQL and provide alternative licensing and support for commercial customers. Whereas miniSQL was affordable, for most users MySQL was practially free.

Since the licensing terms for MySQL were amenable for inclusion in most Linux distributions, and because its API was compatible with miniSQL but made more features available, MySQL quickly ate most of miniSQL's market share. Today, MySQL is the second most popular SQL RDBMS (the number one spot is held by SQLite thanks in large part to its use in smartphones and embedded software).

Alternatives and the Future of MySQL

Sun Microsystems bought MySQL AB in 2008 for $1 billion, and in 2010, Oracle Corporation acquired Sun Microsystems and its assets (including MySQL) for $7.4 billion. The same company that beat IBM and INGRES in the 1980s now owned the copyrights to MySQL. And Oracle already had its own flagship database, so any fears the community had about the future of MySQL under Sun were only exacerbated by the Oracle acquisition.

But thanks to the GPL, anyone can make improvements and build upon MySQL, so long as those changes are properly licensed. This means others can make enhancements to MySQL, or even fork it, and release their own version. And forks there are!

- **Dorsal Source** — the first MySQL fork made by Proven Scaling in response to complaints over Sun's slow release process and the company handled community-submitted bug fixes and enhancements. The project is now defunct.
- **Drizzle**[1] — a fork of MySQL by Brian Aker with the goal of being a faster, pared-down version of MySQL specifically for supporting web applications. Core functionality is provided by a kernel and additional features are provided by plugins. The project isn't defunct, but development seems to have stalled.
- **Percona Server**[2] — a fork maintained by the consulting firm Percona LLC. Its goal is to be a drop-in MySQL replacement that offers improved performance and various enterprise-grade features not found in Oracle's Community edition.
- **MariaDB**[3] — a fork by Monty Widenius himself in response to the Sun and Oracle acquisitions. It aims to be a community-friendly replacement that maintains feature-parity for most use cases.

[1] http://www.drizzle.org/
[2] http://www.percona.com/software/percona-server
[3] https://mariadb.org/

Learn More about the Forks

To learn more about the MySQL forks, watch the talk "Different MySQL Forks for Different Folks[4]" given by Sheeri Cabral at Confoo in 2013.

The long-term outlook for the MySQL "brand" is strong despite tensions in the community. Oracle hasn't shuttered MySQL as many feared, and the quality of releases has actually improved under their stewardship. The forks provide competition, which hopefully is a good thing. Even in the most cynical sense, the past decade has seen an uptick in the use of open source in the enterprise setting so MySQL won't be going anywhere anytime soon.

Who Should Read This Book

This book is aimed at those interested in working with data and want to learn how to use MySQL. To get the most out of some parts of this book, you should have some previous programming experience, although no specific language is required.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout the book to signify different types of information. Look out for the following items:

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

[4] https://www.youtube.com/watch?v=dcWoHusSAsE

Where existing code is required for context, rather than repeat all the code, a vertical ellipsis will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/blogs/2015/05/28/user-style-she
➥ets-come-of-age/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

http://www.learnable.com/books/jsmysql1/

The book's website, which contains links, updates, resources, and more.

http://community.sitepoint.com/

SitePoint's forums, for help on any tricky web problems.

books@sitepoint.com

Our email address, should you need to contact us for support, to report a problem, or for any other reason.

Want to Take Your Learning Further?

Thanks for buying this book—we appreciate your support. Do you want to continue learning? You can now gain unlimited access to courses and ALL SitePoint books at Learnable for one low price. Enroll now and start learning today! Join Learnable and you'll stay ahead of the newest technology trends: http://www.learnable.com.

Chapter 1

Getting Started with MySQL

This chapter presents the first steps of getting started with MySQL. I'll show you how to install MySQL on both Linux and Windows systems, so be sure to follow along on the platform of your choice. Then you'll begin to get acquainted with MySQL's command-line client as we use it to connect to the database server and create our first database.

Often the first step of installing an application is to determine which version is appropriate, so it's worth noting that MySQL is available in several "flavors." From Oracle there is the freely available Community Edition and the paid commercial Standard, Enterprise, and Cluster Carrier Grade editions. The differences between Community Edition and the paid versions boil down to licensing and support contracts, some additional server plugins, and backup and monitoring utilities.

MySQL is open-source software released under the GNU General Public License so it should come as no surprise there are also alternative forks available. Two popular forks are MariaDB, a community-maintained "enhanced, drop-in replacement" for MySQL, and Percona Server, a drop-in maintained by the consulting firm Percona LLC. The differences between MySQL, MariaDB, and Percona are mostly imperceptible to the casual user.

