SQL

Pocket Primer

O. CAMPESATO

SQL

Pocket Primer

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and companion files (the "Work"), you agree that this license grants permission to use the contents contained herein, including the disc, but does not give you the right of ownership to any of the textual content in the book/disc or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to ensure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or disc, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Companion files for this title are available by writing to the publisher at info@merclearning.com.

SQL

Pocket Primer

Oswald Campesato

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Copyright ©2022 by Mercury Learning and Information LLC. All rights reserved.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai

Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
800-232-0223

O. Campesato. *SQL Pocket Primer.*
ISBN: 978-1-68392-814-0

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2022930720
222324321 This book is printed on acid-free paper in the United States of America.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at *academiccourseware.com* and other digital vendors. Companion files (figures and code listings) for this title are available by contacting info@merclearning.com. The sole obligation of Mercury Learning and Information to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

I'd like to dedicate this book to my parents
– may this bring joy and happiness into their lives.

Contents

Preface xvii

Chapter 1: Introduction to RDBMSs and MySQL 1

What is MySQL? 2
What about MariaDB? 3
Installing MySQL 3
Useful Links for MySQL 3
What is an RDBMS? 4
What Relationships Do Tables Have in an RDBMS? 4
Features of an RDBMS 5
What is ACID? 5
When Do We Need an RDBMS? 6
Transferring Money Between Bank Accounts 6
The Importance of Normalization 7
A Four-Table RDBMS 8
Detailed Table Descriptions 9
The Customers Table 10
The purchase_orders Table 11
The line_items Table 12
The item_desc Table 13
SQL Statements for the Impatient (Optional) 14
What About an Item Inventory Table? 17
The Role of SQL 17
DCL, DDL, DQL, DML, and TCL 18
SQL Privileges 18
Properties of SQL Statements 19
The CREATE Keyword 19
Data Types in MySQL 20
The CHAR and VARCHAR Data Types 20

String-Based Data Types 20
FLOAT and DOUBLE Data Types 21
BLOB and TEXT Data Types 21
MySQL Database Operations 22
Creating a Database 22
Display a List of Databases 22
Display a List of Database Users 23
Dropping a Database 23
Exporting a Database 23
Renaming a Database 24
Show Database Tables 25
The INFORMATION_SCHEMA Table 27
The PROCESSLIST Table 28
SQL Formatting Tools 29
Summary 29

Chapter 2: Working with SQL and MySQL 31

Drop Database Tables 32
Create Database Tables 32
Manually Creating Tables for mytools.com 32
Creating Tables via a SQL Script for mytools.com 34
Creating Tables with Japanese Text 35
Creating Tables from the Command Line 36
Defining Table Attributes 37
Working with Aliases in SQL 38
Alter Database Tables with the ALTER Keyword 39
Add a Column to a Database Table 39
Drop a Column from a Database Table 41
Change the Data Type of a Column 41
What are Referential Constraints? 43
Combining Data for a Table Update (Optional) 44
Merging Data Columns in Multiple CSV Files via Pandas 44
Concatenating Data from Multiple CSV Files 45
Appending Table Data from CSV Files via SQL 47
Inserting Data into Database Tables 48
Populating Tables from Text Files 49
Working with Simple SELECT Statements 51
Duplicate Versus Distinct Rows 52
Unique Rows Versus Distinct Rows 53
The EXISTS Keyword 53
The LIMIT Keyword 54
DELETE, TRUNCATE, and DROP in SQL 54
SELECT, DELETE, and LIMIT Combinations 55
More Options for the DELETE Statement in SQL 56

Creating Tables from Existing Tables in SQL 56
Working with Temporary Tables in SQL 57
Creating Copies of Existing Tables in SQL 58
What is a SQL Index? 58
Types of Indexes 59
Creating an Index 59
Disabling and Enabling an Index 60
View and Drop Indexes 60
Overhead of Indexes 61
Considerations for Defining Indexes 61
When to Disable Indexes on a Table 62
Selecting Columns for an Index 62
Finding Columns Included in Indexes 63
Enhancing the mytools Database (Optional) 63
Entity Relationships 64
Summary 65

Chapter 3: Joins, Views, and Subqueries 67

Query Execution Order in SQL 67
Joining Tables in SQL 68
Types of SQL JOIN Statements 68
Examples of SQL JOIN Statements 69
An INNER JOIN Statement 71
A LEFT JOIN Statement 72
A RIGHT JOIN Statement 72
A CROSS JOIN Statement 73
MySQL NATURAL JOIN Statement 73
An INNER JOIN to Delete Duplicate Attributes 74
JOIN Statements on Tables with International Text 75
What is a View? 76
Creating a View 77
Dropping a View in SQL 77
Advantages of Views in SQL Statements 77
Views Involving a Single Table 78
Views Involving Multiple Tables 78
Updatable Views 79
Keys, Primary Keys, and Foreign Keys 79
Foreign Keys versus Primary Keys 79
A MySQL Example of Foreign Keys 80
Working with Subqueries in SQL 82
Two Types of Subqueries 82
A Subquery to Find Customers Without Purchase Orders 83
Subqueries with IN and NOT IN Clause 85
Subqueries with SOME, ALL, ANY Clause 86

Subqueries with the MAX() and AVG() Functions 88
Find Tallest Students in Each Classroom via a Subquery 88
SQL and Histograms 90
What are GROUP BY, ORDER BY, and HAVING Clauses? 90
Displaying Duplicate Attribute Values 92
Examples of the SQL GROUP BY and ORDER BY Clause 92
SQL Histograms on a Table Copy 93
Combine GROUP BY and ROLLUP Clause 95
The 2021 Olympics Medals and the ROLLUP Keyword 97
The 2021 Olympics Medals and the RANK Operator 98
The PARTITION BY Clause 99
GROUP BY, HAVING, and ORDER BY Clause 100
Combined GROUP BY, HAVING, and ORDER BY Clause 101
Updating the item_desc Table from the new_items Table 102
A SQL Query Involving a Four-Table Join 102
Operations with Dates in SQL 106
Day and Month Components of Dates in SQL 107
Rounding Dates in SQL 108
Working with Date Ranges 109
Tables Containing Modification Times 110
Arithmetic Operations with Dates 111
Date Components and Date Formats 112
Finding the Week in Date Values 114
Displaying Weekly Revenue 114
Assorted SQL Operators 116
Working with Column Aliases 116
SQL Variables 117
SQL Summary Reports 118
Simple SQL Reports 119
Calculating SubTotals 122
Calculating “Running” (Cumulative) Totals 123
Summary 124

Chapter 4: Assorted SQL Functions 125

Numeric Functions in SQL 126
Calculated Columns 128
The round(), ceil(), and floor() Functions 129
SQL Queries with the rand() Function 132
Log, Exponential, and Trig Functions in SQL 132
Scalar Functions in SQL 135
Aggregate Functions in SQL 136
SQL Queries with the max() and min() Functions 138
Find Maximum Values with SQL Subqueries 139
Simplify SQL Queries Containing Subqueries 142

Find Top-Ranked Numeric Values 143
Find the Second and Third Largest Values in a Column 143
Find the Top Three Values in a Column 144
Find Values with the OFFSET Keyword 145
String Functions in SQL 146
SQL Queries with the SUBSTRING() Function 148
The SUBSTRING() Function in SQL 149
Boolean Operators in SQL 150
The IN Keyword 151
Set Operators in SQL 152
AND, OR, and NOT Operators in SQL 153
Working with Arithmetic Operators 154
Arithmetic Aggregate Operators in SQL 156
Finding Average Values 157
SELECT Clauses with Multiple Aggregate Functions 158
The ORDER BY Clause in SQL 159
ORDER BY with Aggregate Functions 160
Largest Distinct Values and Frequency of Values 161
Character Functions and String Operators 163
SQL Character Functions 164
String Operators in SQL 165
The MATCH() Function and Text Search 165
CTEs and the "with" Keyword in MySQL (Version 8) 166
The with Keyword and a Recursive SQL Query 168
CTEs and the Mean, Stddev, and Z-scores 169
Linear Regression in SQL 171
Window Functions 172
Types of Window Functions in SQL 173
The SQL CASE Clause 174
Working with NULL Values in SQL 176
Miscellaneous One-Liners 179
Working with the CAST() Function in SQL 181
Summary 183

Chapter 5: NoSQL, SQLite, and Python 185

Non-Relational Database Systems 186
Advantages of Non-Relational Databases 187
What is NoSQL? 187
What is NewSQL? 188
RDBMSs Versus NoSQL: Which One to Use? 188
Good Data Types for NoSQL 188
Some Guidelines for Selecting a Database 189
NoSQL Databases 189
What is MongoDB? 190

Features of MongoDB 190
Installing MongoDB 190
Launching MongoDB 190
Useful Mongo APIs 191
Meta Characters in Mongo Queries 192
MongoDB Collections and Documents 193
Document Format in MongoDB 193
Create a MongoDB Collection 193
Working with MongoDB Collections 195
Find all Android Phones 195
Find All Android Phones in 2018 196
Insert a New Item (document) 196
Update an Existing Item (document) 196
Calculate the Average Price for Each Brand 197
Calculate the Average Price for Each Brand in 2019 197
Import Data with mongoimport 197
What is Fugue? 197
What is Compass? 198
What is PyMongo? 199
MySQL, SQLAlchemy, and Pandas 200
What is SQLAlchemy? 200
Read MySQL Data via SQLAlchemy 200
Export SQL Data from Pandas to Excel 202
MySQL and Connector/Python 203
Establishing a Database Connection 204
Reading Data from a Database Table 204
Creating a Database Table 205
What is SQLite? 206
SQLite Features 207
SQLite Installation 207
SQLiteStudio Installation 208
DB Browser for SQLite Installation 209
SQLiteDict (Optional) 209
Summary 211

Chapter 6: Miscellaneous Topics 213

Managing Users 214
Listing Current Users 214
Creating and Altering MySQL Users 214
Dropping MySQL Users 215
What are Roles? 216
Create Roles and Grant Privileges 216
Revoke Roles and Drop Roles 218
What is a User-Defined Function? 218

What is a Stored Procedure? 218
- IN and OUT Parameters in Stored Procedures 219
- A Simple Stored Procedure 220

What is a Stored Function? 222
- A Simple Stored Function 222

What are SQL Triggers? 223
- A Simple MySQL Trigger 224

MySQL Engines 225

What is Normalization? 226
- What is Denormalization? 227

What are Schemas? 227

MySQL Workbench 228
- Exporting a Schema in Workbench 228
- Creating a Schema in Workbench 229
- ERM and Tools 230

What is a Transaction? 230
- The COMMIT and ROLLBACK Statements 231
- The SAVEPOINT Statement 231

Database Optimization and Performance 232
- Performance Tuning Considerations 232

SQL Query Optimization 233
- Analyzing SQL Queries for Their Performance 233
- Performance Tuning Tools 233
- Cost-Based Optimizers (Optional) 234
- Table Fragmentation 234
- Table Partitioning 234

What is an EXPLAIN Plan? 235
- Explain Analyze 236

Scaling an RDBMS 237
- What is SQL Tuning? 237
- What is Sharding? 238
- RDBMS Support for Sharding 238
- What is Federation? 239

Database Replication 239

Distributed Databases, Scalability, and the CAP Theorem 240
- Master-Slave Replication 240
- The CAP Theorem 240
- What are Consistency Patterns? 241

MySQL Command Line Utilities 241

Database Backups, Restoring Data, and Upgrades 241

MySQL and JSON Data 242

Data Cleaning in SQL 244
- Replace NULL with 0 244
- Replace NULL Values with Average Value 244

Replace Multiple Values with a Single Value 246
Handle Mismatched Attribute Values 247
Convert Strings to Date Values 248
Data Cleaning From the Command Line (Optional) 250
Working with the sed Utility 250
Working with the awk Utility 252
Next Steps 254
Summary 254

Appendix: Introduction to Probability and Statistics 257

Index 285

PREFACE

What is the Value Proposition for This Book?

This book is primarily for data scientists and machine learning engineers who want to expand their current knowledge of SQL using MySQL as the primary RDBMS. While this book does contain relevant information for novices in other fields, the structure of this book differs from typical database books.

In addition, this book attempts to balance depth and breadth, along with a decent number of SQL statements to illustrate the important features of SQL. Although it's not possible to describe the *exact* set of features that constitute basic, intermediate, and advanced SQL queries (i.e., opinions will differ), this book contains SQL examples that belong to each of those three groups.

At the same time, remember that some topics in the final chapter are presented in a cursory manner, which is for two main reasons. First, although you don't need an in-depth understanding of every facet of SQL and RDBMSs, it's important that you be aware of these concepts if you plan to become highly proficient in managing database data. In addition, you will be in a better position to plan an itinerary for the set of topics that you will learn at some point in the future.

Second, a full treatment of every topic in this book would significantly increase the page count, and it's debatable whether all the additional details would be beneficial to you as a machine learning engineer or a data scientist.

The Target Audience

As you read in the previous section, *this book is meant primarily for machine learning engineers and data scientists who already have a basic*

understanding of SQL, which means that they have executed some SQL statements in a database such as MySQL. As such, they will learn more details about SQL and MySQL so they can manage data in database tables. Moreover, the knowledge that they gain while working with MySQL can easily transfer to other RDBMSs such as ORACLE.

In addition, this book is intended to reach an international audience of readers, so this book uses standard English rather than colloquial expressions. As you know, many people learn by different types of imitation, which includes reading, writing, or hearing new material. This book takes these points into consideration in order to provide a comfortable and meaningful learning experience for the intended readers.

What's Different About This SQL Book?

Before delving into the differences, it's worth noting that this book covers many topics that you will find in database books of comparable length. At a minimum, any RDBMS book needs to include SQL, along with examples of how to select, delete, update, and insert data into a database table. Other mandatory topics include an explanation of views, indexes, joining tables, subqueries, normalization, and database schemas.

However, this book differs from generic database books because there are topics that are relevant to this target audience, which are not necessary for readers of generic database books. Some of those additional topics are discussed in chapter 6 (miscellaneous topics).

Another difference is a portion of Chapter 5, which contains Python-based code samples to access data from a MySQL table in a Pandas data frame. A third difference is the inclusion of the appendix that contains an introduction to probability and statistics, and a discussion of of entropy, cross-entropy, and KL divergence. Thus, it's the *collective* set of differences that differentiate this book from generic SQL books.

What Will I Learn From This Book?

The first chapter contains a short introduction to RDBMSs and MySQL, along with information about installing MySQL. In addition, you will see SQL statements for creating, dropping, and exporting a database. Although other books sometimes defer these operations to later chapters, they are easy to perform with empty or very small databases that do not contain any critical data. Therefore, you don't have to worry about making costly mistakes because of a blunder in a SQL query.

The second chapter delves into creating database tables and various ways to populate them with data. This chapter also describes various ways of deleting data from database tables, followed by a discussion of indexes on tables and why they are important.

The third chapter explains the concept of "joining" database tables, followed by a discussion of views: what they are, what advantages they provide, and how to create them over a single table or multiple tables. You will also learn how to work with subqueries in `SQL`. In addition, this chapter introduces you to the notion of normalization, along with a clear and compelling reason for adopting database normalization.

The fourth chapter is primarily about `SQL` functions, which involves numeric functions such as `ceil()`, `floor()`, and `random()`. Aggregate functions are also discussed, followed by string-oriented `SQL` functions such as the `substring()` function. This chapter contains an assortment of `SQL` statements, some of which involve various combinations of `GROUP BY`, `HAVING`, and `ORDER BY`.

The fifth chapter introduces `NoSQL`, followed by an overview of `MongoDB`, which is a popular `NoSQL` database. Next you will learn about `SQLite`, which is an open-source `RDBMS` that is available on mobile devices.

Chapter six contains a diverse set of miscellaneous topics, such as normalization, schemas, database optimization, and performance. Then you will be introduced to `EXPLAIN` plans, `SQL` tuning, managing users, roles, stored procedures, and triggers.

A Simple Way to Create the Entire mytools Database

As a convenience, Chapter 6 contains the `SQL` file `mytools.sql` that contains all the tables that are defined in this book. Moreover, the `SQL` file also contains the data for all the database tables. Of course, you can launch the individual `SQL` files for each of the tables if you prefer to do so the "long way".

You can import the complete `mytools` database by starting `MySQL` and then issuing the following command from the command line in the directory that contains `mytools.sql`:

```
mysql -u root -p mytools < mytools.sql
```

NOTE *If you encounter issues when you launch the preceding command, read the section in Chapter 6 regarding* `MySQL Workbench` *that enables you to import databases and export databases.*

What Do I Need to Know for This Book?

Although this is an introductory book with minimal prerequisites, obviously you will benefit from having existing knowledge of various topics. Specifically, some knowledge of `SQL` will facilitate learning the `SQL`-related concepts more quickly. In addition, knowledge of `Java` is helpful for Appendix A, as well as some familiarity with `XML` and `JSON`. Familiarity with normalization will help you understand the relationships among the tables in the fictitious application that is discussed in Chapter 1 and Chapter 2.

If you want to be sure that you can grasp the material in this book, glance through some of the code samples to get an idea of how much is familiar to you and how much is new for you.

Do the Companion Files Obviate the Need for This Book?

The companion files contains all the code samples to save you time and effort from the error-prone process of manually typing code into a text file. Furthermore, there are situations in which you might not have easy access to the companion files. In addition, the code samples in the book provide explanations that are not available on the companion files.

Does This Book Contain Production-Level Code Samples?

The primary purpose of the code samples in this book is to provide a variety of `SQL` statements that enable you to perform common and useful tasks in `MySQL`. Clarity has higher priority than writing more compact code that is more difficult to understand (and possibly more prone to bugs). If you decide to use any of the code in this book in a production website, you ought to subject that code to the same rigorous analysis as the other parts of your code base.

What Are the Non-Technical Prerequisites for This Book?

Although the answer to this question is more difficult to quantify, it's very important to have strong desire to learn about data analytics, along with the motivation and discipline to read and understand the code samples.

How Do I Set Up a Command Shell?

If you are a Mac user, there are three ways to do so. The first method is to use `Finder` to navigate to `Applications > Utilities` and then

double click on the `Utilities` application. Next, if you already have a command shell available, you can launch a new command shell by typing the following command:

```
open /Applications/Utilities/Terminal.app
```

A second method for Mac users is to open a new command shell on a MacBook from a command shell that is already visible simply by clicking `command+n` in that command shell, and your Mac will launch another command shell.

If you are a PC user, you can install `Cygwin` (open source *https://cywin.com*) which simulates bash commands, or use another toolkit such as `MKS` (a commercial product). Please read the online documentation that describes the download and installation process. Note that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as .bash_login).

Companion Files

All of the code samples and figures in this book may be obtained by writing to the publisher at info@merclearning.com.

What Are the "Next Steps" After Finishing This Book?

The answer to this question varies, mainly because the answer depends heavily on your objectives. If you are interested primarily in working with structured data, then you can look for online resources that delve into more advanced topics.

If you are primarily interested in machine learning, then you have several options: `NLP` (natural language processing), deep learning, and reinforcement learning (and also deep reinforcement learning).

Fortunately, you can perform an Internet search to find many resources. One other point: the aspects of machine learning for you to learn depend on who you are: the needs of a machine learning engineer, data scientist, manager, student or software developer are all different.

CHAPTER 1

INTRODUCTION TO RDBMSs AND MySQL

This chapter introduces you to RDBMSs and various SQL concepts, along with a quick introduction to MySQL. MySQL is used in most of this book because it is a robust RDBMS that is available as a free download from an ORACLE website. Current trends suggest that MySQL will continue its dominant role for the foreseeable future. Moreover, virtually everything that you learn about MySQL in this chapter transfers to other RDBMSs, such as PostgreSQL and ORACLE.

This chapter describes a fictitious website that enables users to register themselves for the purpose of purchasing various home improvement tools (hammers, wrenches, and so forth). Instead of SQL statements, you will learn about the tables that are required, their relationships, and the structure of those tables. You will also see some SQL `INSERT` statements for inserting data into database tables. Although we have yet to create any database tables, these SQL statements are intuitive and easy to grasp. Then, in Chapter 2, you will see the SQL statements that create the tables that are discussed in this chapter.

The first part of this chapter introduces the concept of an RDBMS, and the rationale for using an RDBMS. In particular, you will see an example of an RDBMS with a single table, two tables, and four tables (and much larger RDBMSs exist). This section also introduces the notion of database normalization, and how it assists you in maintaining data integrity ("single source of truth") in an RDBMS.

The second part of this chapter describes the structure of the tables in a four-table database that keeps track of customer purchases of home improvement tools that consumers can purchase through the associated Web page. You will also see the different relationships among pairs of tables, and how a one-to-many relationship enables you to find all the line items that belong to a given purchase order.

The third portion of this chapter contains a brief introduction to SQL and some basic examples of SQL queries (more details are in Chapter 2). You will also learn about the terminology for various types of SQL statements that can be classified as DCL (Data Control Language, DDL (Data Definition Language), DQL (Data Query Language), or DML (Data Manipulation Language).

The fourth portion of this chapter discusses SQL data types, and the fifth portion discusses database operations, such as creating, dropping, and renaming a database in MySQL. The final portion discusses two useful built-in tables that enable you to find the columns of a given table and the status of SQL statements.

There are several points to keep in mind before reading this chapter. First, the style for this chapter (and also the next chapter) is a "top-down" approach whereby high-level details are described and then hands-on coding details are discussed. However, you are free to reverse the order in which you read the first two chapters, if you prefer a "bottom-up" approach whereby you first learn more details regarding SQL statements and then learn about a use case in this chapter.

Second, there is an important detail that is mentioned in the preface that is worth repeating here: this book is primarily for data scientists who want to increase their knowledge of SQL to manage data in a database. Although this book can be useful for any motivated beginner, its primary purpose is different from books that prepare readers to become database administrators (DBAs).

Third, there is a section in the middle of this chapter that shows you the SQL statements that create several tables, along with details of purchase orders. This section is a preview of what you will learn in subsequent chapters, and it's intended primarily for readers who already have a good understanding of SQL statements. However, if you are unfamiliar with the syntax of the SQL statements in that section, there's no need to worry: you can return to this section after reading subsequent chapters that explain the details of the SQL syntax and functionality.

WHAT IS MYSQL?

MySQL is an open source database that is portable and provides many features that are available in commercial databases. Oracle is the steward of the MySQL database, and you can download MySQL 8.0 from the following site:

https://www.mysql.com/downloads/

MySQL is a highly popular database that is used by many companies, including Amazon, Google, LinkedIn, Netflix, and Twitter. MySQL is written in C++, whereas the user-level interaction is through SQL. Other add-ons for MySQL can be purchased from Oracle, as well as free third-party tools are available for monitoring and managing MySQL databases.

If you prefer, MySQL also provides a GUI interface for performing database-related operations. The code samples in this book have been written for MySQL 8, which provides the following new features beyond earlier versions:

- A transactional data dictionary
- Improved support for BLOB, TEXT, GEOMETRY, and JSON data types
- Support for CTEs (common table expressions)
- Support for window functions

As you will see in Chapter 6, MySQL supports pluggable storage engines, such as InnoDB (the most commonly used MySQL storage engine). In addition, Facebook developed an open source storage engine called MyRocks that has better compression and performance, so it might be worth while to explore the advantage of MyRocks over the other storage engines for MySQL.

What about MariaDB?

MySQL began as an open source project, and retained its name after the `Oracle` acquisition. Shortly thereafter, the MariaDB database was created, which is a "fork" of the MySQL database. Although MariaDB supports all the features of MySQL, there are important differences between MySQL and MariaDB that you can read about online:

https://mariadb.com/kb/en/mariadb-vs-mysql-compatibility/

Installing MySQL

Download the MySQL distribution for your machine and perform the installation procedure. After you complete the installation, log into MySQL as `root` with the following command, which will prompt you for the `root` password:

```
$ mysql -u root -p
```

If you installed MySQL via a DMG file, then the `root` password is the same as the password for your machine.

Useful Links for MySQL

This section contains various links that may be useful as you read the chapters of this book. Although SQL is not discussed in detail until the next chapter, the SQL links are included here for your convenience.

MySQL won the *DBMS of the Year* award in 2019:

https://db-engines.com/en/blog_post/83

The following link contains the list of platforms that support MySQL:

https://www.mysql.com/de/support/supportedplatforms/database.html

The following link contains a comparison between SQL and MySQL:

https://www.softwaretestinghelp.com/sql-vs-mysql-vs-sql-server/

A comparison of MySQL, Microsoft SQL Server, and PostgreSQL is available online:

https://db-engines.com/en/system/Microsoft+SQL+Server%3BMySQL%3BPostgreSQL

The latest version of SQL is SQL:2016:

https://en.wikipedia.org/wiki/SQL:2016

The following website contains details regarding MySQL Standards Compliance:

https://dev.mysql.com/doc/refman/8.0/en/compatibility.html

The following website describes MySQL Extensions to Standard SQL:

https://dev.mysql.com/doc/refman/8.0/en/extensions-to-ansi.html

The following website is a FAQ for MySQL 8.0, along with download links for the MySQL manual in multiple formats:

https://dev.mysql.com/doc/refman/8.0/en/faqs.html

WHAT IS AN RDBMS?

RDBMS is an initialism for Relational DataBase Management System. RDBMSs store data in tables that contain labeled *attributes* (sometimes called *columns*) that have a specific data type. Examples of an RDBMS include MySQL, ORACLE, and IBM DB2. While an RDBMS is software that manages data, a DBMS is the underlying "store" where the data resides.

Although relational databases often provide a very good solution for managing data, speed and scalability might be an issue in some cases. Chapter 5 discusses NoSQL databases, such as MongoDB, that might be more suitable for speed and scalability.

What Relationships Do Tables Have in an RDBMS?

While an RDBMS can consist of a single table, it often comprises multiple tables that can have various types of associations with each other. For example, when you buy various items at a food store, your receipt consists of one purchase order that contains one or more "line items," where each line item indicates the details of a particular item that you purchased. This is called a *one-to-many* relationship between a purchase order (which is stored in a `purchase_orders` table) and the line items (stored in a `line_items` table) for each item that you purchased.

Another example involves students and courses: each student is enrolled in one or more courses, which is a one-to-many relationship from students to courses. Moreover, each course contains one or more students, so there is a one-to-many relationship from courses to students. Hence, the students and course tables have a many-to-many relationship.

A third example is an `employees` table, where each row contains information about one employee. If each row includes the `id` of the manager of the given employee, then the `employees` table is a *self-referential* table because

finding the manager of the employee involves searching the `employees` table with the manager's `id` that is stored in the given employee record. However, if the rows in an `employees` table do *not* contain information about an employee's manager, then the table is not self-referential.

In addition to table definitions, a database frequently contains indexes, primary keys, and foreign keys that facilitate searching for data in tables and also connecting a row in a given table with its logically related row (or rows) in another table. For example, if we have the `id` value for a particular purchase order in the `purchase_orders` table, we can find all the line items (i.e., the items that were purchased) in the `line_items` table that contain the same purchase order `id`.

Features of an RDBMS

An RDBMS provides a convenient way to store data, often associated with some type of application. For example, later you will see the details of a four-table RDBMS that keeps track of tools that are purchased via a Web-based application. From a high-level perspective, an RDBMS provides the following characteristics:

- a database contains one or more tables
- data is stored in tables
- data records have the same structure
- well-suited for vertical scaling
- support for ACID (explained below)

Another useful concept is a *logical schema* that consists of the collection of tables and their relationships (along with indexes, views, triggers, and so forth) in an RDBMS. The schema is used for generating a *physical schema*, which consists of all the SQL statements that are required to create the specified tables and their relationships.

For example, Chapter 6 contains a SQL file `mytools.sql` that contains the definition of every entity in the `mytools` database, as well as the directory `mytools-sql-files-20211120` that contains a SQL file for every table in the `mytools` database. Moreover, Chapter 6 describes two techniques for exporting a MySQL database. After the tables have been generated, you can begin inserting data and then managing the consistency of the data.

What is ACID?

ACID is an acronym for Atomicity, Consistency, Isolation, and Durability, which refers to properties of RDBMS transactions, as summarized below.

- *Atomicity* means that each transaction is all-or-nothing, so if a transaction fails, the system is rolled back to its previous state.
- *Consistency* means that successful transactions always result in a valid database state.

- *Isolation* means that executing transactions concurrently or serially will result in the state.
- *Durability* means that a committed transaction will remain in the same state.

RDBMSs support ACID, whereas NoSQL databases usually do not support ACID.

WHEN DO WE NEED AN RDBMS?

The short answer is that an RDBMS is useful when we need to store records of events that have occurred, which can be involve simple item purchases as well as complex multi-table financial transactions.

An RDBMS allows you to define a collection of tables that contain rows of data, where a row contains one or more attributes (informally called *fields*). A row of data is a record of an event that occurred at a specific point in time, which can involve more than one table, and can also involve some type of transaction.

Transferring Money Between Bank Accounts

Consider a simple money transfer between two bank accounts in which you want to transfer $100 from a savings account to a checking account. The process involves two steps:

1. debiting (subtracting) the savings account by $100 and
2. crediting (adding) the checking account with $100.

However, if a system failure occurs after step 1 and *before* step 2 can be completed, you have lost $100. Obviously, steps 1 and 2 must be treated as an *atomic transaction*, which means that the transaction is successful only when both steps have completed successfully. If the transaction is unsuccessful, the transaction is "rolled back" so the system is returned to the state prior to transferring money between the two accounts.

As you learned earlier in this chapter, RDBMSs support ACID, which ensures that the previous transaction (i.e., transferring money between accounts) is treated as an atomic transaction.

Although atomic transactions are indispensable in financial systems, they might not be as critical for other systems. For example, a database that contains a lone `events` table in which each row contains information about a single event that you created by some process (such as a registration form) whenever a new event occurs in a system. Although this is conceptually simple, notice that the following attributes are relevant for each row in the `events` table: `event_id`, `event_time`, `event_title`, `event_duration`, and `event_location`, and possibly additional attributes.

As another example, displaying a set of pictures might not show the pictures in the correct order (e.g., based on their creation time). However, a failure in the event creation is not as critical as a failure in a financial system, and displaying images in the wrong sequence will probably be rectified when the page is refreshed.

THE IMPORTANCE OF NORMALIZATION

This section contains an introduction to the concept of *normalization*. As a starting point, consider an RDBMS that stores records for the temperature of a room during a time interval (such as a day, a week, or some other time interval). We just need one `device_temperature` table where each row contains the temperature of a room at a specific time. In the case of the IoT (Internet of Things), the temperature is recorded during regular time intervals (such as minute-by-minute or hourly).

If you need to track only one room, the `device_temperature` table is probably sufficient. However, if you need to track multiple devices in a room, then it's convenient to create a second table called `device_details` that contains attributes for each device, such as `device_id`, `device_name`, `device_year`, `device_price`, and `device_warranty`.

Whenever we want the details of a temperature-related event, we need information from *both* tables, which consists of one row in the `device_temperature` table and its associated row in the `device_details` table. The way to perform the two-table connection is simple: each row in the `device_details` table contains a `device_id` that uniquely identifies the given row. Moreover, the *same* `device_id` appears in any row of the `device_temperature` table that refers to the given device.

The preceding two-table structure is a minimalistic example of something called database *normalization*, whose purpose is to reduce data redundancy in database tables. Normalization can result in a slower performance during the execution of some types of SQL statements (e.g., those that contain a `JOIN` keyword).

If you are new to the concept of database normalization, you might be thinking that normalization increases complexity and reduces performance without providing tangible benefits. While this is a valid thought, the trade-off *is* worthwhile because normalization enables you to maintain data consistency.

For example, suppose that every record in the `purchase_orders` table contains all the details of the customer who made the associated purchase. As a result, we can eliminate the `customers` table. However, if we ever need to update the address of a particular customer, *we need to update all the rows in the* `purchase_orders` *table that contain that customer*. By contrast, *if we maintain a* `customers` *table, then updating a customer's address involves changing a single row in the* `customers` *table.*

Normalization enables us to avoid data duplication so that there is a single "source of truth" in the event that information (such as a customer's address) must be updated. From another perspective, data duplication means that the same data appears in two (or possibly more) locations, and if an update is not applied to all those locations, the database data is in an inconsistent state. Depending on the nature of the application, the consequences of inconsistent data can range from minor to catastrophic.

Always remember the following point: whenever you need to update the same data that resides in two different locations, you increase the risk of a data inconsistency, which can adversely affect the data integrity.

As another example, suppose that a site sells widgets online. At a minimum, the associated database needs the following four tables:

- `customer_details`
- `purchase_orders`
- `po_line_items`
- `item_desc`

The preceding scenario is explored in greater detail in the next section that specifies the attributes of each of the preceding tables.

A FOUR-TABLE RDBMS

Suppose that *www.mytools.com* sells tools (the details of which are not important). For simplicity, let's pretend that an actual website is available at the preceding URL and it contains the following sections:

- new user register registration
- existing user log in
- input fields for selecting items for purchase (and the quantities)

For example, the registered user `John Smith` wants to purchase one hammer, two screwdrivers, and three wrenches. The website needs to provide users with the ability to search for products by their type (e.g., a hammer, a screwdriver, or a wrench) and then display a list of matching products. Each product in that list would also contain an SKU, which is an industry-standard labeling mechanism for products (just like ISBNs for identifying books).

The preceding functionality is necessary in order to develop a website that enables users to purchase products. However, the purpose of this section is to describe a set of tables (and their relationships to each other) in an RDBMS, so we will assume that the necessary Web-based features are available at our URL.

Let's describe a use case that contains the sequence of steps that are performed on behalf of an existing customer John Smith (whose customer ID is 1000), who wants to purchase 1 hammer, 2 screwdrivers, and 3 wrenches:

Step 1: Customer John Smith (with `cust_id 1000`) initiates a new purchase.
Step 2: A new purchase order is created with the value `12500` for `po_id`.
Step 3: John Smith selects 1 hammer, 2 screwdrivers, and 3 wrenches.
Step 4: The associated prices for the items are $20.00, $16.00, and $30.00.
Step 5: The subtotals for the items are $20.00, $16.00, and $30.00.
Step 6: A 10% tax for the items is $2.00, $1.60, and $3.00.
Step 7: The total cost of this purchase order is $72.60.

There are additional steps that you could perform. For example, Step 8 would allow John Smith to remove an item, increase/decrease the quantity for each selected item, delete items, or cancel the purchase order. Step 9 would enable John Smith to make a payment. Once again, for the sake of simplicity, we will assume that Step 8 and Step 9 are available in an enhanced version of this Web application.

Note that Step 8 involves updating several of our tables with the details of the purchase order. Step 9 creates a time stamp for the date when the purchase order was created, as well as the status of the purchase order ("paid" versus "pending"). The status of a purchase order is used to generated reports to display the customers whose payment is overdue (and perhaps also send them friendly reminders). Sometimes companies have a reward-based system whereby customers who have paid on time can collect credits that can be applied to other purchases (which is essentially a discount mechanism).

DETAILED TABLE DESCRIPTIONS

If you visualize the use case described in the previous section, you can probably see that we need

- a table for storing customer-specific information
- a table to store purchase orders (which is somehow linked to the associated customer)
- a table that contains the details of the items and quantity that are purchased (which are commonly called "line items")
- a table that contains information about each tool (which includes the name, the description, and the price of the tool).

Hence, the RDBMS for our website requires the following tables:

- `customers`
- `purchase_orders`
- `line_items`
- `item_desc`

The following subsections describe the contents of the preceding tables, along with the relationships among these tables.

The Customers Table

Although there are different ways to specify the attributes of the `customers` table, you need enough information to uniquely identify each customer in the table. By analogy, the following information (except for `cust_id`) is required to send physical mail to a person:

- `cust_id`
- `first_name`
- `last_name`
- `home_address`
- `city`
- `state`
- `zip_code`

We will create the `customers` table with the attributes in the preceding list. Although we'll defer the discussion of keys to a later chapter, it's obvious that we need a mechanism for uniquely identifying every customer. In this table, notice that the `cust_id` attribute uniquely identifies every customer, and therefore it's a *key* for this table. Other examples of keys for database tables include

- social security numbers for people
- student id numbers for students
- course id numbers for classes
- drivers' licenses

Whenever we need to refer to the details of a particular customer, we will use the associated value of `cust_id` to retrieve those details from the row in the `customers` table that has the associated `cust_id`.

The preceding paragraph describes the essence of linking related tables `T1` and `T2` in an `RDBMS`: the key in `T1` is stored as an attribute value in `T2`. If we need to access related information in table `T3`, then we store the key in `T2` as an attribute value in `T3`.

Note that a `customers` table in a *production* system would contain additional attributes, such as the following:

```
title (Mr, Mrs, Ms, and so forth)
shipping_address
cell_phone
```

For the sake of simplicity, we'll use the initial set of attributes to define the `customers` table. Later, you can add the new attributes to the four-table schema to make the system more like a real system.

Suppose that the following information pertains to customer `John Smith`, who has been assigned a `cust_id` of 1000:

```
cust_id: 1000
first_name: John
last_name: Smith
home_address: 1000 Appian Way
city: Sunnyvale
state: California
zip_code:95959
```

Whenever John Smith makes a new purchase, we will use the cust_id value of 1000 to create a new row for this customer in the purchase_orders table. Then whenever we need to find the purchase orders associated with John Smith, we simply look for the rows in the purchase_orders table whose cust_id value equals 1000.

The purchase_orders Table

When existing customers visit the website, they must log into the system, after which they can initiate a new purchase. After they select one or more items, the system creates a purchase order to insert as a new row in the purchase_orders table, and a new row in the line_items table for each item that was selected. While you might be tempted to place all the customers' details in the new row, we will identify the customer by the associated cust_id and use this value instead.

However, we must create a new row in the customers table whenever *new* users register at the website. Repeat customers are identified by an existing cust_id that must be determined by searching the customers table with the information that the customer types into the input fields of the main webpage.

The customers table contains a key attribute; similarly, the purchase_orders table contains an attribute that we call po_id (you are free to use a different string) in order to associate a purchase order for a given customer.

Keep in mind the following detail: a row with a given po_id also contains the cust_id value of the customer (in the customers table) who initiated the current purchase order. Although there are multiple ways to define a set of suitable attributes, let's use the following set of attributes for the purchase_orders table:

```
cust_id
po_id
purchase_date
```

For example, suppose that customer John Smith, whose cust_id is 1000, purchases some tools on December 01, 2021. Although there are dozens of different date formats that are supported in RDBMS, we use the YYYY-MM-DD format (which you can change to suit your particular needs). Then the new row for John Smith in the purchase_orders looks like this, where the po_id value was arbitrarily assigned:

```
cust_id: 1000
po_id: 12500
purchase_date: 2021-12-01
```

As mentioned earlier, a purchase order involves one or more items, each of which is stored in the `line_items` table that is discussed in the next section.

The line_items Table

As a concrete example, suppose that customer `John Smith` requested 1 hammer, 2 screwdrivers, and 3 wrenches in his most recent purchase order. Each of these purchased items requires a row in the `line_items` table that

- is identified by a `line_id` value
- specifies the quantity of each purchased item
- contains the value for the associated `po_id` in the `purchase_orders` table
- contains the value for the associated `item_id` in the `item_desc` table

For simplicity, let's assign the values 5001, 5002, and 5003 to the `line_id` attribute for the three new rows in the `line_items` table that represent the hammer, screwdriver, and wrench items in the current purchase order. A `line_item` row might look like the following code:

```
po_id: 12500
line_id: 5001
item_id: 100 <= we'll discuss this soon
item_count: 1
item_price: 20.00
item_tax:  2.00
item_subtotal: 22.00
```

Notice there is no `cust_id` in the preceding `line_item`: that's because of the top-down approach for retrieving data. Specifically, we start with a particular `cust_id` that we use to find a list of purchase orders in the `purchase_orders` table that belong to the given `cust_id`. For each purchase order in the `purchase_orders` table, we perform a search for the associated line items in the `line_items` table. We can repeat the preceding sequence of steps for each customer in a list of `cust_id` values.

Let us return to the `line_item` details. We need to reference each purchased item by its associated identifier in the `item_desc` table. Once again, we arbitrarily assign `item_id` values of 100, 200, and 300, respectively, for the hammer, screwdriver, and wrench items. The actual values will undoubtedly be different in your application, so there is no special significance to the numbers 100, 200, and 300.

The three rows in the `line_items` table (that belong to the same purchase order) look like this (we'll look at the corresponding SQL statements later):

```
po_id: 12500
line_id: 5001
item_id: 100
item_count: 1
item_price: 20.00
item_tax:  2.00
item_subtotal: 22.00

po_id: 12500
line_id: 5002
item_id: 200
item_count: 2
item_price: 8.00
item_tax:   1.60
item_subtotal: 17.60

po_id: 12500
line_id: 5003
item_id: 300
item_count: 3
item_price: 10.00
item_tax:    3.00
item_subtotal: 33.00
```

The item_desc Table

Recall that the `customers` table contains information about each customer, and a new row is created each time that a new customer creates an account for our Web application. In a somewhat analogous fashion, the `item_desc` table contains information about each item (aka product) that can be purchased from our website. If our website becomes popular, the contents of the `item_desc` table contents are updated more frequently than the `customers` table, typically in the following situations:

- A new tool (aka product) is available for purchase
- An existing tool is no longer available for purchase

Thus, the `item_desc` table contains all the details for every tool that is available for sale, and it's the "source of truth" for the tools that customers can purchase from the website. At a minimum, this table contains three fields (we'll discuss the SQL statement for creating and populating this table later):

```
SELECT *
FROM item_desc;
+---------+-------------+------------+
| item_id | item_desc   | item_price |
+---------+-------------+------------+
|     100 | hammer      |      20.00 |
|     200 | screwdriver |       8.00 |
|     300 | wrench      |      10.00 |
+---------+-------------+------------+
3 rows in set (0.001 sec)
```

There is one more important detail to discuss: if an item is no longer for sale, can we simply drop its row from the item_desc table? The answer is "no" because we need this row to generate reports that contain information about the items that customers purchased.