You're free to use whichever flavor of MySQL you like, but to maintain focus and consistency I'll use Oracle's Community Edition version 5.6.23 (the current stable release at the time I'm writing this book). I'll also limit these instructions to Debian/Ubuntu, RedHat/CentOS, and Windows Server 2012. This list of operating systems covers the major platforms that MySQL is likely to run on in a production environment.

Local Development Environment

For readers looking to set up an installation for local development, I recommend creating a virtual machine using Oracle's VirtualBox[1]. You can install one of the aforementioned operating systems on the virtual machine and then install MySQL using this chapter's instructions. Not only does this give you the ability to work with a dev environment which can be configured as closely as possible to production without being tied down to a specific server or network, but also your local system remains clean from extra services and applications, whether your system is running Linux, Windows, or OS X.

Installing MySQL on Linux

Linux isn't a homogeneous platform and each distro has a preferred way to install software. In this section, I'll cover how to install MySQL on Debian/Ubuntu and Red Hat/CentOS systems using a package manager and how to compile and install MySQL from source. This will equip you with the necessary skills to handle most any Linux-based installation scenarios you may encounter.

Installing via a Package Manager

Most modern Linux systems use a package manager to make software installation a trivial task. And because it's so popular, chances are MySQL or one of its forks is available in your distro's package repositories. Debian/Ubuntu offers Oracle's MySQL Community Edition in their repos, and users can get up and running by simply typing `sudo apt-get install mysql-server`. Red Hat/CentOS repositories recently replaced MySQL with MariaDB; users can install MariaDB with `su -c 'yum install mariadb-server'`.

[1] https://www.virtualbox.org/

Installing software from a distro-maintained repository is fine for most users, but relying on these repos may not give you the most current release. Luckily, we don't have to give up the convenience that working with packages affords us. Oracle provides up-to-date RPM and DEB packages which can be installed using `rpm` and `dpkg`. They also maintain APT and Yum repositories and provide special packages to automatically add these repos to your system's list of known repositories.

The following steps register one of Oracle's repositories and install MySQL Community Edition from it. If your server isn't running a graphical interface and you can't use a text-based browser like Lynx, you'll need to complete the first four steps on another system and copy the file to your server.

1. Open a browser and navigate to the MySQL Repositories page at http://dev.mysql.com/downloads/repo.

2. Click the **Download** link for the **MySQL Yum Repository** or **MySQL APT Repository** depending on your platform's package manager. You'll be redirected to a page that lists various configuration packages.

3. Click the **Download** button next to the package appropriate for your system. For example, a Red Hat/CentOS 7 user should download the package **Red Hat Enterprise Linux 7 / Oracle Linux 7 (Architecture Independent), RPM Package**. An Ubuntu user using Trusty Tahr should download the package **Ubuntu Linux 14.04 (Architecture Independent), DEB**.

4. Oracle will try to trick you into signing up for an account. This isn't mandatory, so scroll down to the bottom of the page and click the link **No thanks, just start my download** to start the download.

5. Using a terminal window, navigate to the directory you downloaded (or copied) the package to and execute the appropriate command to install it:

   - Red Hat/CentOS users should run `rpm -i mysql-community-release-el7-5.noarch.rpm`.
   - Debian/Ubuntu users should run `dpkg -i mysql-apt-config_0.2.1-1ubuntu14.04_all.deb`.

6. The repository is now registered and you can install MySQL Community Edition with your package manager:

- Red Hat/CentOS users should run `su -c 'yum install mysql-community-server'`.
- Debian/Ubuntu users should run `sudo apt-get install mysql-server-5.6`.

Ubuntu users will be prompted during the installation process for a password for MySQL's root user (Debian and Red Hat/CentOS users will provide this password with a post-install command in the next step). MySQL maintains its own list of accounts separate from the user accounts on our system—that is, while the username may be the same, the MySQL root user isn't the same as the Linux root user.

Red Hat/CentOS users should run these post-install commands to set the password for MySQL's root user, register MySQL as a system service, and start a running instance (Debian/Ubuntu automatically registers and starts MySQL):

1. Set the root user's password for MySQL: `mysqladmin -u root password`.
2. Register MySQL to start when the system boots: `su -c 'chkconfig --level 2345 mysqld on'`.
3. Start the MySQL server: `su -c 'systemctl start mysql'`.