Hence, it is a good idea to add another attribute called AVAILABLE (or something similar) that contains either 1 or 0 to indicate whether the product is available for purchase. As a result, some of the SQL queries that involve this table will also need to take into account this new attribute. Implementation of this functionality is not central to the purpose of this book, and therefore it is left as an enhancement to the reader.

SQL STATEMENTS FOR THE IMPATIENT (OPTIONAL)

Before delving into the details of a purchase order, there are two implementation detail regarding tax rates. First, do we store the tax-related details for each product in the associated row in the line_items table, or do we calculate those values dynamically when the purchase orders are generated at run time? For simplicity, this book follows the first option.

Second, if the tax rates can change, then you have two choices: update the hard-coded tax rate in the application code, or create another application table (let's call this the TAX_RATE table) that contains the current tax rate, which in this case is 0.10). The advantage of the latter option is that you do not need to alter the application code, and you could also define multiple rows with different tax rates.

The following list describes the sequence of steps each time that a customer (for convenience, let's assume it's John Smith) purchases one or more items from our website:

- Step 1: Customer John Smith (with cust_id 1000) initiated a purchase.
- Step 2: The newly generated purchase_order has the value 12500.
- Step 3: John Smith purchased 1 hammer, 2 screwdrivers, and 3 wrenches.
- Step 4: The cost for the three items is 20.00, 16.00, and 30.00, respectively.
- Step 5: The subtotal for the purchase order is 66.00.
- Step 6: The tax is 6.60 (a tax rate of 10%).
- Step 7: The total cost is 72.60.

As you can see, the tables customers, purchase_orders, and line_items have been updated as follows:

1. customers: a row for new customer John Smith
2. purchase_orders: a new row for customer John Smith
3. line_items: three new rows for the new purchase order
4. item_desc: no changes to this table

We need to create the `customers` table, then the `purchase_orders` table, and then the `line_items` table, as shown in the following code:

```
use mytools;
DROP TABLE IF EXISTS customers;
CREATE TABLE customers (cust_id INTEGER, first_name
VARCHAR(20), last_name VARCHAR(20), home_address
VARCHAR(20), city VARCHAR(20), state VARCHAR(20), zip_code
VARCHAR(10));

INSERT INTO customers
VALUES (1000,'John','Smith','123 Main
St','Fremont','CA','94123');

DROP TABLE IF EXISTS purchase_orders;
CREATE TABLE purchase_orders (cust_id INTEGER, po_id
INTEGER, purchase_date date);

DROP TABLE IF EXISTS line_items;
CREATE TABLE line_items (po_id INTEGER, line_id
INTEGER, item_id INTEGER, item_count INTEGER, item_
price DECIMAL(8,2), item_tax DECIMAL(8,2), item_subtotal
DECIMAL(8,2));
```

Next, creating a new purchase order involves the following steps:

1. Insert a new row in the `purchase_orders` table (new `po_id` and current `cust_id`).
2. For *each* purchased item,
   a. Insert a new row in `line_items` with the `po_id` from Step 1.

Let's use the data values in the previous section to write the pseudocode that performs the preceding steps for `cust_id 1000` who makes a purchase that consists of 1 hammer, 2 screwdrivers, and 3 wrenches:

```
Create four new rows:
  insert row into purchase_orders for a new purchase order
  insert row into line_items for 1 hammer
  insert row into line_items for 2 screwdrivers
  insert row into line_items for 3 wrenches
```

Finally, here are the four SQL statements that create the required new row in the table `purchase_orders` and the three new rows in the table `line_items`:

```
-- create a new purchase order:
INSERT INTO purchase_orders VALUES (1000,12500, '2021-12-01');

-- line item => one hammer:
INSERT INTO line_items VALUES (12500,5001,100,1,20.00,2.00,22.00);
```

```
-- line item => two screwdrivers:
INSERT INTO line_items VALUES (12500,5002,200,2,8.00,1.60,17.60);

-- line item => three wrenches:
INSERT INTO line_items VALUES (12500,5003,300,3,10.00,3.00,33.00);
```

If you want to see the details of the newly created purchase order, here is the SQL statement, keeping in mind that the discussion of SQL statements and the GROUP BY clause is postponed until Chapter 3:

```
SELECT c.cust_id, p.po_id, l.line_id, l.item_subtotal
FROM customers c, purchase_orders p, line_items l
WHERE c.cust_id = p.cust_id
AND   p.po_id = l.po_id;
+---------+-------+---------+---------------+
| cust_id | po_id | line_id | item_subtotal |
+---------+-------+---------+---------------+
|    1000 | 12500 |    5001 |         22.00 |
|    1000 | 12500 |    5002 |         17.60 |
|    1000 | 12500 |    5003 |         33.20 |
+---------+-------+---------+---------------+
3 rows in set (0.003 sec)
```

If you want to see the billable cost of the newly created purchase order, the SQL statement is as follows:

```
SELECT c.cust_id, p.po_id, sum(l.item_subtotal) AS po_total
FROM customers c, purchase_orders p, line_items2 l
WHERE c.cust_id = p.cust_id
AND   p.po_id = l.po_id
GROUP BY c.cust_id, p.po_id;
+---------+-------+----------+
| cust_id | po_id | po_total |
+---------+-------+----------+
|    1000 | 12500 |    72.60 |
+---------+-------+----------+
1 row in set (0.000 sec)
```

A variant of the preceding SQL statement includes the purchase date for the purchase order:

```
SELECT c.cust_id, p.po_id, p.purchase_date, sum(l.item_
subtotal) AS po_total
FROM customers c, purchase_orders p, line_items2 l
WHERE c.cust_id = p.cust_id
AND   p.po_id = l.po_id
GROUP BY c.cust_id, p.po_id, p.purchase_date;
+---------+-------+---------------+----------+
| cust_id | po_id | purchase_date | po_total |
+---------+-------+---------------+----------+
|    1000 | 12500 | 2021-12-01    |    72.60 |
+---------+-------+---------------+----------+
1 row in set (0.001 sec)
```

What About an Item Inventory Table?

An `item_inventory` table is useful for ordering new items when the inventory level drops below a predefined value. Specifically, this table contains a row for each item in the `item_desc` table, where a row consists of an `item_id` attribute and an `on_hand` attribute that specifies how many items are available ("on hand") that can be purchased.

When the inventory level of an item is low (such as 20% of capacity), one technique involves executing a trigger (discussed in Chapter 6) that sends an alert to a system that then generates a purchase order to re-stock the item. Note that the "system" can be an application that automatically generates purchase orders or it could be a person who initiates the necessary purchase order.

For our example, we'll make a simplifying assumption that we will always have enough inventory available for purchase orders. However, if you are looking for enhancement ideas, consider 1) adding an `item_inventory` table to the application that is discussed in this book or 2) adding the appropriate attributes to the `item_desc` table.

THE ROLE OF SQL

SQL is an acronym for Structured Query Language, which is used for managing data in tables in a relational database (RDBMS). SQL is a standard language for managing the contents of structured databases. In high-level terms, a SQL statement to retrieve data generally involves the following:

- what data you want (`SELECT`)
- the table(s) where the data resides (`FROM`)
- constraints (if any) on the data (`WHERE`)

For example, suppose that a `friends` table contains the attributes (database parlance for "fields") `lname` and `fname` for the last name and first name, respectively, of a set of friends, and each row in this table contains details about one friend.

In Chapter 2, we'll learn how to create database tables and how to populate those tables with data, but for now let's just pretend that those tasks have already been performed. Then the SQL statement for retrieving the first and last names of the people in the `friends` table is as follows:

```
SELECT lname, fname
FROM friends;
```

Suppose that the `friends` table also contains a `height` attribute, which is a number (in centimeters) for each person in the `friends` table. We can extend the preceding SQL statement to specify that we want the people (rows) whose `height` attribute is less than 180:

```
SELECT lname, fname
FROM friends
WHERE height < 180;
```

SQL provides numerous keywords that enable you to specify sophisticated queries for retrieving data from multiple tables. Both of the preceding SQL statements are called DML statements, which is one of the four categories of SQL statements:

- DCL (Data Control Language
- DDL (Data Definition Language)
- DQL (Data Query Language)
- DML (Data Manipulation Language)

The following subsections provide additional information for each item in the preceding list.

DCL, DDL, DQL, DML, and TCL

DCL is an acronym for Data Control Language, which refers to any SQL statement that contains the keywords `GRANT` or `REVOKE`. Both of the keywords affect the permissions that are either granted or revoked for a particular user.

DDL is an acronym for Data Definition Language, which refers to any SQL statements that specify the following: `CREATE`, `ALTER`, `DROP`, `RENAME`, `TRUNCATE`, or `COMMENT`. These SQL keywords are used in conjunction with database tables and in many cases with database views (discussed later).

DQL is an acronym for Data Query Language, which refers to any SQL statement that contains the keyword `SELECT`.

DML is an acronym for Data Manipulation Language, which refers to SQL statements that execute queries against one or more tables in a database. The SQL statements contain any of the keywords `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `CALL`, `EXPLAIN PLAN`, or `LOCK TABLE`. In most cases, these keywords modify the existing values of data in one or more tables.

TCL is an acronym for Transaction Control Language, which refers to any of the keywords `COMMIT`, `ROLLBACK`, `SAVEPOINT`, or `SET TRANSACTION`.

SQL Privileges

There are two types of privileges available in SQL, both of which are described briefly in this section. These privileges refer to database objects such as database tables and indexes that are discussed in greater detail in subsequent chapters.

System privileges involve an object of a particular type and specifies the right to perform one or more actions on the object. Such actions include the administrator giving users permission to perform tasks such as `ALTER ANY INDEX, ALTER ANY CACHE GROUP, CREATE/ALTER/DELETE TABLE,` or `CREATE/ALTER/DELETE VIEW.`

Object privileges allow users to perform actions on an object or object of *another* user, such as tables, views, and indexes. Additional object privileges are EXECUTE, INSERT, UPDATE, DELETE, SELECT, FLUSH, LOAD, INDEX, and REFERENCES.

PROPERTIES OF SQL STATEMENTS

SQL statements and SQL functions (discussed in Chapter 4) are not case sensitive, but *quoted* text *is* case sensitive. Here are some examples of SQL statements that are executed from the MySQL prompt:

```
MySQL [mytools]> select VERSION();
+-----------+
| VERSION() |
+-----------+
| 8.0.21    |
+-----------+
1 row in set (0.000 sec)

MySQL [mytools]> SeLeCt Version();
+-----------+
| Version() |
+-----------+
| 8.0.21    |
+-----------+
1 row in set (0.000 sec)
```

Keep in mind the following useful details regarding SQL statements:

- SQL statements are not case sensitive.
- SQL statements can be on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indentation is for enhancing readability.

The CREATE Keyword

In general, you will use the CREATE keyword to create a database and more often to create tables, views, and indexes. However, the following list contains all the objects that you can create via the CREATE statement:

- DATABASE
- EVENT
- FUNCTION
- INDEX
- PROCEDURE
- TABLE
- TRIGGER
- USER
- VIEW

With the exception of EVENT, all the keywords in the preceding list are discussed, along with SQL statements, in various chapters of this book.

DATA TYPES IN MYSQL

This section starts with a lengthy list of data types that MySQL supports, followed by some comments about several of the data types, all of which you can use in table definitions:

- The BIT datatype is for storing bit values in MySQL.
- The BOOLEAN datatype stores True/False values.
- The CHAR data type is for storing fixed length strings.
- The DATE datatype is for storing date values.
- The DATETIME datatype is for storing combined date and time values.
- The DECIMAL datatype is for storing exact values in decimal format.
- The ENUM datatype is a compact way to store string values.
- The INT datatype is for storing an integer data type.
- The JSON data type is for storing JSON documents.
- The TEXT datatype is for storing text values.
- The TIME datatype is for storing time values.
- The TIMESTAMP datatype is for a wider range of date and time values.
- The TO_SECONDS datatype is for converting time to seconds.
- The VARCHAR datatype is for variable length strings.
- The XML data type provides support for XML documents.

The CHAR and VARCHAR Data Types

The CHAR type has a fixed column length whose value is declared while creating tables, which can range from 1 to 255. CHAR values are right padded with spaces to the specified length, and trailing spaces are removed when CHAR values are retrieved.

By contrast, the VARCHAR type indicates variable length CHAR values whose length can be between 1 and 2000, and it occupies the space for NULL values.

The VARCHAR2 type indicates variable length CHAR values whose length can be between 1 and 4000, but cannot occupy the space for NULL values. Hence, VARCHAR2 has better performance than VARCHAR.

String-Based Data Types

The previous bullet list contains various string types, and the latter have been extracted and placed in a separate list below for your convenience:

- BLOB
- CHAR
- ENUM

- SET
- TEXT
- VARCHAR

The ENUM datatype is string object that specifies a set of predefined values, which can be used during table creation:

```
CREATE TABLE PIZZA(name ENUM('Small', 'Medium','Large'));
Query OK, 0 rows affected (0.021 sec)

DESC pizza;
+-------+-----------------+------+-----+---------+-------+
| Field | Type            | Null | Key | Default | Extra |
+-------+-----------------+------+-----+---------+-------+
| name  | enum('Small',   | YES  |     | NULL    |       |
|       |'Medium','Large')|      |     |         |       |
+-------+-----------------+------+-----+---------+-------+
1 row in set (0.004 sec)
```

FLOAT and DOUBLE Data Types

Numbers in the FLOAT format are stored in four bytes and have eight decimal places of accuracy. Numbers in the DOUBLE format are stored in eight bytes and have eighteen decimal places of accuracy.

BLOB and TEXT Data Types

A BLOB is an acronym for binary large object that can hold a variable amount of data. There are four BLOB types whose only difference is their maximum length:

- TINYBLOB
- BLOB
- MEDIUMBLOB
- LONGBLOB

A TEXT data type is a case-insensitive BLOB, and there are four TEXT types whose difference pertains to their maximum length (all of which are non-standard data types):

- TINYTEXT
- TEXT
- MEDIUMTEXT
- LONGTEXT

Keep in mind the following difference between BLOB types and TEXT types: BLOB types involve case-sensitive sorting and comparisons, whereas these operations are case-insensitive for TEXT types.

MYSQL DATABASE OPERATIONS

There are several operations that you can perform with a MySQL database:

- Create a database
- Import/Export a database
- Drop a database
- Rename a database

You will see examples of how to perform each of the preceding bullet items in the following subsections.

Creating a Database

Log into MySQL and execute the following SQL statement to create the `mytools` database:

```
MySQL [mysql]> create database mytools;
Query OK, 1 row affected (0.004 sec)
```

Now select the `mytools` database with the following command:

```
MySQL [(none)]> use mytools;
Reading table information for completion of table and
column names
You can turn off this feature to get a quicker startup
with -A
Database changed
```

Display a List of Databases

Display the existing databases by invoking the following SQL statement:

```
mysql> SHOW DATABASES;
```

The preceding command displays the following output (which might be different for your machine):

```
+--------------------+
| Database           |
+--------------------+
| beans              |
| information_schema |
| minimal            |
| mysql              |
| mytools            |
| performance_schema |
| sys                |
+--------------------+
9 rows in set (0.002 sec)
```

Display a List of Database Users

Display the list of existing users by invoking the following SQL statement:

```
mysql> select user from mysql.user;
```

The preceding SQL statement displays the following output:

```
+------------------+
| user             |
+------------------+
| mysql.infoschema |
| mysql.session    |
| mysql.sys        |
| root             |
+------------------+
4 rows in set (0.001 sec)
```

Dropping a Database

Log into MySQL and invoke the following SQL statement to create, select, and then drop the `pizza` database:

```
MySQL [(none)]> create database pizza;
Query OK, 1 row affected (0.004 sec)
MySQL [(none)]> use pizza;
Database changed
MySQL [pizza]> drop database pizza;
Query OK, 0 rows affected (0.007 sec)
```

Performing this task with a database that does not contain any data is straightforward, without the loss of any data.

EXPORTING A DATABASE

Although you currently have an empty database, it's still good to know how the steps for exporting a database, which is handy as a backup and also provides a simple way to create a copy of an existing database on a different machine.

By way of illustration, let's first create the database called `minimal` in MySQL, as shown here:

```
MySQL [mytools]> create database minimal;
Query OK, 1 row affected (0.006 sec)
```

Next, invoke the `mysqldump` command from the command line to export the minimal database, as shown here:

```
mysqldump -u username -p"password" -R minimal > minimal.sql
```

Notice the details of the preceding command. First, there are no intervening spaces between the -p flag and the password in order to bypass a command line prompt to enter the password. Second, make sure that you omit the quote marks. Third, the -R flag instructs mysqldump to copy stored procedures and functions in addition to the database data.

As a specific example, if the user is root and the password is mypassword, then the preceding command is as follows:

```
mysqldump -u root -pmypassword -R minimal > minimal.sql
```

At this point, you can create tables in the minimal database and periodically export its contents. Listing 1.1 shows the content of minimal.sql, which is the complete description of the minimal database.

LISTING 1.1: minimal.sql

```
-- MariaDB dump 10.18  Distrib 10.5.8-MariaDB, for osx10.15 (x86_64)
--
-- Host: localhost    Database: minimal
-- ------------------------------------------------------
-- Server version  8.0.21

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_
CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Dumping routines for database 'minimal'
--
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2021-12-23 22:44:54
```

RENAMING A DATABASE

Since the database is empty, it's convenient to see how to rename a database (and besides, it's faster to do so with an empty database).

Older versions of MySQL provided the `RENAME DATABASE` command to rename a database; however, newer versions of MySQL have removed this functionality to avoid security risks.

Perform the following three-step process using MySQL command line utilities to rename a MySQL database `OLD_DB` (which you need to replace with the name of the database that you want to rename) to a new database, `NEW_DB` (replaced with the actual new database name):

Step 1) Create an exported copy of database `OLD_DB`.
Step 2) Create a new database called `NEW_DB`.
Step 3) Import data from `OLD_DB` into `NEW_DB`.

Perform Step 1) by invoking the following command (see the previous section):

```
mysqldump -u username -p"password" -R OLD_DB > OLD_DB.sql
```

Perform Step 2) by invoking the following command:

```
mysqladmin -u username -p"password" create NEW_DB
```

Perform Step 3) by invoking the following command:

```
mysql -u username -p"password" newDbName < OLD_DB.sql
```

Verify that everything worked correctly by logging into MySQL and selecting the new database:

```
MySQL [mysql]> use NEW_DB;
Database changed
```

SHOW DATABASE TABLES

Log into MySQL and select the `mytools` database as described in the preceding section, and then display the tables in the `mytools` database with the following command:

```
use mytools;
Database changed

show tables;
+--------------------+
| Tables_in_mytools  |
+--------------------+
| account            |
| courses            |
| curr_exchange_rate |
| currencies         |
| cust_history       |
| customers          |
| employees          |
```

```
| FRIENDS             |
| FRIENDS2            |
| item_desc           |
| japn1               |
| japn2               |
| japn3               |
| japn_emps           |
| json1               |
| line_items          |
| new_items           |
| people              |
| people2             |
| purchase_orders     |
| sample              |
| schedule            |
| students            |
| temp_cust2          |
| user                |
| user2               |
| user3               |
| weather             |
| weather2            |
+---------------------+
29 rows in set (0.001 sec)
```

The preceding output displays the tables in the `mytools` database that you will encounter in various chapters of this book. Note that Chapter 6 contains the SQL file `mytools.sql` that you can execute to generate the `mytools` database that creates and populates the tables in `mytools` with the data that is stored in `mytools.sql`.

Now let's switch to the `mysql` database and show the list of tables in that database:

```
SQL [mytools]> use mysql;
Reading table information for completion of table and
column names
You can turn off this feature to get a quicker startup with -A

Database changed

MySQL [mysql]> show tables;
+----------------------------+
| Tables_in_mysql            |
+----------------------------+
| columns_priv               |
| component                  |
| db                         |
| default_roles              |
| engine_cost                |
| func                       |
| general_log                |
| global_grants              |
```

```
| gtid_executed              |
| help_category              |
| help_keyword               |
| help_relation              |
| help_topic                 |
| innodb_index_stats         |
| innodb_table_stats         |
| password_history           |
| plugin                     |
| procs_priv                 |
| proxies_priv               |
| role_edges                 |
| server_cost                |
| servers                    |
| slave_master_info          |
| slave_relay_log_info       |
| slave_worker_info          |
| slow_log                   |
| tables_priv                |
| time_zone                  |
| time_zone_leap_second      |
| time_zone_name             |
| time_zone_transition       |
| time_zone_transition_type |
| user                       |
+----------------------------+
33 rows in set (0.004 sec)
```

Although we won't explore this database, you can read the online documentation for more information about the tables in this database.