MySQL Community Edition is now installed on your system. For future reference, the following commands are used to start, stop, and check the running status of MySQL:

- **Start MySQL**
  - Ubuntu — `sudo service mysql start`
  - Debian — `sudo systemctl start mysqld`
  - Red Hat/CentOS — `su -c 'systemctl start mysql'`
- **Stop MySQL**
  - Ubuntu — `sudo service mysql stop`
  - Debian — `sudo systemctl stop mysqld`
  - Red Hat/CentOS — `su -c 'systemctl stop mysql'`

- **Query MySQL's running state**
  - Ubuntu — `service mysql status`
  - Debian — `sudo systemctl status mysqld`
  - Red Hat/CentOS — `su -c 'systemctl status mysql'`

A Simpler Future

Different commands are used to start, stop, and monitor MySQL because Ubuntu uses Upstart and the other distros use systemd. The Ubuntu developers plan to migrate to the systemd init system starting in 15.04. By the time 16.04 LTS rolls out, the commands to perform these tasks will be the same as those on Debian.

Installing from Source

It's becoming less and less common for system administrators to compile software from source code, but doing so often gives complete control over an application's features, optimizations, and configuration settings. As you might expect, it's also the most involved installation method.

The following steps show how to download the MySQL Community Edition source code, compile it, and install it. Again, if you don't have access to a graphical interface or text-based browser on the server then you'll need to complete the first few steps on another system and copy over the download.

1. Open a browser and navigate to the MySQL Community Downloads page at http://dev.mysql.com/downloads.
2. Click the **MySQL Community Server** link to be taken to the Download MySQL Community Server page. The various platform options are filtered by the drop-down labeled **Select Platform**.
3. Set the drop-down to **Source Code**, scroll down to the **Generic Linux (Architecture Independent), Compressed TAR Archive** entry, and click its **Download** button.
4. An Oracle account isn't mandatory for continuing with the download. Scroll to the bottom of the page and click the link **No thanks, just start my download** to begin the download.

5. Using a terminal window, create a new user account dedicated solely to running the MySQL server:

```
sudo groupadd mysql
sudo useradd -r -g mysql mysql
```

6. Navigate to the directory you downloaded the source archive to. Extract the archive and change into the code's directory:

```
cd /tmp
gzip -cd mysql-5.6.23.tar.gz | tar xvf -
cd mysql-5.6.23
```

7. Generate the build scripts by running `cmake`. I don't specify any options below, but a full list of configuration options can be found in the online documentation[2].

```
cmake .
```

8. Run `make` to compile MySQL, and then with elevated privileges run `make install` to copy the resulting binaries, utilities, libraries, and documentation files to their new home on your system:

```
make
sudo make install
```

9. Make sure the installed files are assigned the correct ownership and access permissions:

```
sudo chown -R mysql /usr/local/mysql
sudo chgrp -R mysql /usr/local/mysql
```

10. MySQL's data directory and system tables need to be initialized by the `mysql_install_db` script found in the installation's `scripts` directory. The script

[2] http://dev.mysql.com/doc/refman/5.6/en/source-configuration-options.html

uses paths relative to the installation directory, so invoke it from the installation directory rather than the `scripts` directory or somewhere else:

```
cd /usr/local/mysql
sudo scripts/mysql_install_db --user=mysql
```

11. Start MySQL and set its root user's password:

```
sudo mysqld_safe &
mysqladmin -u root password
```

The installation of MySQL itself is complete, but there's still some additional system configuration tasks you should consider. I recommend adding the installation's `bin` directory to the `PATH` environment variable so you can run MySQL's utilities without providing a full path each time. Assuming you use Bash, add the following lines to `/etc/profile`:

```
PATH=/usr/local/mysql/bin:$PATH
export PATH
```

Working with PATH

Setting the value of `PATH` in `/etc/profile` makes the utilities conveniently accessible for all system users. If you only want your own account to have this ability then add the lines to your `~/.bash_profile` or `~/.bashrc` file instead.

It's also likely you'll want MySQL to start automatically when the system boots. These steps assume your system uses a SysV-style init process.

1. Place a copy of the `mysql.server` script found in the source code's `support-files` directory in your system's `init.d` directory and make the script executable:

```
sudo cp /tmp/mysql-5.6.23/support-files/mysql.server \
 /etc/init.d/mysql
sudo chmod 755 /etc/init.d/mysql
```

2. Create symbolic links that point to the script from the desired runlevels:

```
ln -s /etc/init.d/mysql /etc/rc3.d/S99mysql
ln -s /etc/init.d/mysql /etc/rc0.d/K01mysql
```

You can now run the command `sudo /etc/init.d/mysql start` to start MySQL and run `sudo /etc/init.d/mysql stop` to stop it.

Installing MySQL on Windows

Windows is a relatively homogeneous platform compared to Linux even though several versions of the OS are actively maintained at any given time by Microsoft. The instructions here target Server 2012, but may be more or less applicable to a desktop OS like Windows 8.