THE INFORMATION_SCHEMA TABLE

The `INFORMATION_SCHEMA.COLUMNS` table enables you to retrieve information about the columns in a given table. Execute the following SQL statement:

```
desc INFORMATION_SCHEMA.COLUMNS;
```

Some of the columns in the preceding table are as follows:

```
TABLE_SCHEMA
TABLE_NAME
COLUMN_NAME
ORDINAL_POSITION
COLUMN_DEFAULT
IS_NULLABLE
DATA_TYPE
CHARACTER_MAXIMUM_LENGTH
NUMERIC_PRECISION
NUMERIC_SCALE
DATETIME_PRECISION
```

We can query the preceding table to obtain more information about the structure of the `weather` table that is created later in this book:

```
MySQL [mytools]> desc weather;
+----------+----------+------+-----+---------+-------+
| Field    | Type     | Null | Key | Default | Extra |
+----------+----------+------+-----+---------+-------+
| day      | date     | YES  |     | NULL    |       |
| temper   | int      | YES  |     | NULL    |       |
| wind     | int      | YES  |     | NULL    |       |
| forecast | char(20) | YES  |     | NULL    |       |
| city     | char(20) | YES  |     | NULL    |       |
| state    | char(20) | YES  |     | NULL    |       |
+----------+----------+------+-----+---------+-------+
6 rows in set (0.001 sec)
```

Now invoke the following SQL statement:

```
SELECT COLUMN_NAME, DATA_TYPE, IS_NULLABLE, COLUMN_DEFAULT
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'weather'
AND table_schema = 'mytools';
```

The preceding SQL query generates the following output:

```
+-------------+-----------+-------------+----------------+
| COLUMN_NAME | DATA_TYPE | IS_NULLABLE | COLUMN_DEFAULT |
+-------------+-----------+-------------+----------------+
| city        | char      | YES         | NULL           |
| day         | date      | YES         | NULL           |
| forecast    | char      | YES         | NULL           |
| state       | char      | YES         | NULL           |
| temper      | int       | YES         | NULL           |
| wind        | int       | YES         | NULL           |
+-------------+-----------+-------------+----------------+
6 rows in set (0.001 sec)
```

THE PROCESSLIST TABLE

The PROCESSLIST table contains information about the status of SQL statements. This information is useful when you want to see the status of table-level or row-level locks on a table (which is outside the scope of this book). The following SQL statement shows you an example of the contents of this table.

```
MySQL [mytools]> show processlist;
+----+-----------------+-----------+---------+---------+--------+------
---------+------------------+
| Id | User            | Host      | db      | Command | Time   | State
| Info             |
+----+-------------+-----------+--------+---------+--------+-----------
--------+------------------+
| 5 | event_scheduler | localhost | NULL | Daemon  | 138765 | Waiting on
empty queue | NULL             |
| 9 | root           | localhost |mytools | Query   |      0 | starting
| show processlist |
+----+-----------------+-----------+---------+---------+--------+------
---------+------------------+
2 rows in set (0.000 sec)
```

SQL FORMATTING TOOLS

As you might expect, there are various formatting styles for SQL statements, and you can peruse them to determine which style is most appealing to you. The following site has an online SQL formatter:

https://codebeautify.org/sqlformatter

The following site contains 18 SQL formatters, some of which are commercial and some are free:

https://www.sqlshack.com/sql-formatter-tools/

The following site contains a list of SQL formatting conventions (i.e., it's not about formatting tools):

https://opendatascience.com/best-practices-sql-formatting

If you work in an environment where the SQL formatting rules have already been established, it might be interesting to compare that style with those of the SQL formatting tools in the preceding links.

If you are a SQL beginner working on your own, it's also worth exploring these sites as you learn more about SQL statements throughout this book. As you gain more knowledge about writing SQL statements, you will encounter various styles in blog posts and the conventions that they follow for formatting SQL statements.

SUMMARY

This chapter started with an introduction to the concept of an RDBMS, and the rationale for using an RDBMS. In particular, you saw an example of an RDBMS with a single table, two tables, and four tables (and there are much larger RDBMSs).

Then you got a brief introduction to the notion of database normalization, and how doing so will help you maintain data integrity ("single source of truth") in an RDBMS.

Next, you learned about the structure of the tables in a four-table database that keeps track of customer purchases of tools through a webpage. You also saw which tables have a one-to-many relationship so that you can find all the line items that belong to a given purchase order.

In addition, you obtained a brief introduction to SQL and some basic examples of SQL queries (more details are in Chapter 2). You also learned about various types of SQL statements that can be classified as DCL (Data Control Language), DDL (Data Definition Language), DQL (Data Query Language), or DML (Data Manipulation Language).

Next, you learned about SQL data types, and then you learned how to perform database operations, such as creating, dropping, and renaming a database in MySQL. Finally, you learned about two useful built-in tables that enable you to find the columns of a given table and the status of SQL statements.

CHAPTER 2

WORKING WITH SQL AND MYSQL

The previous chapter provided a fast-paced introduction to RDBMSs and SQL concepts, whereas this chapter contains more details about MySQL and illustrates various SQL statements that are necessary to create and manage the database tables for a fictitious website.

The first part of this chapter presents various ways to create MySQL tables, which can be performed manually, from SQL scripts, or from the command line. You will also see how to create a MySQL table that contains Japanese text that contains a mixture of Kanji and Hiragana. This section also shows you how to drop and alter MySQL tables, and how to populate MySQL tables with seed data.

The second part of this chapter contains an assortment of SQL statements that involve the `SELECT` keyword. You will see SQL statements that find the distinct rows in a MySQL table as well as the unique rows, along with using the `EXISTS` and `LIMIT` keywords. This section also explains the differences among the `DELETE`, `TRUNCATE`, and `DROP` keywords in SQL.

The third part of this chapter introduces the concept of an index, and then shows you how to create indexes on MySQL tables, along with criteria for defining indexes, followed by how to select columns for an index. Although the four tables in Chapter 1 are very small enough and do not require any indexes, it's important to understand the purpose of indexes and how to create them.

Depending on the configuration of MySQL on your system, you might encounter issues when you attempt to export data in a database table to a text file or when you attempt to import CSV data into a database table. A simpler alternative is to download and install MySQL Workbench (discussed in Chapter 6) to export MySQL data or to import data into MySQL tables.

DROP DATABASE TABLES

You might be wondering why we're discussing how to drop a database table when we haven't learned how to create a table. SQL scripts, as well as command line invocations of SQL statements, will often drop a table and then recreate the table for the following reasons:

1. The table definition needs to be modified.
2. The table data needs to be modified.
3. Both 1) and 2).

Listing 2.1 shows the content of `mytools_drop_tables.sql` that illustrates the syntax to drop multiple database tables (but without recreating them).

LISTING 2.1: mytools_drop_tables.sql

```
USE DATABASE mytools;
-- drop tables if they already exist:
DROP TABLE IF EXISTS customers;
DROP TABLE IF EXISTS purchase_orders;
DROP TABLE IF EXISTS line_items;
DROP TABLE IF EXISTS item_desc;
```

Listing 2.1 contains four SQL statements to drop four tables if they already exist. If they do not exist, then no error occurs. Now let's see how to create database tables, as discussed in the next section.

CREATE DATABASE TABLES

There are three ways to create database tables in MySQL as well as other RDBMSs. One technique is manual (shown first); another technique (shown second) invokes a SQL file that contains suitable SQL commands; and a third technique involves redirecting a SQL file to the MySQL executable from the command line.

The next section shows you how to create the four tables (described in Chapter 1) for the Web application.

Manually Creating Tables for mytools.com

This section shows you how to *manually* create the four tables for the `mytools` database based on the attributes (column names) that were discussed in Chapter 1. Specifically, you will see how to create the following four tables:

- `customers`
- `purchase_orders`

- line_items
- item_desc

Now log into MySQL, and after selecting the mytools database, type the following commands to create the required tables:

```
MySQL [mytools]> CREATE TABLE customers (cust_id INTEGER,
first_name VARCHAR(20), last_name VARCHAR(20), home_address
VARCHAR(20), city VARCHAR(20), state VARCHAR(20), zip_code
VARCHAR(10));

MySQL [mytools]> CREATE TABLE purchase_orders ( cust_id
INTEGER, po_id INTEGER, purchase_date date);

MySQL [mytools]> CREATE TABLE line_items (po_id INTEGER,
line_id INTEGER, item_id INTEGER, item_count INTEGER, item_
price DECIMAL(8,2), item_tax DECIMAL(8,2), item_subtotal
DECIMAL(8,2));

MySQL [mytools]> CREATE TABLE item_desc (item_id INTEGER,
item_desc VARCHAR(80), item_price DECIMAL(8,2));
```

Describe the structure of the customers table with the following command:

```
MySQL [mytools]> desc customers;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| cust_id      | int         | YES  |     | NULL    |       |
| first_name   | varchar(20) | YES  |     | NULL    |       |
| last_name    | varchar(20) | YES  |     | NULL    |       |
| home_address | varchar(20) | YES  |     | NULL    |       |
| city         | varchar(20) | YES  |     | NULL    |       |
| state        | varchar(20) | YES  |     | NULL    |       |
| zip_code     | varchar(10) | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
7 rows in set (0.003 sec)
```

Describe the structure of the purchase_orders table with the following command:

```
MySQL [mytools]> desc purchase_orders;
+---------------+------+------+-----+---------+-------+
| Field         | Type | Null | Key | Default | Extra |
+---------------+------+------+-----+---------+-------+
| cust_id       | int  | YES  |     | NULL    |       |
| po_id         | int  | YES  |     | NULL    |       |
| purchase_date | date | YES  |     | NULL    |       |
+---------------+------+------+-----+---------+-------+
3 rows in set (0.004 sec)
```

Describe the structure of the `line_items` table with the following command:

```
MySQL [mytools]> desc line_items;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Yes  | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| po_id         | int          | YES  |     | NULL    |       |
| line_id       | int          | YES  |     | NULL    |       |
| item_id       | int          | YES  |     | NULL    |       |
| item_count    | int          | YES  |     | NULL    |       |
| item_price    | decimal(8,2) | YES  |     | NULL    |       |
| item_tax      | decimal(8,2) | YES  |     | NULL    |       |
| item_subtotal | decimal(8,2) | YES  |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
7 rows in set (0.002 sec)
```

Describe the structure of the `item_desc` table with the following command:

```
MySQL [mytools]> desc item_desc;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| item_id    | int          | YES  |     | NULL    |       |
| item_desc  | varchar(80)  | YES  |     | NULL    |       |
| item_price | decimal(8,2) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
3 rows in set (0.006 sec)
```

Creating Tables via a SQL Script for mytools.com

The previous section shows you a manual technique for creating database tables. By contrast, Listing 2.2 shows the content of `mytools_create_tables.sql` that illustrates how to define multiple SQL statements for creating database tables. Note that the SQL statements are identical to the SQL statements in the previous section.

LISTING 2.2: mytools_create_tables.sql

```
USE DATABASE mytools;

-- drop tables if they already exist:
DROP TABLE IF EXISTS customers;
DROP TABLE IF EXISTS purchase_orders;
DROP TABLE IF EXISTS line_items;
DROP TABLE IF EXISTS item_desc;

—- these SQL statements are the same as the previous
section:
CREATE TABLE customers (cust_id INTEGER, first_name
VARCHAR(20), last_name VARCHAR(20), home_address
```

```
VARCHAR(20), city VARCHAR(20), state VARCHAR(20), zip_code
VARCHAR(10));

CREATE TABLE purchase_orders ( cust_id INTEGER, po_id
INTEGER, purchase_date date);

CREATE TABLE line_items (po_id INTEGER, line_id
INTEGER, item_id INTEGER, item_count INTEGER, item_
price DECIMAL(8,2), item_tax DECIMAL(8,2), item_subtotal
DECIMAL(8,2));

CREATE TABLE item_desc (item_id INTEGER, item_desc
VARCHAR(80), item_price DECIMAL(8,2));
```

Listing 2.2 contains three sections. The first section selects the `mytools` database, and the second section drops any of the four required tables if they already exist. The third section contains the SQL commands to create the four required tables.

Creating Tables with Japanese Text

Although this section is not required for any of the code samples in this book, it's nonetheless interesting to see how easily you can create a MySQL table with Japanese text. The Japanese text was inserted from a MacBook after adding a Hiragana keyboard and a Katana keyboard. Perform an online search for instructions that explain how to add these keyboards to your laptop.

Listing 2.3 shows the content of `japanese1.sql` that illustrates how to create a MySQL table that is populated with Japanese text.

LISTING 2.3: japanese1.sql

```
use mytools;
DROP TABLE IF EXISTS japn1;

CREATE TABLE japn1
(
  emp_id INT NOT NULL AUTO_INCREMENT,
  fname VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  lname VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  title VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL,
  PRIMARY KEY (emp_id)
);

INSERT INTO japn1 SET fname="ひでき", lname="日浦",title="しちお";
INSERT INTO japn1 SET fname="ももたろ", lname="つよい",title="かちょ";
INSERT INTO japn1 SET fname="オズワルド", lname="カmポ",title="悪ガキ";
INSERT INTO japn1 SET fname="東京", lname="日本",title="すごい！";

\! echo '=> All rows in table japn1:';
SELECT * FROM japn1;

\! echo '=> Rows whose lname contains カ:';
SELECT * FROM japn1
WHERE lname LIKE '%カ%';
```

Listing 2.3 starts with the definition of the table `japn1` that defines the `fname`, `lname`, and `title` attributes as `VARCHAR(100)` and also specifies `utf8` as the character set and `utf8_general_ci` as the collating sequence. These extra keywords enable us to store Hiragana and Kanji characters in these three attributes. Launch the code in Listing 2.3 from the MySQL prompt to see the following output:

```
Database changed
Query OK, 0 rows affected (0.005 sec)
Query OK, 0 rows affected, 6 warnings (0.005 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)

=> All rows in table japn1:
+--------+-----------------+-----------+--------------+
| emp_id | fname           | lname     | title        |
+--------+-----------------+-----------+--------------+
|      1 | ひでき          | 日浦      | しちお       |
|      2 | ももたろ        | つよい    | かちょ       |
|      3 | オズワルド      | カmポ     | 悪ガキ       |
|      4 | 東京            | 日本      | すごい！     |
+--------+-----------------+-----------+--------------+
4 rows in set (0.000 sec)

=> Rows whose lname matches カ:
+--------+-----------------+---------+-----------+
| emp_id | fname           | lname   | title     |
+--------+-----------------+---------+-----------+
|      3 | オズワルド      | カmポ   | 悪ガキ    |
+--------+-----------------+---------+-----------+
1 row in set (0.000 sec)
```

The preceding example is a rudimentary example of working with Japanese text in a MySQL table. Chapter 4 shows you how to perform a join on the table `japn1` with the table `japn2`, where the text in `japn2` contains the English counterpart to the text in `japn1`. You can also search online for other SQL-based operations that you can perform with this data, as well as examples of creating MySQL tables for other languages.

Creating Tables from the Command Line

The third technique for invoking a SQL file is from the command line. First make sure that the specified database already exists (such as `mytools`). Next, invoke the following command from the command line to execute the SQL statements in `employees.sql` in MySQL:

```
mysql --password=<your-password> --user=root mytools <
user.sql
```

Listing 2.4 shows the content of `user.sql` that illustrates how to create a database table and populate that table with data.

LISTING 2.4: user.sql

```
USE mytools;

DROP TABLE IF EXISTS user;
CREATE TABLE user (user_id INTEGER(8), user_title VARCHAR(20));

INSERT INTO user VALUES (1000, 'Developer');
INSERT INTO user VALUES (2000, 'Project Lead');
INSERT INTO user VALUES (3000, 'Dev Manager');
INSERT INTO user VALUES (4000, 'Senior Dev Manager');
```

Log into MySQL with the following command from the command line:

```
mysql --password=<your-password> —user=root
```

Enter the following two commands (shown in bold):

```
MySQL [(none)]> use mytools;
Reading table information for completion of table and
column names
You can turn off this feature to get a quicker startup with
-A

Database changed
MySQL [mytools]> desc user;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| user_id    | int         | YES  |     | NULL    |       |
| user_class | int         | YES  |     | NULL    |       |
| user_title | varchar(20) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
3 rows in set (0.002 sec)
```

The section in this chapter that discusses the concept of *keys* contains an example of creating a table that contains a primary key of type `AUTOINCREMENT`, which is incremented each time that a row is inserted into a given table.

Defining Table Attributes

You have already seen examples of table columns that use `CHAR` as well as `VARCHAR` in their definition. In some cases, you might see a performance improvement if you adopt the following recommendations:

- Use CHAR instead of VARCHAR for fixed-length fields.
- Use TEXT for large blocks of text such as blog posts.
- Use INT for larger numbers up to 2^32 or 4 billion.
- Use DECIMAL for currency to avoid floating point representation errors.
- Avoid storing large BLOBS, store the location of where to get the object instead.
- Set the NOT NULL constraint where applicable to improve search performance.

CHAR is recommended because it enables fast random access, whereas VARCHAR necessitates finding the end of the current string before processing the next string.

Another point to remember is that a TEXT attribute supports Boolean searches: using a TEXT field involves storing a pointer on disk that is used to locate the text block.

WORKING WITH ALIASES IN SQL

An *alias* can be used for 1) an existing table, 2) dynamically creating a table based on an existing table, 3) creating a view, or 4) assigning a temporary name to an attribute of a table in a SQL statement.

You can create an alias with the AS keyword, whose scope is limited to the SQL statement in which it appears. The AS keyword is used in multiple ways. For example, the following SQL statement uses the AS keyword as an alias for an existing table:

```
SELECT emp_id, mgr_id
FROM employees AS emps;
+--------+--------+
| emp_id | mgr_id |
+--------+--------+
|   1000 |   2000 |
|   2000 |   3000 |
|   3000 |   4000 |
|   4000 |   4000 |
+--------+--------+
4 rows in set (0.002 sec)
```

Use the AS keyword to create a new table based on an existing table, as shown here:

```
CREATE TABLE user2 AS (SELECT * FROM user);
```

Use the AS keyword to create a new table based on a *subset* of the attributes of an existing table, as shown here:

```
CREATE TABLE user3 AS (SELECT user_title FROM user);
```

Use the AS keyword to create a view based on an existing table, as outlined here:

```
CREATE VIEW V3 AS (SELECT ...);
```

Use the `AS` keyword to specify aliases for table attributes, as shown here:

```
SELECT emp_id AS e, mgr_id AS m, title AS t
FROM employees;
+------+------+--------------------+
| e    | m    | t                  |
+------+------+--------------------+
| 1000 | 2000 | Developer          |
| 2000 | 3000 | Project Lead       |
| 3000 | 4000 | Dev Manager        |
| 4000 | 4000 | Senior Dev Manager |
+------+------+--------------------+
4 rows in set (0.006 sec)
```

Later you will see the SQL statement to create and populate the `employees` table with data.

ALTER DATABASE TABLES WITH THE ALTER KEYWORD

If you want to modify the columns in a table, you can use the `ALTER` command to add new columns, drop existing columns, or modify the data type of an existing column. Whenever a new column is added to a database table, that column will contain `NULL` values. However, you can invoke SQL statements to populate the new column with values, as shown in the next section.