1. Open a browser and navigate to the MySQL Community Downloads page at http://dev.mysql.com/downloads.
2. Click the link for **MySQL Community Server** to be taken to the Download MySQL Community Server page. The various platform options here are filtered by the drop-down labeled **Select Platform**.
3. Set the drop-down to **Microsoft Windows** and click the **Download** button next to the appropriate Windows MSI Installer for your architecture, most likely 64-bit.
4. Scroll to the bottom of the page and click the link **No thanks, just start my download** to begin the download.
5. Navigate to the folder you downloaded the MSI file to and double-click the file to launch the installation wizard.
6. Advance through the wizard's welcome screen by pressing the **Next** button.
7. At the **License Agreement** screen, click the checkbox to accept the terms of the agreement, and press **Next**.
8. At the **Choose Setup Type** screen, choose **Typical**, then press the **Install** button to begin the installation. You may be prompted by User Account Control to proceed depending on the security policies in effect.
9. Press the **Finish** button once the wizard is finished.

Now follow these post-install configuration steps to add the installation's `bin` directory to the system `PATH` variable and register MySQL as a service.

1. Open the **System Properties** window.
   a. Press the key combination **WIN-C** to bring up the Edge UI.
   b. Click the **Search** charm, search for Control Panel, and click on the **Control Panel** icon when it appears in the results.
   c. If Control Panel is in **Category** view, click the **System and Security** entry and then **System** to launch the System panel item. If Control Panel is in **Icon** view, click the **System** icon.
2. Click the **Advanced systems settings** link to open the **System Properties** window.
3. Select the **Advanced** tab if it's not already selected and then press the **Environment Variables** button to open the **Environment Variables** window.
4. Select the **Path** entry in the **System variables** section and press the **Edit** button.
5. Add the `bin` directory's path (`C:\Program Files\MySQL\MySQL Server 5.6\bin`) to the end of the existing value, separating the entry from the previous entries with a semicolon.
6. Open Command Prompt with administrator privileges. Depending on the security policies in effect, you may be prompted by User Account Control to continue.
   a. Press the key combination **WIN-C** to bring up the Edge UI.
   b. Click the **Search** charm and search for Command Prompt.
   c. Right-click the **Command Prompt** icon when it appears in the results and select **Run as administrator**.
7. Run `mysqld.exe --install` at the prompt. The command should report back the service was successfully installed.

You're now able to invoke the utilities when using Command Prompt without providing their full path because MySQL's `bin` directory appears in the list that Windows searches for executables. And since MySQL is registered as a service, it

will start automatically when the system boots and can be controlled from Windows Service Manager. Alternatively, the following commands may be executed in Command Prompt with administrator privileges to start and stop the MySQL server as well.

- **Start MySQL** — `net start mysql`
- **Stop MySQL** — `net stop mysql`

Communicating with the Server

A MySQL server sits idle, waiting to receive queries. When it receives one, the server performs the requested action on our behalf and responds back with the result. There are several ways we can communicate with MySQL, for example programmatically from an application we wrote or interactively using a dedicated client program. We'll use the command-line client that's included in the MySQL installation to connect and communicate with the running server throughout most of this book, and in Chapter 5 we'll discuss sending SQL statements programmatically.

Open a terminal window or Command Prompt and run `mysql -u root -p`. The `-u` option specifies the username of the MySQL account used for the connection and `-p` will prompt for the account's password. When prompted, enter the root account's password you set earlier.

Options Galore

`-u` and `-p` are just two of many options accepted by the client. Here's a list of some other options you may find yourself using frequently (you can call the client with the option `-?` for a complete listing):

- `-A` — don't re-initialize the auto-complete lookup
- `-B` — run in batch mode
- `-e` *`statement`* — execute the given SQL statement
- `-h` *`hostname`* — specify a hostname to a remote database server
- `-N` — suppress column names from the result output
- `-p` — prompt for the account's password to connect

- `-u username` — specify the username of an account to connect
- `-?` — list all of the available options

The client displays the `mysql>` prompt once you've successfully connected to MySQL. It's at this prompt we'll submit our SQL statements. The client displays the server's response, timing information for how long it took to execute the request, and whether any errors or warnings were encountered.

The MySQL server is capable of managing more than one database at a time. To ask what databases it's managing, enter `SHOW DATABASES;` at the prompt. The response will show a list of all the databases MySQL is managing. If you're connected to a newly installed instance then you'll only see the three databases that are used by MySQL itself: `information_schema`, `mysql`, and `performance_schema`. You may also see a `test` database which is created by `mysql_install_db` for use as a sandbox.

The `CREATE DATABASE` statement creates a new database. To create a database named "jumpstart", send the statement `CREATE DATABASE jumpstart;` at the prompt. Then send `SHOW DATABASES;` again, and you'll see the new database added to the list.

To let the client know we want to work with a specific database, we use the `USE` command. Enter `USE jumpstart;` at the prompt, and all subsequent statements we send will be executed against the `jumpstart` database. It's possible to specify a target database when connecting with the command-line client, for example `mysql -u root -p jumpstart`.

The `SHOW TABLES` statement instructs MySQL to return a list of tables in the currently active database. Of course, we haven't added any tables to the `jumpstart` database yet so sending `SHOW TABLES;` will be met with the response "Empty set." There's a fair bit of planning involved to create a table properly, and we've covered a lot already, so I'll save that for the next chapter.

To quit the client, either type `exit` or use the key combination **CTRL-D**.

MySQL Accounts and Security

The final thing I feel the need to cover in this chapter is MySQL user accounts. Even though MySQL's root user isn't the same as the system's root account, it's still not intended to be used on a regular basis. The MySQL root user should only be used

for administrative tasks such as creating new user accounts, setting permissions, and flushing access caches. Less privileged accounts should be used on a day-to-day basis.

To create a new user account, connect to the MySQL server with the command-line client using the root account and send the following `CREATE USER` statement:

```
CREATE USER 'jump'@'localhost' IDENTIFIED BY 'secret';
```

The statement creates a new account with the username “jump” and password “secret” that will permit the user to authenticate from the same system MySQL is running on. Different hostnames and IP addresses can be used in place of localhost to allow connections from different systems and networks. However, bear in mind that MySQL considers each username/hostname pair to be a separate account. That is, jump@localhost and jump@192.168.1.100 are treated as separate accounts, each with their own set of privileges.

Wildcards

The _ and % characters are wildcards that can be used in the hostname part to provide partial matches, for example “192.168.1.10_” or “%.example.com”. _ matches a single character and % matches any number of characters. Thus, the following can be used to create an account capable of authenticating from any system—a convenient but potentially very insecure practice:

```
CREATE USER 'jump'@'%' IDENTIFIED BY 'secret';
```

Whether MySQL permits a user to perform an activity depends on what privileges are associated with the account. New accounts are created without any privileges so we must explicitly grant any that the account will need. The "jump" user will require several privileges as you use it to follow along throughout the rest of this book. For now, let’s grant a basic set of privileges to start with (you can grant additional privileges as they become necessary). Enter the following statement:

```
GRANT CREATE, DROP, ALTER, INSERT, UPDATE, SELECT, DELETE,
INDEX ON jumpstart.* TO 'jump'@'localhost';
```

The syntax of MySQL's `GRANT` statement is flexible enough that we can narrow the scope of a privilege down to specific columns of a table, or to certain tables in a database. Here, we've simply instructed MySQL to allow these permissions for all tables (denoted by the `*`) in our `jumpstart` database. The privileges granted are:

- `CREATE` — allows the user to create databases and tables
- `DROP` — allows the user to delete entire tables and databases
- `ALTER` — allows the user to change the definition of an existing table
- `INSERT` — allows the user to add records to a table
- `UPDATE` — allows the user to update existing records in a table
- `SELECT` — allows the user to retrieve existing records from a table
- `DELETE` — allows the user to delete existing records from a table
- `INDEX` — allows the user to create or delete indexes

A full list of privileges and what they allow an account to do can be found in the documentation[3]. In the future, if it's determined an account needs extra privileges then they can be granted by issuing another `GRANT` statement. Privileges that are no longer needed can be revoked with a `REVOKE` statement, the syntax of which is identical to that of `GRANT`:

```
REVOKE CREATE, DROP, ALTER, INDEX ON jumpstart.* TO
'jump'@'localhost';
```

Whenever a user-related or privilege-related change is made, we need to send a `FLUSH PRIVILEGES` statement to instruct MySQL to reload the cache of account information it maintains so the updates can take effect. Otherwise, the changes will go unnoticed until MySQL is restarted:

[3] http://dev.mysql.com/doc/refman/5.6/en/privileges-provided.html

```
FLUSH PRIVILEGES;
```

Exit the command-line client after you send the `FLUSH PRIVILEGES` statement and reconnect using the new "jump" account. If you've entered the statements correctly, and provided the correct password when prompted, you'll be greeted with the `mysql>` prompt.

Conclusion

We've definitely covered a lot of ground in this chapter. You've learned how to install MySQL on various platforms, how to connect to a MySQL server using the command-line client, how to create a new database, and even a bit about basic MySQL user management.

Although you may be anxious to dive into the next chapter, I suggest you skim through the online MySQL manual first—specifically to see what it has to say on the topics we've covered so far. Review the details of the `CREATE USER` and `GRANT` statements. Learn how to change an account's password and how to delete an account that's no longer needed. Think about what privileges you'd assign to an account that needs to store and retrieve data as part of some back-end process for a website.