Add a Column to a Database Table

As a simple example, let's create the table `user2` from table user, as shown here:

```
CREATE TABLE user2 AS (SELECT * FROM user);
```

Let's add the character columns `fname` and `lname` to table `user2` by executing the following SQL commands:

```
MySQL [mytools]>
ALTER TABLE user2
ADD COLUMN fname VARCHAR(20);
Query OK, 0 rows affected (0.011 sec)
Records: 0  Duplicates: 0  Warnings: 0

MySQL [mytools]>
ALTER TABLE user2
ADD COLUMN lname VARCHAR(20);
Query OK, 0 rows affected (0.012 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Let's look at the structure of table `user2`, which contains two new columns with `NULL` values:

```
MySQL [mytools]> desc user2;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| user_id    | int         | YES  |     | NULL    |       |
| user_title | varchar(20) | YES  |     | NULL    |       |
| fname      | varchar(20) | YES  |     | NULL    |       |
| lname      | varchar(20) | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
4 rows in set (0.002 sec)
```

Let's look at the rows in table `user2` by issuing the following SQL query:

```
select * from user2;
+---------+---------------------+-------+-------+
| user_id | user_title          | fname | lname |
+---------+---------------------+-------+-------+
|    1000 | Developer           | NULL  | NULL  |
|    2000 | Project Lead        | NULL  | NULL  |
|    3000 | Dev Manager         | NULL  | NULL  |
|    4000 | Senior Dev Manager  | NULL  | NULL  |
+---------+---------------------+-------+-------+
4 rows in set (0.001 sec)
```

How do we insert the appropriate values for the new `fname` and `lname` attributes for each existing row? One way to update these attributes is to issue a SQL query for each row that updates these attributes based on the `user_id`:

```
UPDATE user2
SET fname = 'John', lname = 'Smith'
WHERE user_id = 1000;

UPDATE user2
SET fname = 'Jane', lname = 'Stone'
WHERE user_id = 2000;

UPDATE user2
SET fname = 'Dave', lname = 'Dodds'
WHERE user_id = 3000;

UPDATE user2
SET fname = 'Jack', lname = 'Jones'
WHERE user_id = 4000;
```

We can confirm that the `user2` table has been updated correctly with the following SQL query:

```
select * from user2;
+---------+--------------------+-------+-------+
| user_id | user_title         | fname | lname |
+---------+--------------------+-------+-------+
|    1000 | Developer          | John  | Smith |
|    2000 | Project Lead       | Jane  | Stone |
|    3000 | Dev Manager        | Dave  | Dodds |
|    4000 | Senior Dev Manager | Jack  | Jones |
+---------+--------------------+-------+-------+
4 rows in set (0.000 sec)
```

Unfortunately, the preceding solution is not scalable if you need to update hundreds or thousands of rows with values for the new attributes. There are several options available, depending on the location of the values for the new attributes: one option involves importing data and another involves programmatically generating SQL statements.

If you have a CSV file that contains the complete data for the table rows, including values for the `fname` and `lname` attributes, the solution is straightforward: delete the rows from the `user2` table and then import the data from the CSV file into the `user2` table.

However, if the existing data is located in one CSV file and the data for the two new attributes is located in a separate CSV file, you need to merge the two CSV files into a single CSV file, after which you can import the CSV file directly into the `user2` table. An example of performing this task is discussed after the following section that drops a column and changes column types.

Drop a Column from a Database Table

The following SQL statement illustrates how to drop the column `str_date` from the table `mytable`:

```
ALTER TABLE mytable
DROP COLUMN str_date;
```

Chapter 6 contains a complete example of dropping a column from a MySQL table.

Change the Data Type of a Column

Listing 2.5 shows the content of `people_ages.sql` that illustrates how to change the data type of a column in a MySQL table.

LISTING 2.5: people_ages.sql

```
USE mytools;
DROP TABLE IF EXISTS people_ages;
CREATE TABLE people_ages (float_ages DECIMAL(4,2), floor_ages INT);

INSERT INTO people_ages VALUES (12.3,0);
INSERT INTO people_ages VALUES (45.6,0);
INSERT INTO people_ages VALUES (78.9,0);
INSERT INTO people_ages VALUES (-3.4,0);
DESC people_ages;
SELECT * FROM people_ages;

-- populate floor_ages with FLOOR (=INT) value:
UPDATE people_ages
SET floor_ages = FLOOR(float_ages);
SELECT * FROM people_ages;

-- change float_ages to INT data type:
ALTER TABLE people_ages CHANGE float_ages int_ages INT;
DESC people_ages;
SELECT * FROM people_ages;

-- rows whose minimum age is less than min_value:
SELECT @min_value := 2;
SELECT * FROM people_ages WHERE floor_ages < @min_value;
```

Listing 2.5 creates and populates the people_ages table with data. The other code in Listing 2.5 contains three SQL statements, each of which starts with a comment statement that explains its purpose.

The first SQL statement populates the integer-valued column floor_ages with the floor of the float_ages column via the built-in FLOOR() function.

The second SQL statement alters the decimal-valued column float_ages to a column of type INT.

The third SQL statement displays the rows in the people_ages table whose floor_ages value is less than min_value.

Launch the code in Listing 2.5 to see the following output:

```
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| float_ages | decimal(4,2) | YES  |     | NULL    |       |
| floor_ages | int          | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
2 rows in set (0.001 sec)

+------------+------------+
| float_ages | floor_ages |
+------------+------------+
|      12.30 |          0 |
|      45.60 |          0 |
|      78.90 |          0 |
|      -3.40 |          0 |
+------------+------------+
4 rows in set (0.000 sec)
```

```
Query OK, 4 rows affected (0.001 sec)
Rows matched: 4  Changed: 4  Warnings: 0

+------------+------------+
|  float_ages |  floor_ages |
+------------+------------+
|      12.30 |         12 |
|      45.60 |         45 |
|      78.90 |         78 |
|      -3.40 |         -4 |
+------------+------------+
4 rows in set (0.000 sec)

Query OK, 4 rows affected (0.014 sec)
Records: 4  Duplicates: 0  Warnings: 0

+------------+------+------+-----+---------+-------+
| Field      | Type | Null | Key | Default | Extra |
+------------+------+------+-----+---------+-------+
| int_ages   | int  | YES  |     | NULL    |       |
| floor_ages | int  | YES  |     | NULL    |       |
+------------+------+------+-----+---------+-------+
2 rows in set (0.001 sec)

+----------+------------+
| int_ages |  floor_ages |
+----------+------------+
|       12 |         12 |
|       46 |         45 |
|       79 |         78 |
|       -3 |         -4 |
+----------+------------+
4 rows in set (0.000 sec)

+-----------------+
| @min_value := 2 |
+-----------------+
|               2 |
+-----------------+
1 row in set, 1 warning (0.000 sec)

+----------+------------+
| int_ages |  floor_ages |
+----------+------------+
|       -3 |         -4 |
+----------+------------+
1 row in set (0.000 sec)
```

What are Referential Constraints?

Referential constraints (also called constraints) prevent the insertion of invalid data into database tables. In general, constraints on a table are specified during the creation of the table. Here is a list of constraints that SQL implementations support:

- CHECK
- DEFAULT
- FOREIGN KEY

- PRIMARY KEY
- NOT NULL
- UNIQUE

In case you don't already know, an *orphan row* in a database table is a row without its associated parent row that's typically stored in a separate table. An example would be a customer in the (parent) customers table and the associated (child) rows in the purchase_orders table. Note that a similar relationship exists between the (parent) purchase_orders table and the associated (child) rows in the line_items table.

COMBINING DATA FOR A TABLE UPDATE (OPTIONAL)

This section shows you how to perform the task described in an earlier section: how to merge two CSV files and load the result into a database table. This section is optional because the solution involves Pandas, which has not been discussed yet. You can skip this section with no loss of continuity, and perhaps return to this section when you need to perform this task. There are other ways to perform the tasks in this section.

The first subsection shows you how to *merge* the columns of a CSV file into the columns of another CSV file, and then save the updated CSV file to the file system. The second subsection shows you how to *append* the contents of a CSV file to the contents of another CSV file, and then save the updated CSV file to the file system.

Merging Data Columns in Multiple CSV Files via Pandas

Suppose that we have a CSV file called user.csv with a set of columns and that we want to merge the columns of user.csv with columns of the CSV file user2.csv. For simplicity, let's assume that there are no missing values in either CSV file. In this scenario, we will create a new CSV file whose rows and columns are from two CSV files. In other scenarios, you might need to select only a subset of the columns from two (or more) CSV files.

Listing 2.6 shows the content of user.csv that contains the original data for the user table, and Listing 2.7 shows the content of user2.csv that contains the data for the fname and lname attributes.

LISTING 2.6: user.csv

```
fname,lname
id,title
1000,Developer
2000,Project Lead
3000,Dev Manager
4000,Senior Dev Manager
```

LISTING 2.7: user2.csv

```
fname,lname
1000,John,Smith
2000,Jane,Stone
3000,Dave,Dodds
4000,Jack,Jones
```

Listing 2.8 shows the content of `user_merged.py` that illustrates how to use Pandas data frames to merge two CSV files and generate a new CSV file with the merged data.

LISTING 2.8: user_merged.py

```
import pandas as pd

df_user = pd.read_csv("user.csv")
df_user2 = pd.read_csv("user2.csv")
df_user['fname'] = df_user2['fname'].values
df_user['lname'] = df_user2['lname'].values
df_user.to_csv('user_merged.csv', index=False)
```

Listing 2.8 contains an `import` statement followed by the assignment of the contents of `user.csv` and `user2.csv` to the Pandas data frames `df_user` and `df_user2`, respectively.

The next pair of code snippets creates the columns `fname` and `lname` in the `df_users` data frame and initializes their values from the corresponding columns in the `df_user2` data frame.

The last code snippet in Listing 2.8 saves the updated data frame to the CSV file `user_merged.csv`, which is located in the same directory as the CSV files `user.csv` and `user2.csv`. Launch the code in Listing 2.8 to generate the CSV file `user_merged.csv`, whose contents are shown in Listing 2.9.

LISTING 2.9: user_merged.csv

```
id,title,fname,lname
1000,Developer,John,Smith
2000,Project Lead,Jane,Stone
3000,Dev Manager,Dave,Dodds
4000,Senior Dev Manager,Jack,Jones
```

Note: If need be, the code in Listing 2.8 can be modified to insert the `fname` values and the `lname` values in the first two columns.

Concatenating Data from Multiple CSV Files

Suppose that we have two CSV files called `user_merged.csv` and `user_merged2.csv` that contain the same columns, and you want to append the rows of the latter file to the rows of the former file. For simplicity, let's also

assume that there are no missing values in either CSV file. Listing 2.10 shows the content of `user_merged.csv` and Listing 2.11 shows the content of `user_merged2.csv`.

LISTING 2.10: user_merged.csv

```
id,title,fname,lname
5000,Developer,Sara,Edwards
6000,Project Lead,Beth,Woodward
7000,Dev Manager,Donald,Jackson
8000,Senior Dev Manager,Steve,Edwards
```

LISTING 2.11: user_merged2.csv

```
id,title,fname,lname
5000,Developer,Sara,Edwards
6000,Project Lead,Beth,Woodward
7000,Dev Manager,Donald,Jackson
8000,Senior Dev Manager,Steve,Edwards
```

Listing 2.12 shows the content of `merge_all_data.py` that illustrates how to use Pandas data frames to concatenate the contents of two or more CSV files in the same directory and generate a CSV file with the merged data. This code sample generalizes the code in Listing 2.8 that concatenates only two CSV files.

LISTING 2.12: merge_all_data.py

```
import glob
import os
import pandas as pd

# merge the data-related files as one data frame:
df = pd.concat(map(pd.read_csv, glob.glob(os.path.join('', 'data*.csv'))))

# save data frame to a CSV file:
df.to_csv('all_data.csv', index=False)
```

Listing 2.12 contains `import` statements followed by initializing the Pandas data frame `df` with the result of reading all the CSV files in the current directory and then concatenating their contents.

The last code snippet in Listing 2.12 saves the data frame `df` to the CSV file `all_data.csv`, which is located in the same directory as the other CSV files. Launch the code in Listing 2.12 to generate the CSV file `all_data.csv`, whose contents are given in Listing 2.13.

LISTING 2.13: all_data.csv

```
id,title,fname,lname
id,title
1000,Developer
2000,Project Lead
```

```
3000,Dev Manager
4000,Senior Dev Manager
1000,Developer
2000,Project Lead
3000,Dev Manager
4000,Senior Dev Manager
1000,Developer
2000,Project Lead
3000,Dev Manager
4000,Senior Dev Manager
1000,Developer
2000,Project Lead
3000,Dev Manager
4000,Senior Dev Manager
```

Appending Table Data from CSV Files via SQL

Suppose that the data in the CSV file `user_merged.csv` has already been inserted into table `user3`. We can use the following SQL statement to insert the contents of the CSV file `user_merged2.csv` into the table `user3` as follows:

```
LOAD DATA INFILE 'user_merged.csv'
    INTO TABLE user3
    FIELDS TERMINATED BY ','
    ENCLOSED BY '"'
    LINES TERMINATED BY '/n';
```

Depending on the manner in which the MySQL server was launched, you might encounter the following error message:

```
ERROR 1290 (HY000): The MySQL server is running with the --secure-
file-priv option so it cannot execute this statement
```

The preceding error occurs due to either of the following reasons:

- the SQL statement specified an incorrect path to the file
- no directory is specified under the `--secure--file--priv` variable

```
Select @@global.secure_file_priv;
+---------------------------+
| @@global.secure_file_priv  |
+---------------------------+
| NULL                      |
+---------------------------+
1 row in set (0.001 sec)
```

Another similar query is as follows:

```
SHOW VARIABLES LIKE "secure_file_priv";
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| secure_file_priv | NULL  |
+------------------+-------+
1 row in set (0.022 sec)
```

If you have verified that the path to the file is correct and you still see the same error message, then launch the following command (requires `root` access):

```
sudo /usr/local/mysql/support-files/mysql.server restart  --
secure_file_priv=/tmp
```

You might need to replace the preceding command with a command that is specific to your system, which depends on a combination of the following:

1. the operating system (Windows/Mac/Linux)
2. the version of MySQL on your system
3. the utility that installed MySQL (brew, .dmg file, and so forth)

Perform an online search to find a solution that is specific to your MySQL installation on your machine. Some solutions specify modifying the file `/etc/my.ini` or `/etc/my.cnf`, neither of which exists on Mac Catalina with MySQL 8.

Another possibility is the following SQL statement that specifies `LOCAL`:

```
LOAD DATA LOCAL INFILE "user_merged.csv" INTO TABLE user3;
```

Unfortunately, the preceding SQL statement does not work with MySQL 8: you will see the following error message:

```
ERROR 3948 (42000): Loading local data is disabled; this
must be enabled on both the client and server sides
```

Fortunately, Chapter 6 contains a MySQL Workbench section that shows you how to export tables and databases via a GUI interface, and also how to import databases and CSV files into tables.

INSERTING DATA INTO DATABASE TABLES

In Chapter 1, you saw how to create database tables and also how to insert data into those tables via SQL statements. For your convenience, the SQL statements from Chapter 1 are reproduced here:

```
use mytools;

-- create a new customer:
INSERT INTO customers
VALUES (1000,'John','Smith','123 Main St','Fremont','CA','94123');

-- create a new purchase order:
INSERT INTO purchase_orders VALUES (1000,12500, '2021-12-01');

-- line item => one hammer:
INSERT INTO line_items VALUES (12500,5001,100,1,20.00,2.00,22.00);
```

```
-- line item => two screwdrivers:
INSERT INTO line_items VALUES (12500,5002,200,2,8.00,1.60,17.60);

-- line item => three wrenches:
INSERT INTO line_items VALUES (12500,5003,300,3,10.00,3.00,33.20);
```

You can also create a SQL file that consists of multiple SQL INSERT statements that populate one or more tables with data. In addition, you can upload data from CSV files into database tables, which is discussed in the next section.

Populating Tables from Text Files

This section will show you how to create a database table and populate that table with data from a CSV file. Log into MySQL, select the mytools database, and invoke the following command to create the people table:

```
MySQL [mytools]> use mytools;

CREATE TABLE people (fname VARCHAR(20), lname VARCHAR(20),
age VARCHAR(20), gender CHAR(1), country VARCHAR(20));
```

Describe the structure of the people table with the following command:

```
MySQL [mytools]> desc people;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| fname   | varchar(20) | YES  |     | NULL    |       |
| lname   | varchar(20) | YES  |     | NULL    |       |
| age     | varchar(20) | YES  |     | NULL    |       |
| gender  | char(1)     | YES  |     | NULL    |       |
| country | varchar(20) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
5 rows in set (0.002 sec)
```

Listing 2.14 shows the content of people.csv that contains data for inserting into the people table.

LISTING 2.14: people.csv

```
fname,lname,age,gender,country
john,smith,30,m,usa
jane,smith,31,f,france
jack,jones,32,m,france
dave,stone,33,m,italy
sara,stein,34,f,germany
```

Listing 2.15 shows the content of people.sql that contains several SQL commands for inserting data into the people table.

LISTING 2.15: people.sql

```
INSERT INTO people VALUES ('john','smith','30','m','usa');
INSERT INTO people VALUES ('jane','smith','31','f','france');
```

```
INSERT INTO people VALUES ('jack','jones','32','m','france');
INSERT INTO people VALUES ('dave','stone','33','m','italy');
INSERT INTO people VALUES ('sara','stein','34','f','germany');
INSERT INTO people VALUES ('eddy','bower','35','m','spain');
```

As you can see, the `INSERT` statements in Listing 2.15 contain data that is located in `people.csv`. In Chapter 6, you will see an example of a Unix shell script that generates SQL statements from a CSV file. Now, log into MySQL, select the `mytools` database, and invoke the following command to populate the `people` table:

```
MySQL [mysql]> use mytools;

source people.sql;
Query OK, 1 row affected (0.004 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
```

Execute the following SQL statement to display the contents of the `people` table:

```
MySQL [mysql]> select * from people;
+-------+-------+------+--------+---------+
| fname | lname | age  | gender | country |
+-------+-------+------+--------+---------+
| john  | smith | 30   | m      | usa     |
| jane  | smith | 31   | f      | france  |
| jack  | jones | 32   | m      | france  |
| dave  | stone | 33   | m      | italy   |
| sara  | stein | 34   | f      | germany |
| eddy  | bower | 35   | m      | spain   |
+-------+-------+------+--------+---------+
6 rows in set (0.000 sec)
```

The second option involves manually executing each SQL statement in Listing 2.15, which is obviously inefficient for a large number of rows. The third option involves loading data from a CSV file into a table:

```
MySQL [mysql]> LOAD DATA LOCAL INFILE 'people.csv' INTO
TABLE people;
```

However, you might encounter the following error (which depends on the configuration of MySQL on your machine):

```
ERROR 3948 (42000): Loading local data is disabled; this
must be enabled on both the client and server sides
```

In general, a SQL script is preferred because it's easy to execute multiple times; you can also schedule SQL scripts to run as "cron" jobs.

WORKING WITH SIMPLE SELECT STATEMENTS

Earlier in this chapter, you saw examples of the SELECT keyword in SQL statements, and this section contains additional SQL statements to show you additional ways to select subsets of data from a table. In its simplest form, a SQL statement with the SELECT keyword looks like this:

```
SELECT [one-or-more-attributes]
FROM [one-or-more-tables]
```

Specify an asterisk ("*") after the SELECT statement if you want to select all the attributes of a table. For example, the following SQL statement illustrates how to select all rows from the people table:

```
MySQL [mytools]> select * from people;
+-------+-------+------+--------+---------+
| fname | lname | age  | gender | country |
+-------+-------+------+--------+---------+
| john  | smith | 30   | m      | usa     |
| jane  | smith | 31   | f      | france  |
| jack  | jones | 32   | m      | france  |
| dave  | stone | 33   | m      | italy   |
| sara  | stein | 34   | f      | germany |
| eddy  | bower | 35   | m      | spain   |
+-------+-------+------+--------+---------+
6 rows in set (0.000 sec)
```

Issue the following SQL statement that contains the LIMIT keyword (with additional examples later in this chapter) if you want only the first row from the people table:

```
select * from people limit 1;
+-------+-------+------+--------+---------+
| fname | lname | age  | gender | country |
+-------+-------+------+--------+---------+
| john  | smith | 30   | m      | usa     |
+-------+-------+------+--------+---------+
1 row in set (0.000 sec)
```

Replace the number 1 in the previous SQL query with any other positive integer to display the number of rows that you need. Incidentally, if you replace the number 1 with the number 0, you will see 0 rows returned,

Moreover, include a WHERE keyword to specify a condition on the rows, which will return a (possibly empty) subset of rows from the specified table:

```
SELECT [one-or-more-attributes]
FROM [one-or-more-tables]
WHERE [some condition]
```

For example, the following SQL statement illustrates how to display all the attributes of the rows in the `people` table where the first name is `john`:

```
MySQL [mytools]> select * from people where fname = 'john';
+-------+-------+------+--------+---------+
| fname | lname | age  | gender | country |
+-------+-------+------+--------+---------+
| john  | smith | 30   | m      | usa     |
+-------+-------+------+--------+---------+
1 row in set (0.000 sec)
```

Include `ORDER BY` to specify the order in which you want to display the rows:

```
SELECT *
FROM weather
ORDER BY city;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
+------------+--------+------+----------+------+-------+
11 rows in set (0.003 sec)
```

In Chapter 3, you will learn how to use the `JOIN` keyword in order to retrieve data from two tables, and in Chapter 4, you will learn how to specify `GROUP BY` and `HAVING` in SQL statements.

Duplicate Versus Distinct Rows

Unless it's explicitly stated, the default action for a SQL `SELECT` statement is to select all rows (which includes duplicates), as shown here:

```
SELECT year
FROM employees;
+------+
| year |
+------+
| 2020 |
| 2020 |
| 2020 |
| 2020 |
| 2020 |
| 2020 |
| 2020 |
```

```
| 2020 |
| 2020 |
| 2020 |
| 2020 |
| 2020 |
+------+
12 rows in set (0.000 sec)
```

By contrast, the following SQL statement returns only a *single* row with the year 2020:

```
SELECT DISTINCT year
FROM employees;
+------+
| year |
+------+
| 2020 |
+------+
1 row in set (0.003 sec)
```

Later you will learn how to use the `GROUP BY` clause and the `HAVING` clause in SQL statements.

Unique Rows Versus Distinct Rows

The `UNIQUE` keyword selects a row only if that row does not have any duplicates. A SQL query that contains the `UNIQUE` keyword returns the same result set as a query that contains the `DISTINCT` keyword *if and only if* there are no duplicate rows.

As a preview, the following SQL query contains a SQL subquery, which is a topic that is discussed in detail in Chapter 3. However, the SQL query is included in this section of the chapter so that you can compare the functionality of `DISTINCT` versus `UNIQUE`. With the preceding in mind, here is the SQL statement to find unique rows in a database table:

```
select city, state
from weather
where unique (select state from weather);
```

If you are unfamiliar with SQL subqueries, you can return to this example after you learn about them in one of the sections in Chapter 3.

The EXISTS Keyword

The `EXISTS` keyword selects a row based on the existence of a particular value in an attribute of a table. This section shows you two SQL statements that use the `EXISTS` keyword: one statement involves a subquery (details are provided in Chapter 3) and the second involves a `SELECT` keyword.

The purpose of showing both SQL statements is to illustrate that sometimes a SQL statement can be replaced by an equivalent SQL statement that is much easier to understand (and might also be more efficient). For example,

the following SQL statement checks for the string "abc" in the `city` attribute of the `weather` table:

```
select city, state
from weather
where exists
  (select city from weather where city = 'abc');
Empty set (0.001 sec)
```

The preceding is somewhat contrived because it can be replaced with this simpler and intuitive SQL query:

```
select city, state
from weather
where city = 'abc';
```

The LIMIT Keyword

The `LIMIT` keyword limits the number of rows that are in a result set. For example, the `weather` table contains 11 rows, as shown here:

```
SELECT COUNT(*)
FROM weather;
+----------+
| count(*) |
+----------+
|       11 |
+----------+
1 row in set (0.001 sec)
```

If you want to see only *three* rows instead of all the rows in the `weather` table, issue the following SQL query:

```
SELECT city,state
FROM weather ORDER
BY state, city
LIMIT 3;
+------+-------+
| city | state |
+------+-------+
|      | ca    |
| sf   | ca    |
| sf   | ca    |
+------+-------+
3 rows in set (0.000 sec)
```

DELETE, TRUNCATE, AND DROP IN SQL

The following list summarizes the various ways of removing data from a database table:

- The DELETE keyword deletes the data in a table but leaves the table intact.
- The TRUNCATE keyword is a faster way to delete the data in a table.

- The TRUNCATE keyword also preserves the table structure.
- The DROP keyword drops the data and the table itself from a database.

Here is an example of deleting *all* the data from a table using the DELETE keyword:

```
DELETE from customers;
```

However, if a database table has a large number of rows, a faster technique is the TRUNCATE statement, as shown here:

```
TRUNCATE customers;
```

Both of the preceding SQL commands involve removing rows from a table without dropping the table. If you want to drop the rows in a table *and* the table, use the DROP statement as shown here:

```
DROP TABLE IF EXISTS customers;
```

SELECT, DELETE, and LIMIT Combinations

Another useful combination involves the SELECT and DELETE keyword when you want to delete a row in a database table. For example, execute a SQL statement with the SELECT keyword before you delete any rows:

```
SELECT *
FROM table_name
WHERE lname = 'SMITH';
```

If the preceding SQL statement returns the row (or rows) that you want to delete, then you can safely issue the following DELETE statement:

```
DELETE
FROM table_name
WHERE lname = 'SMITH';
```

As variant of the preceding pair of SQL statements, you can also specify the LIMIT keyword (with a suitable integer value) to limit the number of rows that are returned:

```
SELECT *
FROM table_name
WHERE lname = 'SMITH'
LIMIT 1;

DELETE
FROM table_name
WHERE lname = 'John'
LIMIT 1;
```

Keep in mind the following caveat: the preceding SQL statement will delete one row, but there is an exception. The preceding SQL query will delete *all* rows whose name equals John if you specify ON DELETE CASCADE in the table definition of the customers table.

More Options for the DELETE Statement in SQL

The preceding section showed you how to delete all the rows in a table, and this section shows you how to delete a subset of the rows in a table, which involves specifying a condition for the rows that you want to drop from a table.

The following SQL statement deletes the rows in the `customers` table where the first name is `John`:

```
DELETE
FROM customers
Where FNAME = 'John';
```

The next SQL statement deletes the rows in the `customers` table where the first name is `John` and the rows in the `purchase_orders` table that are associated with `John`:

```
DELETE
FROM customers
Where FNAME = 'John'
CASCADE;
```

The preceding SQL statement is called a *cascading delete,* which is very useful when the rows in a table have external dependencies, such as the `customers` table that has a one-to-many relationship with the `purchase_orders` table.

If you remove a "parent" row that appears in the `customers` table, you need to remove the "child" rows from the `purchase_orders` table. Otherwise, you will have "orphan" purchase orders that do not have a corresponding row in the `customers` table. The final section of this chapter contains more information about different types of relationships that can exist between tables.

CREATING TABLES FROM EXISTING TABLES IN SQL

SQL provides two ways to create new tables without an explicit list of attributes for the new table. One technique involves a SQL statement that contains the `TEMPORARY` keyword, and the second technique does not specify the `TEMPORARY` keyword.

Although MySQL supports temporary tables, that support provides limited functionality. In particular, you cannot assign values to variables nor can you create global templates. As a side note, MySQL also supports *memory-stored tables*, but such tables cannot be accessed during transactions. Furthermore, memory-stored tables are used only for read operations.

A *temporary* table is useful when it's impractical to query data that requires a single `SELECT` statement with `JOIN` clauses. Instead, use a temporary table to store an immediate result and then process that data with other SQL queries. However, keep in mind that the query optimizer that improves performance of SQL statements cannot optimize SQL queries containing a temporary table.

Working with Temporary Tables in SQL

Before we create a temporary table, let's drop the `temp_cust` table in case it already exists:

```
MySQL [mytools]> DROP TEMPORARY TABLE IF EXISTS temp_cust;
Query OK, 0 rows affected, 1 warning (0.000 sec)
```

The following SQL statement illustrates how to create the temporary table `temp_cust` from the `customers` table:

```
MySQL [mytools]> CREATE TEMPORARY TABLE IF NOT EXISTS temp_cust
                 AS (SELECT * FROM customers);
Query OK, 1 row affected (0.019 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

The following SQL statement displays the structure of `temp_cust`:

```
MySQL [mytools]> DESC temp_cust;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| cust_id      | int         | YES  |     | NULL    | NULL  |
| first_name   | varchar(20) | YES  |     | NULL    | NULL  |
| last_name    | varchar(20) | YES  |     | NULL    | NULL  |
| home_address | varchar(20) | YES  |     | NULL    | NULL  |
| city         | varchar(20) | YES  |     | NULL    | NULL  |
| state        | varchar(20) | YES  |     | NULL    | NULL  |
| zip_code     | varchar(10) | YES  |     | NULL    | NULL  |
+--------------+-------------+------+-----+---------+-------+
7 rows in set (0.005 sec)
```

The `temp_cust` table contains the same data as the `customers` table, as shown here:

```
MySQL [mytools]> SELECT * FROM temp_cust;
+---------+------------+-----------+--------------+---------+-------+----------+
| cust_id | first_name | last_name | home_address | city    | state | zip_code |
+---------+------------+-----------+--------------+---------+-------+----------+
|    1000 | John       | Smith     | 123 Main St  | Fremont | CA    | 94123    |
+---------+------------+-----------+--------------+---------+-------+----------+
1 row in set (0.001 sec)
```

In addition, you can specify an index for a temporary table, as shown here:

```
CREATE TEMPORARY TABLE IF NOT EXISTS
  temp_cust3 ( INDEX(last_name) )
ENGINE=MyISAM
AS (
  SELECT first_name, last_name
  FROM customers
);
```

Alternatively, you can create a temporary table and specify the MySQL engine MEMORY, as shown here:

```
MySQL [mytools]> CREATE TEMPORARY TABLE temp_cust4 ENGINE=MEMORY
    -> as (select * from customers);
Query OK, 1 row affected (0.003 sec)
Records: 1  Duplicates: 0  Warnings: 0
```

However, keep in mind the following point: `ENGINE=MEMORY` is *not* supported when table contains `BLOB/TEXT` columns. Now that you understand how to create tables with the `TEMPORARY` keyword, let's look at the preceding SQL statements when we omit the `TEMPORARY` keyword.

Creating Copies of Existing Tables in SQL

Another technique to create a copy of an existing table is to execute the previous SQL statements *without* the `TEMPORARY` keyword, as shown here:

```
MySQL [mytools]> DROP TABLE IF EXISTS temp_cust2;
Query OK, 0 rows affected, 1 warning (0.008 sec)

MySQL [mytools]> CREATE TABLE IF NOT EXISTS temp_cust2
                AS (SELECT * FROM customers);
Query OK, 1 row affected (0.028 sec)
Records: 1  Duplicates: 0  Warnings: 0

MySQL [mytools]> SELECT COUNT(*) FROM temp_cust2;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.009 sec)
```

If you need to create a table that has the same structure as an existing table but does not contain any data, you can do so with the following type of SQL statement:

```
MySQL [mytools]> DROP TABLE IF EXISTS abc2;
Query OK, 0 rows affected (0.018 sec)

MySQL [mytools]> CREATE TABLE abc2 LIKE weather;
Query OK, 0 rows affected (0.025 sec)

MySQL [mytools]> SELECT * FROM abc2;
Empty set (0.001 sec)
```

WHAT IS A SQL INDEX?

An *index* is a mechanism that enables a faster retrieval of records from database tables and therefore improves performance. An index contains an

entry that corresponds to each row in a table, and the index itself is stored in a tree-like structure. SQL enables you to define one or more indexes for a table, and some guidelines are provided in a subsequent section.

By way of analogy, the index of a book enables you to search for a word or a term, locate the associated page number(s), and then you can navigate to one of those pages. Clearly, the use of the book index is much fast than looking sequentially through every page in a book.

Types of Indexes

A *unique index* prevents duplicate values in a column, provided that the column is also uniquely indexed, which can be performed automatically if a table has a primary key.

A *clustered index* actually changes the order of the rows in a table, and then performs a search that is based in the key values. A table can have only *one* clustered index. A clustered index is useful for optimizing DML statements for tables that use the InnoDB engine (discussed briefly in chapter six).

MySQL 8 introduced *invisible indexes*, but those indexes are unavailable for the query optimizer. MySQL ensures that those indexes are kept current when data in the referenced column are modified. You can make indexes invisible by explicitly declare their visibility during table creation or via the `ALTER TABLE` command, as you will see in a later section.

Creating an Index

An index on a MySQL table can be defined in two ways:

- As part of the table definition during table creation
- After the table has been created

Here is an example of creating an index on the `full_name` attribute *during* the creation of the table `friend_table`:

```
DROP TABLE IF EXISTS friend_table;

CREATE TABLE friend_table  (
  friend_id int(8) NOT NULL AUTO_INCREMENT,
  full_name varchar(40) NOT NULL,
  fname varchar(20) NOT NULL,
  lname varchar(20) NOT NULL,
  PRIMARY KEY (friend_id),INDEX(full_name)
);
```

Here is an example of creating index `friend_lname_idx` on the `lname` attribute *after* the creation of the table `friend_table`:

```
CREATE INDEX friend_lname_idx ON friend_table(lname);
Query OK, 0 rows affected (0.035 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You can create an index on multiple columns, an example of which is shown here:

```
CREATE INDEX friend_lname_fname_idx ON friend_table(lname,fname);
```

An index on a MySQL table can specify a maximum of 16 indexed columns, and a table can contain a maximum of 64 secondary indexes.

Disabling and Enabling an Index

Sometimes, it's useful to disable indexes before performing some intensive operation, and then re-enable the indexes. The syntax for disabling an index is here:

```
alter table friend_table disable keys;
Query OK, 0 rows affected, 1 warning (0.004 sec)
```

The corresponding syntax for re-enabling an index is here:

```
alter table friend_table enable keys;
Query OK, 0 rows affected, 1 warning (0.002 sec)
```

View and Drop Indexes

As you probably guessed, you can drop specific indexes as well as display the indexes associated with a given table and also drop specific indexes. The following SQL statement drops the specified index on the `friend_table` table:

```
DROP INDEX friend_lname_fname_idx ON friend_table;
Query OK, 0 rows affected (0.011 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Invoke the preceding SQL statement again, and the following error message confirms that the index was dropped:

```
ERROR 1091 (42000): Can't DROP 'friend_lname_fname_idx';
check that column/key exists
```

You can also issue the following SQL statement to display the indexes that exist on the table `friend_table`:

```
SHOW INDEXES FROM friend_table;
+--------------+------------+------------------+--------------+-------------+--
---------+-------------+----------+--------+------+------------+---------+------
---------+---------+------------+
| Table        | Non_unique | Key_name         | Seq_in_index | Column_name |
Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment |
Index_comment | Visible | Expression |
```

```
+--------------+------------+-----------------+-------------+------------+--
 | friend_table |          0 | PRIMARY         |           1 | friend_id   | A
|           0 |     NULL |   NULL |       | BTREE      |          |
| YES     | NULL        |
| friend_table |          1 | full_name       |           1 | full_name   | A
         |            0 |     NULL |   NULL |       | BTREE      |          |
| YES     | NULL        |
| friend_table |          1 | friend_lname_idx |           1 | lname       | A
         |            0 |     NULL |   NULL |       | BTREE      |          |
| YES     | NULL        |
+--------------+------------+-----------------+-------------+------------+--
---------+------------+----------+--------+------+------------+---------+-----
----------+---------+------------+
3 rows in set (0.005 sec)
```

When you define a MySQL table, you can specify that an index is invisible with the following code snippet:

```
CREATE INDEX index_name ON table_name(column-list) INVISIBLE;
```

The following SQL statement displays the invisible indexes in MySQL, which is a new feature in version 8:

```
SHOW INDEXES FROM friend_table
WHERE VISIBLE = 'NO';
Empty set (0.003 sec)
```

Overhead of Indexes

An index occupies some memory on secondary storage. In general, if you issue a SQL statement that involves an index, that index is first loaded into memory and then it's utilized to access the appropriate record(s). A SQL query that involves simply accessing (reading) data via an index is almost always more efficient than accessing data without an index.

However, if a SQL statement *updates* records in one or more tables, then *all* the affected indexes must be updated. As a result, there can be a performance impact when multiple indexes are updated as a result of updating table data. Hence, it's important to determine a suitable number of indexes, and the columns in each of those indexes, which can be done either by experimentation (not recommended for beginners) or via open source or commercial tools that provide statistics regarding the performance of SQL statements when indexes are involved.

Considerations for Defining Indexes

As you might already know, a full table scan for large tables will likely be computationally expensive, so consider defining an index on columns that are referenced in the `WHERE` clause in your SQL statements. As a simple example, consider the following SQL statement:

```
SELECT *
FROM customers
WHERE lname = 'Smith';
```

If you do not have an index that starts with the `lname` *attribute of the* `customers` *table, then a full table scan is executed, which means every row is checked.*

Consider defining an index on attributes that appear in SQL query statements that involve `SELECT`, `GROUP BY`, `ORDER BY`, or `JOIN`. As mentioned earlier, updates to table data necessitate updates to indexes, which in turn can result in lower performance.

When to Disable Indexes on a Table

Although you can directly insert a large volume of data into a table (or tables), the following alternative can be more efficient:

1. Disable the indexes.
2. Insert the data.
3. Enable the indexes again.

Although the preceding approach involves rebuilding the indexes, which is performed after Step 3, you might see a performance improvement compared to directly inserting the table data. Of course, you could also try both approaches and calculate the time required to complete the data insertion.

As yet another option, it's possible to perform a multi-row insert in MySQL, which enables you to insert several rows with a single SQL statement, thereby reducing the number of times the indexes must be updated. The maximum number of rows that can be inserted via a multi-row insert depends on the value of `max_allowed_packet` (whose default value is 4M), as described at the following site:

https://dev.mysql.com/doc/refman/5.7/en/packet-too-large.html

Another suggestion: check the order of the columns in multi-column indexes and compare that order with the order of the columns in each index. *MySQL will only use an index if the left-leading column is included in the index.*

Selecting Columns for an Index

As mentioned in the previous section, an index of a database table is used if an attribute in the `WHERE` clause is the left-most column in the definition of an index. For example, the following SQL query specifies the `lname` attribute of the `users` table in the `WHERE` clause:

```
SELECT *
FROM users
WHERE lname = 'SMITH'
```

In the previous section, you learned that when the `users` table does not have an index containing the `lname` attribute, then a full table scan is executed and the contents of the `lname` attribute in every row is compared with `SMITH`.

The *average* number of comparisons in a full table scan is `n/2`, where `n` is the number of rows in the given table. Thus, a table containing 1,024 rows (which is a very modest size) requires an average of 512 comparisons, whereas a suitably defined index reduces the average number of comparisons to 10 (and sometimes even fewer comparisons).

Based on the preceding paragraph, indexes can be useful for improving the performance of read operations. In general, the candidates for inclusion in the definition of an index are the attributes that appear in frequently invoked SQL statements that select, join, group, or order data. However, the space requirement for indexes is related to the number of rows in the tables.

Furthermore, multiple indexes involve more memory, and they must be updated after a write operation, which can incur a performance penalty. An experienced DBA can provide you with very helpful advice regarding index definitions. You can also experiment with the number and type of indexes and profile your system to determine the optimal combination for your system. In addition, use SQL monitoring tools (discussed later) to determine which SQL operations are candidates for optimization.

Finding Columns Included in Indexes

This section contains SQL statements that are specific to MySQL: for information about other databases (such as Oracle), perform an online search to find the correct syntax. MySQL enables you to find columns that are included in indexes with this SQL statement:

```
SHOW INDEX FROM people;
```

You can also query the `STATISTICS` table in the `INFORMATION_SCHEMA` to show indexes for all tables in a schema, an example of which follows:

```
SELECT DISTINCT TABLE_NAME, INDEX_NAME
FROM INFORMATION_SCHEMA.STATISTICS
WHERE TABLE_SCHEMA = 'mytools';
```

ENHANCING THE MYTOOLS DATABASE (OPTIONAL)

A reporting system is obviously important for generating financial statements, billing statements, and *ad hoc* reports. For example, you can create an accounts receivable table `ar_mytools` to keep track of paid purchase orders (and the payment date) and unpaid purchase orders (with past due 30 and past due 60) as follows:

```
use mytools;
DROP TABLE IF EXISTS ar_mytools;

CREATE TABLE ar_mytools (cust_id INTEGER, po_id INTEGER, PURCH_DATE
date, PAID_DATE date, PAST_30 date, PAST_60 date);

-- two rows indicate that cust_id 1000 paid for two purchase orders:
INSERT INTO ar_mytools VALUES (1000, 12500,'2021-12-01','2021-12-15',
NULL, NULL);

INSERT INTO ar_mytools VALUES (1000, 12600,'2022-01-05','2022-01-10',
NULL, NULL);
```

You can also create SQL statements that retrieve a list of customers and purchase order details from the `ar_mytools` table that contain

1. paid purchase orders (per month or a date range)
2. unpaid purchase orders that are 30 days past due
3. unpaid purchase orders that are 60 days past due

You can use the information from #1 to send customers notifications about upcoming sales, reward points, discounts, and so forth. In addition, you can use the information from #2 and #3 to send reminder notifications to the relevant customers.

ENTITY RELATIONSHIPS

In Chapter 1, you were briefly introduced to several types of relationships that can exist between a pair of tables, which are as follows:

- one-to-many
- many-to-many
- self-referential

For example, the `customers` table has a one-to-many relationship with the `purchase_orders` table for each `cust_id` that makes a purchase. Similarly, each `purchase_order` in the `purchase_orders` table has one or more rows in the `line_items` table, and therefore the `purchase_orders` table has a one-to-many relationship with the `line_items` table.

Note that it's possible for a customer to register on the website and never make a purchase. In Chapter 4, you will see an example of a SQL statement that returns all customers who have never made any purchases.

An example of a many-to-many relationship is a `students` table and a `courses` table. Each student can enroll in one or more courses, and each course contains one or more students. This relationship is modeled by creating a so-called "join table" that is interposed between the `students` table and a `courses` table.

The primary key for this new "join table" is the union of the primary key for the `students` table *and* the primary key for the `courses` table. Thus, the `students` table and the `courses` table both have a one-to-many relationship with the intermediate join table.

An example of a *self referential* table is an `employees` table that contains the manager of each employee. However, if the `employees` table does *not* contain an attribute for the employee's manager (or some counterpart to this attribute), then the `employees` table is a not a self-referential table.

SUMMARY

This chapter introduced you to SQL and how to invoke various types of SQL statements. You saw how to create tables manually from the SQL prompt and also by launching a SQL script that contains SQL statements for creating tables.

You also learned how to drop tables, along with the effect of the `DELETE`, `TRUNCATE`, and `DROP` keywords in SQL statements. Next, you learned how to invoke a SQL statement to dynamically create a new table based on the structure of an existing table.

Then you saw an assortment of SQL statements that use the `SELECT` keyword. Examples of such SQL statements include finding the distinct rows in a MySQL table, or finding unique rows containing the `EXISTS` and `LIMIT` keywords. Moreover, you learned about the differences among the `DELETE`, `TRUNCATE`, and `DROP` keywords in SQL.

Next, you saw how to create indexes on MySQL tables, and some criteria for defining indexes, followed by how to select columns for an index.

Furthermore, you learned about entity relationships, such as one-to-many, that are very common in tables that have master-detail relationships, such as customers and purchase orders. You also learned about many-to-many relationships in use cases such as students and classes that they are enrolled in. Next, you learned about self-referential relationships, such as an `employees` table that contains data for employees as well as their managers.

CHAPTER 3

JOINS, VIEWS, AND SUBQUERIES

The major topics in this chapter involve SQL join statements, the creation of views, and SQL subqueries. You will also see SQL queries that contain the SQL clauses `ORDER BY`, `GROUP BY`, and `HAVING` in a SQL statement. This chapter relies on the material in Chapter 2 for creating MySQL tables, extracting data from those tables, and creating views over tables.

The first section of this chapter shows you SQL statements with various types of `JOIN` clauses on two MySQL tables, which can also be extended to multiple tables. Then you will learn about different types of keys, such as primary keys, unique keys, and foreign keys. The second section contains date-related examples, such as finding the year, month, and day of a date, as well as finding the week of a date.

The third section delves into other useful SQL clauses, such as using `GROUP BY` and `HAVING` in a SQL statement. This section shows you SQL statements that contain the `ROLLUP` keyword. You will be introduced to aggregate functions such as `COUNT(*)`, `MIN`, `MAX`, and `AVG`, which are also discussed in Chapter 4.

The final section discusses the concepts of one-to-many and many-to-many relationships between pairs of tables (briefly introduced in Chapter 1), along with some real-life scenarios. In addition, you will learn about self-referential tables.

QUERY EXECUTION ORDER IN SQL

There are many types of SQL keywords and clauses, and it's important to know the sequence in which these clauses can appear in a SQL statement. If you create a SQL statement with clauses that are not in the correct order, you will generate an error message. Instead of guessing the correct order of SQL clauses, here is the correct execution order:

- FROM, JOIN
- WHERE
- GROUP BY
- HAVING
- SELECT
- DISTINCT
- ORDER BY
- LIMIT, OFFSET

We already worked with the clauses SELECT, FROM, JOIN, and WHERE in the preceding list to create basic SQL statements. The other clauses provide extra functionality that enable you to produce sophisticated result sets that can be used as the basis for various reports to summarize aspects of your application and database.

JOINING TABLES IN SQL

An RDBMS whose tables are sufficiently normalized (discussed in Chapter 6) ensures that a given data value appears in a single location. A "single source of truth" for data is crucial whenever you need to update data values, thereby maintaining data integrity in your RDBMS.

You also need a mechanism by which you can retrieve logically related data that resides in multiple tables. Indeed, the JOIN family of keywords enables you to write SQL statements that retrieve such data from multiple tables.

Keep in mind that a SQL JOIN statement can require more execution time to retrieve data from multiple tables than working with one denormalized table that contains all the data values in a single table. However, try to limit a denormalized table to attributes that never (or rarely) need to be updated, thereby maintaining data integrity.

If you are not convinced of the preceding statement, consider this scenario: you have great performance in your SQL statements, but you aren't sure if all the data is correct. If you have mission critical data that requires 100% data integrity, then data integrity has a higher priority than optimal performance.

Fortunately, performance issues can sometimes be addressed by performing the appropriate denormalization of relevant tables. Note that this will involve rewriting the SQL statements that perform a JOIN of the normalized tables so that the new SQL statements query the single denormalized database table.

Types of SQL JOIN Statements

The JOIN keyword enables you to define various types of SQL statements that have slightly different semantics:

- INNER JOIN
- LEFT OUTER JOIN

- RIGHT OUTER JOIN
- CROSS JOIN
- SELF-JOIN

Let's suppose that table `A` has some (but not all) corresponding rows in table `B`, and that table `B` has some (but not all) corresponding rows in table `A`. Moreover, let's also assume that a `JOIN` statement specifies table `A` first and then table `B`.

An `INNER JOIN` returns all rows from table `A` that have *non-empty* matching rows in another table.

A `LEFT JOIN` returns all rows from left-side table `A` and *either* matching rows from the right-side table `B` *or* `NULL` if no matching rows in right-side table `B`.

A `RIGHT JOIN` returns all rows from right-side table `B` and *either* matching rows from the left-side table `A` *or* `NULL` if no matching rows in table `A`.

A `CROSS JOIN` is a Cartesian or "full" product of rows from left-side table `A` and right-side table `B`.

A `SELF JOIN` joins a table to itself. A common use-case involves an `employees` table that contains a manager attribute for each employee. Given an employee in this table, find the value in the manager attribute for that employee, and then search the `employees` table a second time using the manager attribute.

This sequence of steps can be repeated until the top-most employee does not have a manager (such as the CEO). Given an employee, the preceding sequence produces the management hierarchy from the employee to the top-most employee in a company (defined in the table).

EXAMPLES OF SQL JOIN STATEMENTS

A SQL statement with a `JOIN` clause is required whenever information about an entity is stored in two (or more) tables. For example, recall that our four-table RDBMS contains the following tables:

- `customers`
- `purchase_orders`
- `line_items`
- `item_desc`

Let's look at the structure and the contents of the `purchase_orders` table and the `line_items` table and before we execute `JOIN` queries on those tables.

```
desc customers;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| cust_id      | int         | YES  |     | NULL    |       |
| first_name   | varchar(20) | YES  |     | NULL    |       |
| last_name    | varchar(20) | YES  |     | NULL    |       |
| home_address | varchar(20) | YES  |     | NULL    |       |
| city         | varchar(20) | YES  |     | NULL    |       |
| state        | varchar(20) | YES  |     | NULL    |       |
| zip_code     | varchar(10) | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
7 rows in set (0.004 sec)

select cust_id, first_name, last_name
from customers;
+---------+------------+-----------+
| cust_id | first_name | last_name |
+---------+------------+-----------+
|    1000 | John       | Smith     |
|    2000 | Jane       | Jones     |
+---------+------------+-----------+
2 rows in set (0.000 sec)
```

Let's look at the structure of the `purchase_orders` table and then extract some rows from this table based on a subset of the table attributes.

```
desc purchase_orders;
+---------------+------+------+-----+---------+-------+
| Field         | Type | Null | Key | Default | Extra |
+---------------+------+------+-----+---------+-------+
| cust_id       | int  | YES  |     | NULL    |       |
| po_id         | int  | YES  |     | NULL    |       |
| purchase_date | date | YES  |     | NULL    |       |
+---------------+------+------+-----+---------+-------+
3 rows in set (0.016 sec)

select * from purchase_orders;
+---------+-------+---------------+
| cust_id | po_id | purchase_date |
+---------+-------+---------------+
|    1000 | 12500 | 2021-12-01    |
|    1000 | 12600 | 2022-12-03    |
|    1000 | 12700 | 2022-05-07    |
+---------+-------+---------------+
3 rows in set (0.001 sec)

desc line_items;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| po_id         | int          | YES  |     | NULL    |       |
| line_id       | int          | YES  |     | NULL    |       |
| item_id       | int          | YES  |     | NULL    |       |
| item_count    | int          | YES  |     | NULL    |       |
| item_price    | decimal(8,2) | YES  |     | NULL    |       |
| item_tax      | decimal(8,2) | YES  |     | NULL    |       |
| item_subtotal | decimal(8,2) | YES  |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
7 rows in set (0.005 sec)
```

```
select po_id, line_id, item_id from line_items;
+-------+---------+---------+
| po_id | line_id | item_id |
+-------+---------+---------+
| 12500 |    5001 |     100 |
| 12500 |    5002 |     200 |
| 12500 |    5003 |     300 |
+-------+---------+---------+
3 rows in set (0.003 sec)

select po_id, cust_id, purchase_date
from purchase_orders;
+-------+---------+---------------+
| po_id | cust_id | purchase_date |
+-------+---------+---------------+
| 12500 |    1000 | 2021-12-01    |
| 12600 |    1000 | 2022-12-03    |
| 12700 |    1000 | 2022-05-07    |
+-------+---------+---------------+
3 rows in set (0.003 sec)
```

Now we're ready to examine different `JOIN` clauses involving this pair of tables, starting with an `INNER JOIN` that is discussed in the next section.

An INNER JOIN Statement

As you saw in the preceding section, the `customers` table contains two customers, but only one customer has an associated purchase order, and that customer has `cust_id` equal to 1000. Therefore, a SQL statement that specifies an `INNER JOIN` on the `customers` and `purchase_order` tables will return rows whose left-side and right-side are both non-empty and *all* those rows will have a `cust_id` value equal to 1000. Just to confirm the preceding statement, let's define an `INNER JOIN` on the `customers` and `purchase_order` tables.

```
SELECT customers.cust_id, customers.first_name,
customers.last_name, purchase_orders.purchase_date
FROM customers
INNER JOIN purchase_orders
ON customers.cust_id = purchase_orders.cust_id;
```

The result of the preceding SQL query is here, and observe that the `cust_id` in all three rows equals 1000:

```
+---------+------------+-----------+---------------+
| cust_id | first_name | last_name | purchase_date |
+---------+------------+-----------+---------------+
|    1000 | John       | Smith     | 2021-12-01    |
|    1000 | John       | Smith     | 2022-12-03    |
|    1000 | John       | Smith     | 2022-05-07    |
+---------+------------+-----------+---------------+
3 rows in set (0.003 sec)
```

A LEFT JOIN Statement

We know that the customer whose `cust_id` is 2000 does not have a corresponding row in the `purchase_orders` table, which means that a `LEFT JOIN` involving the `cust_id` with value 2000 will be non-null row entry on the left side, whereas the right side will be `NULL`. Let's confirm the preceding sentence with the following `LEFT JOIN` statement:

```
SELECT customers.cust_id, customers.first_name,
customers.last_name, purchase_orders.purchase_date
FROM customers
LEFT JOIN purchase_orders
ON customers.cust_id = purchase_orders.cust_id;
+---------+------------+-----------+---------------+
| cust_id | first_name | last_name | purchase_date |
+---------+------------+-----------+---------------+
|    1000 | John       | Smith     | 2021-12-01    |
|    1000 | John       | Smith     | 2022-12-03    |
|    1000 | John       | Smith     | 2022-05-07    |
|    2000 | Jane       | Jones     | NULL          |
+---------+------------+-----------+---------------+
4 rows in set (0.000 sec)
```

A RIGHT JOIN Statement

Let's perform a `RIGHT JOIN`, as shown in the following SQL statement:

```
SELECT customers.cust_id, customers.first_name,
customers.last_name, purchase_orders.purchase_date
FROM customers
RIGHT JOIN purchase_orders
ON customers.cust_id = purchase_orders.cust_id;
+---------+------------+-----------+---------------+
| cust_id | first_name | last_name | purchase_date |
+---------+------------+-----------+---------------+
|    1000 | John       | Smith     | 2021-12-01    |
|    1000 | John       | Smith     | 2022-12-03    |
|    1000 | John       | Smith     | 2022-05-07    |
+---------+------------+-----------+---------------+
3 rows in set (0.000 sec)
```

Notice that there are no `NULL` values in the `RIGHT JOIN`, because every row in the `purchase_orders` table has a corresponding row in the left-side `customers` table. If a left-side `NULL` appears, then the right-side table rows are called *orphans* because they do not have a parent in the `customers` table. Hence, we would have purchase orders but no information regarding the customer who made that purchase. *We want to ensure that there are never any orphan rows in our tables.*

If we need to delete a customer from the `customers` table, we must delete the associated rows from the `purchase_orders` table as well as the associated rows from the `line_items` table. Fortunately, this task is straightforward: instead of specifying the `DELETE` keyword in our SQL statements, we specify

DELETE CASCADING to delete the child rows in the purchase_orders table as well as the child rows in the line_items table.

A CROSS JOIN Statement

Now let's perform a CROSS JOIN, as shown here:

```
SELECT customers.cust_id, customers.first_name,
customers.last_name, purchase_orders.purchase_date
FROM customers
CROSS JOIN purchase_orders
ON customers.cust_id = purchase_orders.cust_id;
+---------+------------+-----------+---------------+
| cust_id | first_name  | last_name | purchase_date |
+---------+------------+-----------+---------------+
|    1000 | John        | Smith     | 2021-12-01    |
|    1000 | John        | Smith     | 2022-12-03    |
|    1000 | John        | Smith     | 2022-05-07    |
+---------+------------+-----------+---------------+
3 rows in set (0.000 sec)
```

Notice that the preceding output does *not* contain data for the customer whose cust_id is 2000. However, we *do* get the correct results with the following SQL query:

```
SELECT customers.cust_id, customers.first_name,
customers.last_name, purchase_orders.purchase_date
FROM customers
CROSS JOIN purchase_orders
ORDER BY customers.cust_id;
+---------+------------+-----------+---------------+
| cust_id | first_name  | last_name | purchase_date |
+---------+------------+-----------+---------------+
|    1000 | John        | Smith     | 2021-12-01    |
|    1000 | John        | Smith     | 2022-12-03    |
|    1000 | John        | Smith     | 2022-05-07    |
|    2000 | Jane        | Jones     | 2021-12-01    |
|    2000 | Jane        | Jones     | 2022-12-03    |
|    2000 | Jane        | Jones     | 2022-05-07    |
+---------+------------+-----------+---------------+
6 rows in set (0.000 sec)
```

MySQL NATURAL JOIN Statement

A MySQL NATURAL JOIN is a join that performs the same task as an INNER or LEFT JOIN, in which the ON or USING clause refers to all columns that the tables to be joined have in common. The MySQL NATURAL JOIN is structured in such a way that columns with the same name of associate tables will appear once only.

Here are some guidelines for a NATURAL JOIN between two tables:

- The associated tables have one or more pairs of identically named columns.
- The columns must be the same data type.
- Do not use an ON clause in a NATURAL JOIN.

AN INNER JOIN TO DELETE DUPLICATE ATTRIBUTES

Two rows are duplicates if they contain the same data values, with the possible exception of an auto-incrementing primary key. For this section, let's say that two rows are *similar* if they have the same value in one (or more) attributes of a given table. Hence, the notion of duplicate rows is a special case of similar rows.

Listing 3.1 shows the content of `weather.sql` that creates and populates the table `weather` with fictitious data values.

LISTING 3.1: weather.sql

```
DROP TABLE IF EXISTS weather;

CREATE TABLE weather (day DATE, temper INTEGER, wind INTEGER,
forecast CHAR(20), city CHAR(20), state CHAR(20));

INSERT INTO weather VALUES('2021-04-01',42, 16,  'Rain', 'sf', 'ca');
INSERT INTO weather VALUES('2021-04-02',45, 3,   'Sunny','sf', 'ca');
INSERT INTO weather VALUES('2021-04-03',78, -12,  NULL,  'se', 'wa');
INSERT INTO weather VALUES('2021-07-01',42, 16,  'Rain', '',   'ca');
INSERT INTO weather VALUES('2021-07-02',45, -3,  'Sunny','sf', 'ca');
INSERT INTO weather VALUES('2021-07-03',78, 12,   NULL,  'sf', 'mn');
INSERT INTO weather VALUES('2021-08-04',50, 12,  'Snow', '',   'mn');
INSERT INTO weather VALUES('2021-08-06',51, 32,  '',     'sf', 'ca');
INSERT INTO weather VALUES('2021-09-01',42, 16,  'Rain', 'sf', 'ca');
INSERT INTO weather VALUES('2021-09-02',45, 99,  '',     'sf', 'ca');
INSERT INTO weather VALUES('2021-09-03',15, 12,  'Snow', 'chi','il');
```

Listing 3.1 starts with a `DROP` statement, then a `CREATE` statement, and then a set of `INSERT` statements. Log into MySQL and execute the following statements:

```
use mytools;
source weather.sql;
select * from weather;
```

The output from the preceding code snippet is the following (or something similar):

```
select * from weather;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
11 rows in set (0.000 sec)
```

To ensure that we can refresh the contents of the weather table with the preceding rows of data, let's create the table weather2 as a copy of the weather table with the following SQL statements:

```
DROP TABLE IF EXISTS weather2;
CREATE TABLE weather2 AS (SELECT * FROM weather);
```

We can delete similar rows from the weather2 table that have duplicate city values with the following SQL statement:

```
DELETE w1
FROM weather2 w1
INNER JOIN weather2 w2
WHERE
    w1.day < w2.day AND
    w1.city = w2.city;
Query OK, 7 rows affected (0.003 sec)
```

Let's look at the updated contents of the weather2 table to compare with the contents of the weather table:

```
SELECT *
FROM weather2;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
4 rows in set (0.000 sec)
```

If you look closely at the contents of the city attribute in the weather table, you can see that the rows containing the first five occurrences of the sf value in the city attribute have been deleted, as well as the row with the first occurrence of an empty string for the city attribute.

Hence, the SQL statement that deleted similar rows always deletes the all-but-last occurrence of a subset of rows that have the same value in an attribute of a table.

JOIN STATEMENTS ON TABLES WITH INTERNATIONAL TEXT

Earlier you saw how to create the table japn1 that contains Japanese text. Now let's create the table japn2 whose text is the English counterpart to the text in japn1, and then perform a join on the tables japn1 and japn2.

Listing 3.2 shows the content of japanese2.sql that creates and populates the table japn2, and Listing 3.3 shows the content of japn_join.sql that illustrates how to join the tables japn1 and japn2.

LISTING 3.2: japanese2.sql

```
use mytools;
DROP TABLE IF EXISTS japn2;

CREATE TABLE japn2
(
    emp_id INT NOT NULL AUTO_INCREMENT,
    fname VARCHAR(100) NOT NULL,
    lname VARCHAR(100) NOT NULL,
    title VARCHAR(100) NOT NULL,
    PRIMARY KEY (emp_id)
);

INSERT INTO japn2 SET fname="hideki",  lname="hiura", title="CTO";
INSERT INTO japn2 SET fname="momotaro",lname="strong",title="manager";
INSERT INTO japn2 SET fname="oswald",  lname="camp",title="funloving";
INSERT INTO japn2 SET fname="tokyo",   lname="japan",title="awesome!";
```

As you can see, Listing 3.2 contains familiar SQL statements. Let's look at the content of `japn_join.sql` that performs a join on the two tables.

LISTING 3.3: japn_join.sql

```
use mytools;

SELECT j1.emp_id,j1.fname,j1.lname,j2.fname,j2.lname
FROM japn1 j1, japn2 j2
WHERE j1.emp_id = j2.emp_id;
```

Launch the code in Listing 3.3 to see the following output:

```
+--------+----------------+-----------+----------+--------+
| emp_id | fname          | lname     | fname    | lname  |
+--------+----------------+-----------+----------+--------+
|      1 | ひでき         | 日浦      | hideki   | hiura  |
|      2 | ももたろ       | つよい    | momotaro | strong |
|      3 | オズワルド     | カmポ     | oswald   | camp   |
|      4 | 東京           | 日本      | tokyo    | japan  |
+--------+----------------+-----------+----------+--------+
4 rows in set (0.001 sec)
```

Now that you know how to perform a join on two MySQL tables, you are ready to learn about creating views in SQL, which is the topic of the next section.

WHAT IS A VIEW?

A *view* provides a mechanism for restricting access to the data in a table (or tables) such that a subset (specified in the definition of the view) of the rows are accessible to users. A view can specify data from a subset of rows, a subset of columns, or both, which can be accessed from a single table or multiple tables.

A view is sometimes called a *virtual table* because a view does *not* store a copy of the data that is retrieved by the view definition: a view only returns a result set. Note that a view is stored in the database, and you can reference a view in the same way that you reference a table in a SQL statement. In addition, the name of a view must be distinct from the name of all tables and all views in a database.

Creating a View

Use the clause `CREATE VIEW` to create a view, along with the `AS` keyword (discussed in Chapter 1) and the `SELECT` keyword, where the latter specifies the data that is visible through the view. As a simple example, here is the definition of the view `V1` that accesses all the data in the `customers` table:

```
CREATE VIEW V1 AS (SELECT * FROM customers);
```

The preceding example illustrates the syntax for creating a view: in general, a view will access a subset of the rows and/or columns of a table, as you will see later in this chapter.

For example, you can easily allow access only to the rows in the `customers` table whose `cust_id` equals 1000. As another example, you can define a view over a `JOIN` of the `customers` table, the `purchase_orders` table, and the `line_items` table to display the purchase order details belonging to a customer.