In Chapter 2, we'll get into the specifics of storing data in a database. I'll show you how to create a table and insert new rows into it. We'll also discuss what types of data can be stored in a table, what a storage engines is, and how our choice of engine affects the way MySQL manages our data.

Chapter 2

Storing Data

Data stored in a relational database is organized into **tables**. A database table organizes data in a grid-like fashion, where each entry forms a **row** and each **column** identifies a specific value in the entry. To illustrate this, here's a table showing the number of medals won by each of the top five medal-winning countries that participated in the 2014 Winter Olympic Games. Each row lists the country's name, how many gold medals, silver medals, and bronze medals were won, and the total number of medals won.

Country	Gold	Silver	Bronze	Total
Russia	13	11	9	33
United States	9	7	12	28
Norway	11	5	10	26
Canada	10	10	5	25
Netherlands	8	7	9	24

A table like the one above is "physical" in that we can see it printed in a book or drawn on a whiteboard. It's limited only by the amount of physical space available.

On the other hand, a database table is an intangible structure stored somewhere on a hard drive or in computer memory. We can only imagine it or make drawings to represent it. A database table is interpreted by a computer process (such as MySQL), and the limitations of the interpreting process impose restrictions on the table. The number of columns, the number of rows, and even what the individual values in a row can be, all depend upon what the computer system and database server can handle. But despite these limitations, a database table is actually very flexible. We can define relationships between tables, combine multiple tables together, sort rows and view specific entries, remove rows, and easily perform various calculations on the data.

In this chapter, we'll look at the `CREATE TABLE` statement—which defines new database tables—and discuss some important details surrounding table creation: MySQL's supported data types, naming restrictions, and storage engines. We'll also see how to add rows to a table with the `INSERT` statement, and finish by discussing transactions.

Creating Tables

Tables are created using the `CREATE TABLE` statement. In its simplest form, the statement provides the name of the table we we want to create and a list of column names and their data types. Not surprisingly, a `CREATE TABLE` statement can be very very complex depending on the requirements driving the design of the table. We can specify one or more attributes as part of a column's definition; such attributes can limit the range of values the column can store or specify a default value when one isn't provided by the user. Defining any logical relationships that exist between the table and another, and which storage engine MySQL should use to manage the table, is also common. You can see how detailed the statement can be if you look at the syntax and options for `CREATE TABLE` in the MySQL documentation[1].

Let's take a look at a pair of relatively simple `CREATE TABLE` statements. (I'll highlight some common points that add complexity, but I won't get too crazy, I promise.) With the `jumpstart` database created in Chapter 1 as your active database, issue the statements below. MySQL should respond "Query OK" after each one.

[1] http://dev.mysql.com/doc/refman/5.6/en/create-table.html

```
CREATE TABLE employee (
    employee_id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    last_name VARCHAR(30) NOT NULL,
    first_name VARCHAR(30) NOT NULL,
    email VARCHAR(100) NOT NULL,
    hire_date DATE NOT NULL,
    notes MEDIUMTEXT,

    PRIMARY KEY (employee_id),
    INDEX (last_name),
    UNIQUE (email)
)
ENGINE=InnoDB;

CREATE TABLE address (
    employee_id INTEGER UNSIGNED NOT NULL,
    address VARCHAR(50) NOT NULL,
    city VARCHAR(30) NOT NULL,
    state CHAR(2) NOT NULL,
    postcode CHAR(5) NOT NULL,

    FOREIGN KEY (employee_id)
        REFERENCES employee (employee_id)
)
ENGINE=InnoDB;
```

The first statement creates a table named `employee`, designed to store basic information about a company's employees—their name, email address, date of hire, and perhaps any notes the Human Resources director might provide. The formatting is just to keep things readable for ourselves; it makes no difference to MySQL whether we write a statement entirely on one line or across several lines with indentation. The spacing in a statement is also generally irrelevant.

Local Bias

The `address` table has a North American bias. An address in the United States or Mexico fits perfectly, and a Canadian address can store the two-letter province or territory abbreviation in the `state` column. But an address in the Netherlands, for example, needs space for a 6-character postal code. Feel free to adapt the definition to your own locale.

Names chosen for a table and its columns can be anything we like so long as they adhere to the following restrictions:

- The name uses basic Latin letters (A–Z, both uppercase and lowercase), the dollar sign (`$`), underscore (_), or Unicode characters U+0080–U+FFFF.
- The null character 0x00, Unicode characters U+10000 and higher, and characters that are prohibited in file names like slash (`/`), backslash (`\`), and period are not allowed in a name.
- The name must be quoted if it contains characters outside of the above. MySQL uses backticks by default for this (`` `...` ``) although it can be configured to use single quotes (`'...'`) as well. I recommend sticking with the default.
- The name must be quoted if it's a MySQL reserved keyword. A list of reserved words can be found in the online documentation[2].

The `employee_id` column is designated as the table's **primary key**. A primary key is a column in which all of the values are distinct and can be used to uniquely identify each and every row in the table. In more complex table definitions, we may define a primary key from multiple columns together, but using a single `INTEGER` type column is the most common practice. Only one primary key can be defined per table (hence the name *primary* key).

The `employee_id` column also has the `AUTO_INCREMENT` attribute. Whenever we add a row that doesn't provide a value for this column, MySQL will automatically use the next highest sequential integer as its value. Suppose we have a number of rows in the `employee` table and the largest `employee_id` value among them is 42. If we add a new row without an `employee_id` value, MySQL will use 43 for the missing value. If we then add another row without the value, MySQL will use 44, and so on. Only one column in the table can be designated an auto-increment column, and the column must also be a primary key.

Behind the scenes, MySQL maintains various data structures to track data and relationships. The `INDEX` defined on `last_name` lets MySQL know that we might use its value in our selection criteria later when we retrieve rows—for example, if we wanted to search for employees named Smith or Jones. MySQL will create and

[2] http://dev.mysql.com/doc/refman/5.6/en/reserved-words.html

manage a special index structure with the values in the column to make its search more efficient. Don't go overboard adding indexes though. It takes time for MySQL to maintain them so row retrieval may be faster, but adding/updating rows will be slower.

The term **constraint** describes a special condition imposed on a column or table that must be adhered to at all times. Most of the column definitions have `NOT NULL`, a constraint that prohibits storing `NULL` values in the column. `NULL` is a special value that represents the absence of a value. Essentially, `NOT NULL` means the column *must* hold a value. MySQL treats `NULL` differently from an empty value, such as an empty text string.

The `UNIQUE` constraint defined on the `email` column ensures all of the email addresses stored in the table are different. `UNIQUE` and `PRIMARY KEY` are similar, but there are important differences between them. Because the values in a primary key column must be able to unambiguously identify each row, its uniqueness is inherent. We don't explicitly specify `UNIQUE` with `PRIMARY KEY`. And while only one primary key can be defined per table, we can provide any number of `UNIQUE` constraints. A `UNIQUE` column may also contain `NULL` values, something `PRIMARY KEY` doesn't allow.

The `FOREIGN KEY` constraint in the `address` table's `CREATE TABLE` statement references the `employee` table, thus defining a relationship between the two tables. This relationship means that a row in the `address` table is logically related to whatever row in the `employee` table that has the same value in its `employee_id` column. Take, for instance, a row in the `address` table with an `employee_id` value of 42. That row may be associated with the row in the `employee` table whose `employee_id` value is also 42. In other words, an address with `employee_id` 42 is linked to employee 42's employee record. A `FOREIGN KEY` column doesn't need to have the same name as its partner column in the other table, but the two must share the same data type and `NULL` constraint.

We can issue `DESCRIBE` or `SHOW CREATE TABLE` statements to verify a table was created or view the definition of an existing table. The `DESCRIBE` statement returns the list of the table's column names and their data types, and `SHOW CREATE TABLE` returns a statement that can be used later to re-create the table.

```
DESCRIBE employee;

SHOW CREATE TABLE employee;
```

Pick a Convention

A convention I've adopted is to type MySQL keywords in uppercase and my own identifiers in lowercase. MySQL doesn't treat keywords and column names in a case-sensitive manner, but table names might be case-sensitive depending on the file system storing your tables' files. It's best to pick a convention—whatever it may be—and stick with it.

So far, we've discussed the column attributes and table constraints that appear in the example, but we haven't discussed the data types. The next part of this chapter may be a little dry, but it covers some important information. Each type requires a different amount of storage on disk and in memory so we always want to specify the minimum viable type for a column. The amount of wasted space from assigning a data type that's larger than necessary might be negligible at first because there's only a handful of rows, but it can add up quickly as more and more data is added to the table.

Data Types and Storage Requirements

MySQL supports many different data types, most of which we'll discuss in the following paragraphs. The term **data type** refers to the classification of data based on its possible values, the set of operations we can perform on it, and its storage requirements. Values of the `INTEGER` type can only consist of integers like 0, 42, and 1337. This is different from the `DECIMAL` type which consists of decimal numbers like 1.61, 3.14, and 100.0. We can perform operations like addition, subtraction, multiplication, and division on `INTEGER` and `DECIMAL` values, but these cannot be performed on text-based types like `CHAR` and `TEXT`.

Numeric Types

MySQL offers the `INTEGER` (also abbreviated as `INT`), `TINYINT`, `SMALLINT`, `MEDIUMINT`, and `BIGINT` data types for storing integer data. These types differ in the number of bytes they occupy to represent a value. This in turn limits the range of integers each type can hold. For example, `TINYINT` uses 1 byte, so its range is -128 to 127—the

range of numbers than can be expressed in binary with 8 bits. `INTEGER` uses 4 bytes, so its range is larger: -2,147,483,648 to 2,147,483,647.

We can also specify the `UNSIGNED` attribute with integer-based types. The type consumes the same amount of space but negative values are disallowed in exchange for raising the upper bound. For example, the range of `TINYINT UNSIGNED` becomes 0 to 255. Both `TINYINT` and `TINYINT UNSIGNED` represent a range of 256 integers, but their starting points are -128 and 0 respectively.

The following table shows the storage requirements and range for each of MySQL's integer types, both signed and unsigned:

Data Type	Storage Used (Bytes)	Min. Signed	Max. Signed	Min. Unsigned	Max. Unsigned
`TINYINT`	1	-128 127	127	0	255
`SMALLINT`	2	-32,768	32,767	0	65,535
`MEDIUMINT`	3	-8,388,608	8,388,607	0	6,777,215
`INTEGER`	4	-2,147,483,648	2,147,483,647	0	4,294,967,295
`BIGINT`	8	-9,223,372, 036,854,775, 808	9,223,372, 036,854,775, 807	0	18,446,744, 073,709,551, 615

`DECIMAL`, `FLOAT`, and `DOUBLE` are types that support real numbers. We also must provide the precision (the number of total digits) and scale (the number of digits that follow the decimal point) when we use one of these types. `DECIMAL(5,2)` has a range of -999.99 to 999.99—that is, five digits in total with two of them following the decimal point. We can specify `UNSIGNED` for these types as well, but doing so only disallows negative values. This is because the upper limit is defined by the precision and scale we provide.

The `DECIMAL` type is a fixed-point data type which means it preserves the exact precision of its value in calculations. This is useful for representing values like monetary amounts. The maximum precision we can specify for `DECIMAL` is 65, and the maximum scale is 30. On the other hand, `FLOAT` and `DOUBLE` are both floating-point types. Calculations with these types are approximate because some rounding

may occur due to how the values are represented internally in the computer. The difference between `FLOAT` and `DOUBLE` is the amount of space they occupy, which in turn affects their accuracy. `FLOAT` is 4-byte single-precision which is generally accurate up to 7 decimal places. `DOUBLE` is 8-byte double-precision which is generally accurate up to 15 decimal places.

The `BIT` data type stores a bit-sequence. This is useful for storing bit-field values like flags and bit masks. `BIT` has a capacity of 1 to 64 bits. `BIT(1)` can only hold 0 or 1; `BIT(2)` can hold the binary values 00, 01, 10, and 11; `BIT(3)` can hold the binary values 000, 001, 010, 011, 100, 101, 110, and 111, and so on. MySQL uses the notation `b'value'` to specify the value as string of binary digits, like `b'101010'`.

String Types

MySQL devotes several data types to storing textual data: `CHAR`, `VARCHAR`, `BINARY`, `VARBINARY`, `TEXT`, `TINYTEXT`, `MEDIUMTEXT`, `LONGTEXT`, `BLOB`, `TINYBLOB`, `MEDIUMBLOB`, and `LONGBLOB`. The sized types like `TINYTEXT` and `MEDIUMTEXT` behave exactly like `TEXT` although each is constrained by a different maximum amount of text it can hold. The same is true for `BLOB` and its sized counterparts, `TINYBLOB`, `MEDIUMBLOB`, and `LONGBLOB`.

We must provide a length when we specify a `CHAR` or `VARCHAR` type. `CHAR(255)`, for instance, stores text strings 255 characters long, and `VARCHAR(255)` stores strings up to 255 characters in length. Notice that I said "255 characters" and "*up to* 255 characters." `CHAR` is intended to store fixed-length strings, values that will always have the same number of characters across all rows in the table. The amount of space remains constant. `VARCHAR` stores variable-length strings, values that can have different lengths across the rows. The amount of space each value occupies is determined by the length of the string.

I'll highlight the difference between `CHAR` and `VARCHAR` using the string "Hello World". The string is 11 characters long, and it will occupy 11 bytes (plus an extra byte or two that MySQL needs to add for its own bookkeeping) if we store it in a `VARCHAR(255)` column. But with `CHAR(255)`, the storage space is constant across all rows in the table. MySQL pads the string with 244 spaces. The padding is removed when we retrieve the string and the original 11-character "Hello World" string is returned, but all `CHAR(255)` strings occupy 255 bytes when they're stored.