Dropping a View in SQL

You can drop a view in SQL using the same syntax that you use for dropping a table, as shown here:

```
DROP VIEW view_name;
DROP VIEW view1, view2, view3;
```

If you do not know whether or not the view already exists, you can use the following syntax:

```
DROP VIEW IF EXISTS view_name;
DROP VIEW IF EXISTS view1, view2, view3;
```

The preceding syntax is useful when you invoke a SQL script that modifies the definition of an existing view. Incidentally, you can also include such statements in a SQL file.

Advantages of Views in SQL Statements

Now that you understand what views are and how to create them, here is a list of some advantages of a view over SQL statements that directly access data from one or more tables:

- restricted access to data (i.e., security)
- simpler queries
- abstraction of business logic

Earlier, we briefly mentioned restricted access. For example, you might want to prevent users from accessing sensitive information, such as a table attribute that contains social security numbers.

A view can be a replacement for a SQL statement that involves a multi-table join or a subquery. In fact, a view definition can consist of a combination of tables and other view definitions. Finally, a view can "abstract" away complex business logic in transactions, thereby reducing the likelihood of creating and executing incorrect SQL queries to retrieve the desired data. In a sense, SQL views can function as an access layer between users and database tables.

Views Involving a Single Table

The following SQL statement defines a view over the `customers` table:

```
MySQL [mytools]> CREATE VIEW V1 AS (SELECT * FROM customers);
1 row in set (0.002 sec)
```

Note that the data that is visible via view `V1` is identical to the data that is visible from the `customers` table, so view `V1` does not provide any significant advantages. If you select the data rows from view `V1`, you will see the same set of rows that are selected from the `customers` table.

The interesting aspect of `V1` is that it's possible to insert data rows into `V1` as well. However, this is not true in general; specifically, you cannot insert a data row into a view that is defined as a join of two or more tables if there is *any* ambiguity regarding the exact set of attributes in the underlying tables that are affected.

Now examine the definition of view `V2` that selects all the rows of `customers` (i.e., the same as view `V1`), but *only* the attributes `cust_id` and `city`, as defined here:

```
MySQL [mytools]>
CREATE VIEW V2 AS (SELECT cust_id,city FROM customers);
1 row in set (0.002 sec)
+------------------+
| Tables_in_mytools |
```

Views Involving Multiple Tables

Here is an example of creating the view `V3`, which is defined over a join of two tables:

```
MySQL [mytools]>
CREATE VIEW V3 AS
(SELECT cust_id,po_id FROM customers c, purchase_orders p
Where c.cust_id = p.cust_id );
```

If you select the data from the view `V3`, you will see a set of rows contain the field `cust_id` and the field `po_id`. Note that the returned data set does not necessarily "group" the rows so that all the `purchase_orders` for each customer are in the same block. In order to perform the latter, use `GROUP BY c.cust_id` in the definition of the view `V3` (discussed later).

Updatable Views

An *updatable view* refers to a view in which the underlying table or tables can be updated. If a view is based on a single table, then the view is updatable if the underlying table is updatable. If a view is based on two or more tables, then the view is updatable if there is no ambiguity regarding the exact rows and tables that are affected by the update.

Moreover, an updatable view (regardless of how it's defined) cannot contain any of the following SQL clauses:

- Aggregate functions
- Group operators
- GROUP BY expressions
- JOINs
- Set operators

KEYS, PRIMARY KEYS, AND FOREIGN KEYS

A *key* is a value used to identify a record in a table uniquely. A key could be a single column or combination of multiple columns. The columns in a table that are *not* used to identify a record uniquely are called *non-key columns*. A *unique key* constraint ensures that there are no duplicate rows in a table (i.e., all rows are unique).

A *primary key* is a combination of fields that uniquely identifies a row in a table. Keep in mind that a primary key *cannot* have `NULL` values: in fact, there is an implicit `NOT NULL` constraint on a primary key. Consequently, a primary key constraint also has a unique constraint. The crucial point to remember is this: you can define multiple unique constraint on a given table, but there can be only one *primary* key on a table.

A *foreign key* references the primary key of another table. A foreign key exists when a pair of tables have a master/detail relationship, such as a purchase order and its line items, or a student and the list of courses in which the student has enrolled.

Foreign Keys versus Primary Keys

Suppose that `A` and `B` are two MySQL tables. As you saw in the previous section, a *foreign key* from `A` to `B` provides a mechanism for a row in table `A` to reference a related row in table `B`. In addition, a foreign key is specified in the definition of a primary key for a given table. A foreign key has the following properties:

- can have a different name from its primary key
- ensures rows in one table have corresponding rows in another
- is not required to be unique (often foreign keys are not).
- can be null, even though a primary key cannot

On the other hand, a *primary key* is used to identify a database record uniquely, and it has the following properties:

- cannot be NULL
- must be unique
- its values should rarely be changed
- must be given a value when a new record is inserted

Lastly, a *composite key* is a key composed of two or more columns used to identify a record uniquely.

A MYSQL EXAMPLE OF FOREIGN KEYS

Although the definitions of primary keys and foreign keys in the previous section are straightforward, there are some subtle points to keep in mind when you specify them in table definitions.

Listing 3.4 shows the content of `parent_child.sql` that illustrates how to specify a primary key in the tables `parent_tbl` and `child_tbl` and also a foreign key in the table `child_tbl`. Both of these tables have minimal content so that you can focus on the foreign key portion of the code, and also easily observe what happens when you experiment with the table definitions.

LISTING 3.4: parent_child.sql

```
use mytools;

DROP TABLE IF EXISTS parent_tbl;
DROP TABLE IF EXISTS child_tbl;
-- drop parent_tbl again: why?
DROP TABLE IF EXISTS parent_tbl;

CREATE TABLE parent_tbl(
  cust_id INT PRIMARY KEY,
  cust_name VARCHAR(30)
);

CREATE TABLE child_tbl(
    child_id INT PRIMARY KEY,
    cust_id INT,
    FOREIGN KEY (cust_id) REFERENCES parent_tbl(cust_id)
    ON DELETE SET NULL
    ON UPDATE SET NULL
);

INSERT INTO parent_tbl VALUES (100,'John Smith');
INSERT INTO parent_tbl VALUES (200,'Jane Jones');

INSERT INTO child_tbl VALUES (1200, 100);
INSERT INTO child_tbl VALUES (1300, 100);
INSERT INTO child_tbl VALUES (2500, 200);
INSERT INTO child_tbl VALUES (2600, 200);
```

Listing 3.4 starts by dropping the `parent_tbl` and `child_tbl` tables, followed by the creation of these two tables, and then several SQL statements that insert data into both tables. Now launch the SQL file to see the following output:

```
Database changed
ERROR 3730 (HY000) at line 2 in file: 'parent_child.sql':
Cannot drop table 'parent_tbl' referenced by a foreign key
constraint 'child_tbl_ibfk_1' on table 'child_tbl'.
Query OK, 0 rows affected (0.014 sec)
Query OK, 0 rows affected (0.005 sec)
Query OK, 0 rows affected (0.007 sec)
Query OK, 0 rows affected (0.009 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
```

The error message `ERROR 3730 (HY000)` occurs because the first attempt to drop the table `parent_tbl` fails due to the foreign key constraint that is specified in the definition of the table `child_tbl`.

Second, after the `child_tbl` has been recreated, it's possible to drop and re-create the table `parent_tbl`, which is why the `parent_tbl` is dropped (successfully) in the *second* `DROP TABLE` statement.

Third, do *not* specify `NOT NULL` in the definition for `cust_id` or you will see this error message:

```
--Column 'cust_id' cannot be NOT NULL: needed in a foreign
key constraint 'child_tbl_ibfk_1' SET NULL
```

Finally, keep in mind that other RDBMSs might exhibit slightly different behavior, in which case you need to modify the SQL statements accordingly.

Here are the contents of the tables `parent_tbl` and `child_tbl` after you execute the code in Listing 3.4:

```
MySQL [mytools]> select * from parent_tbl;
+---------+------------+
| cust_id | cust_name  |
+---------+------------+
|     100 | John Smith |
|     200 | Jane Jones |
+---------+------------+
2 rows in set (0.001 sec)

MySQL [mytools]> select * from child_tbl;
+----------+---------+
| child_id | cust_id |
+----------+---------+
|     1200 |     100 |
|     1300 |     100 |
```

```
|     2500 |     200 |
|     2600 |     200 |
+----------+---------+
4 rows in set (0.001 sec)
```

WORKING WITH SUBQUERIES IN SQL

A *subquery* is a SQL query that is defined inside another SQL query. The subquery is also called a *nested query* or *inner query*. In addition, there are two types of subqueries:

- correlated subqueries (same table in inner and outer query)
- non-correlated subqueries (can return Boolean values)

One type of subquery can return a result set with zero, one, or multiple rows; by contrast, another type of subquery can return a Boolean value. The next section provides additional details regarding both types of subqueries.

Two Types of Subqueries

As you learned in the previous section, a *correlated subquery* is a subquery that contains a reference to a table that also appears in the outer query. Therefore, a correlated subquery is *not* an independent query. Correlated subqueries execute the *outer* query *before* the inner query is executed. Correlated subqueries contain SQL clauses such as `EXIST`, `NOT EXIST`, `IN`, and `NOT IN`.

By contrast, a *non-correlated subquery* is an independent query whose output is substituted into the main query. The *inner* query is always executed *before* the outer query in a non-correlated subquery.

In both types of subqueries, the nested query returns a (possibly empty) set of values that is then processed by the outermost SQL query. Consequently, there can be a performance penalty because the subquery may be evaluated again for *each* row processed by the outer query, thereby increasing the execution time for the entire SQL statement.

Now that you understand the different types of subqueries, the following SQL statement selects the `day` and the `forecast` for the day (or days) that have the maximum temperature by means of a correlated subquery:

```
SELECT day, forecast
FROM weather
WHERE temper IN (
  SELECT max(temper) FROM weather);
```

As you can see in the preceding SQL statement, a subquery enables you to restrict the data that is queried by the main query. Although we have not discussed the `MAX()` function in SQL, the purpose of this function is intuitive: it returns the maximum value of an attribute of a table. Similarly, `MIN()` and `AVG()` return the minimum and average values, respectively, of an attribute of a table. These aggregate functions are discussed again in Chapter 4.

As a second example, the following SQL statement displays the city and state in which the temperature is above the average temperature for all cities, where the average temperature is determined by a correlated subquery:

```
SELECT city, state
FROM weather
WHERE temper > (SELECT AVG(temper)
                FROM weather);
+------+-------+
| city | state |
+------+-------+
| se   | wa    |
| sf   | mn    |
|      | mn    |
| sf   | ca    |
+------+-------+
4 rows in set (0.008 sec)
```

Obviously, you can replace ">" in the preceding SQL statement with "<" or "=" or other inequalities to retrieve the appropriate subset of rows from the weather table.

As a third example, the following SQL statement contains a correlated subquery that appears in the SELECT clause of the outer query, which prints the entire list of cities and states alongside the average temperature for each city:

```
SELECT city, state, temper,
  (SELECT AVG(temper)
   FROM weather
   WHERE city = w.city) AS city_average
   FROM weather w
   ORDER BY city, state;
+------+-------+--------+--------------+
| city | state | temper | city_average |
+------+-------+--------+--------------+
|      | ca    |     42 |      46.0000 |
|      | mn    |     50 |      46.0000 |
| chi  | il    |     15 |      15.0000 |
| se   | wa    |     78 |      78.0000 |
| sf   | ca    |     42 |      49.7143 |
| sf   | ca    |     45 |      49.7143 |
| sf   | ca    |     45 |      49.7143 |
| sf   | ca    |     51 |      49.7143 |
| sf   | ca    |     42 |      49.7143 |
| sf   | ca    |     45 |      49.7143 |
| sf   | mn    |     78 |      49.7143 |
+------+-------+--------+--------------+
11 rows in set (0.000 sec)
```

A Subquery to Find Customers Without Purchase Orders

Subqueries are useful for finding rows in a table that do not have any rows in a related table. Let's look at the contents of the customers2 table that is

defined in the SQL script `customers2.sql`, which has the same structure as the `customers` table, and also contains six rows of data, as shown below:

```
SELECT DISTINCT(cust_id), first_name, last_name
FROM customers2;
+---------+------------+-----------+
| cust_id | first_name | last_name |
+---------+------------+-----------+
|    1000 | John       | Smith     |
|    2000 | Jane       | Jones     |
|    3000 | Sara       | Smith     |
|    4000 | Dave       | Dean      |
|    5000 | Kenn       | Knuth     |
+---------+------------+-----------+
5 rows in set (0.001 sec)
```

In addition, let's review the contents of two of the attributes of the `purchase_orders` table, as shown below:

```
SELECT DISTINCT(po_id), cust_id
FROM purchase_orders;
+-------+---------+
| po_id | cust_id |
+-------+---------+
| 12500 |    1000 |
| 12600 |    1000 |
| 12700 |    1000 |
+-------+---------+
3 rows in set (0.000 sec)
```

As you can see, *only* the customer with `cust_id` equal to 1000 has made any purchase orders. How do we display information about the customers who have not made any purchase orders? The following SQL statement accomplishes this task:

```
SELECT c.cust_id, c.first_name, c.last_name
FROM customers2 c
WHERE
  NOT EXISTS (
    SELECT po.cust_id
    FROM purchase_orders po
    WHERE po.cust_id = c.cust_id
  );
+---------+------------+-----------+
| cust_id | first_name | last_name |
+---------+------------+-----------+
|    2000 | Jane       | Jones     |
|    3000 | Sara       | Smith     |
|    4000 | Dave       | Dean      |
|    5000 | Kenn       | Knuth     |
+---------+------------+-----------+
4 rows in set (0.001 sec)
```

If you want to determine the *number* of customers who have not made any purchase orders, use the code snippet `SELECT count(*)` instead of this code snippet:

```
SELECT c.cust_id, c.first_name, c.last_name
```

Now that you know how to use the `ORDER BY` and `HAVING` clause, let's see how to display customers without purchase orders *and* specify criteria such as grouping by `zip_code` and by `state`. The answer is in the following SQL statement:

```
SELECT c.cust_id, c.first_name, c.last_name, c.zip_code, c.state
FROM customers2 c
WHERE
  NOT EXISTS (
    SELECT po.cust_id
    FROM purchase_orders po
    WHERE po.cust_id = c.cust_id
  )
ORDER BY zip_code, state;
+---------+------------+-----------+----------+-------+
| cust_id | first_name | last_name | zip_code | state |
+---------+------------+-----------+----------+-------+
|    4000 | Dave       | Dean      | 67123    | IL    |
|    5000 | Kenn       | Knuth     | 67345    | IL    |
|    2000 | Jane       | Jones     | 95015    | CA    |
|    3000 | Sara       | Smith     | 95043    | CA    |
+---------+------------+-----------+----------+-------+
4 rows in set (0.000 sec)
```

SUBQUERIES WITH IN AND NOT IN CLAUSE

Before working with these SQL clauses, let's review the contents of the `weather` table:

```
select * from weather;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
11 rows in set (0.000 sec)
```

Suppose that we want to retrieve the rows where the `state` can be either `ca`, `wa`, or `il`. One way to do so involves using multiple `OR` clauses in a SQL query. However, a simpler solution involves the **`IN`** keyword, as shown here:

```
SELECT * FROM weather
WHERE state IN ('ca','wa','il');
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
9 rows in set (0.003 sec)
```

Similarly, we can find the rows whose `state` is not in the preceding list by using the **`NOT IN`** keyword, as shown here:

```
SELECT * FROM weather
WHERE state NOT IN ('ca','wa','il');
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
+------------+--------+------+----------+------+-------+
2 rows in set (0.000 sec)
```

The results of the preceding two queries make sense: the preceding **`NOT IN`** query returns 2 rows, and the **`IN`** query returns 9 rows, and their sum is 11, which equals the number of rows in the `weather` table.

SUBQUERIES WITH SOME, ALL, ANY CLAUSE

Before working with any of these SQL clauses, let's first look at the data in the `friends` table:

```
select * from friends;
+-----+-------+-------+--------+
| id  | fname | lname | height |
+-----+-------+-------+--------+
| 100 | Jane  | Jones |    170 |
| 200 | Dave  | Smith |    160 |
| 300 | Jack  | Stone |    180 |
+-----+-------+-------+--------+
3 rows in set (0.000 sec)
```

The **ALL** keyword in the next SQL statement selects the `lname` and `fname` attributes of the rows whose `id` attribute is greater than all of the `id` values in the inner `SELECT` statement:

```
SELECT id, lname, fname
FROM friends
WHERE id > ALL(SELECT 100);
+-----+-------+-------+
| id  | lname | fname |
+-----+-------+-------+
| 200 | Smith | Dave  |
| 300 | Stone | Jack  |
+-----+-------+-------+
2 rows in set (0.009 sec)
```

As another example, the `ALL` keyword selects the `lname` and `fname` attributes of the rows whose `id` attribute is greater than all of the `id` values in the `friends` table. Obviously, there are no such rows, as shown here:

```
SELECT id, lname, fname
FROM friends
WHERE id > ALL(
SELECT id FROM friends);
Empty set (0.001 sec)
```

The **SOME** keyword selects the `lname` and `fname` of the rows whose `id` value is greater than *some* (at least one will suffice) of the `id` values in the rows returned by the inner query, an example of which is here:

```
SELECT id, lname, fname
FROM friends
WHERE id > SOME(
SELECT id FROM friends);
+-----+-------+-------+
| id  | lname | fname |
+-----+-------+-------+
| 200 | Smith | Dave  |
| 300 | Stone | Jack  |
+-----+-------+-------+
2 rows in set (0.002 sec)
```

As you can confirm, 200 and 300 are both greater than 100, but 100 is not greater than any of the `id` values in the `friends` table.

The **ANY** keyword selects the `lname` and `fname` of the rows whose `id` value is greater than *any* (at least one) of the `id` values in the rows returned by the inner query, an example of which is here:

```
SELECT id, lname, fname
FROM friends
WHERE id > ANY(
SELECT id FROM friends);
```

```
+-----+-------+-------+
| id  | lname | fname |
+-----+-------+-------+
| 200 | Smith | Dave  |
| 300 | Stone | Jack  |
+-----+-------+-------+
2 rows in set (0.000 sec)
```

Notice that although the result of the `SOME` query and the `ANY` query are the same in the preceding SQL statements, in general, the result sets are different.

SUBQUERIES WITH THE MAX() AND AVG() FUNCTIONS

The following SQL statement returns two rows from the `weather` table where the temperature *equals* the maximum value:

```
SELECT temper, city, state
FROM weather
WHERE temper = (SELECT MAX(temper) FROM weather);
+--------+------+-------+
| temper | city | state |
+--------+------+-------+
|     78 | se   | wa    |
|     78 | sf   | mn    |
+--------+------+-------+
2 rows in set (0.001 sec)
```

The following SQL statement returns seven rows from the `weather` table where the temperature is less than the average value:

```
SELECT temper, city, state
FROM weather
WHERE temper < (SELECT AVG(temper) FROM weather);
+--------+------+-------+
| temper | city | state |
+--------+------+-------+
|     42 | sf   | ca    |
|     45 | sf   | ca    |
|     42 |      | ca    |
|     45 | sf   | ca    |
|     42 | sf   | ca    |
|     45 | sf   | ca    |
|     15 | chi  | il    |
+--------+------+-------+
7 rows in set (0.005 sec)
```

FIND TALLEST STUDENTS IN EACH CLASSROOM VIA A SUBQUERY

Listing 3.5 shows the content of `heights.sql` that creates the table `heights` and then selects the tallest three students in each of the distinct classrooms (i.e., 1000 and 2000) specified in the `room` attribute of the `heights` table.

LISTING 3.5: heights.sql

```
use mytools;

DROP TABLE IF EXISTS heights;
CREATE TABLE heights(id INTEGER, name CHAR(10), height
INTEGER, room INTEGER);

INSERT INTO heights VALUES( 1,'person1', 150, 1000);
INSERT INTO heights VALUES( 2,'person2', 180, 1000);
INSERT INTO heights VALUES( 3,'person3', 200, 1000);
INSERT INTO heights VALUES( 4,'person4', 100, 1000);
INSERT INTO heights VALUES( 5,'person5', 130, 2000);
INSERT INTO heights VALUES( 6,'person6', 100, 2000);
INSERT INTO heights VALUES( 7,'person7', 110, 2000);
INSERT INTO heights VALUES( 8,'person8', 120, 2000);

SELECT *
FROM heights h1
WHERE 3 > (
           SELECT COUNT(DISTINCT height)
           FROM heights h2
           WHERE h2.height > h1.height
           AND h1.room = h2.room
          );
```

Listing 3.5 starts by creating and populating the table `heights` with 8 rows of data, where 4 students are in classroom 1000 and 4 students are in classroom 2000.

The second portion of Listing 3.5 defines a SQL statement that is a correlated subquery. Notice that the `WHERE` clause specifies the value 3 in order to limit the number of rows that are returned by the inner SQL statement.

In Chapter 4, you will learn how to use the `LIMIT` keyword that limits the number of rows that are returned by a SQL statement. However, MySQL 8 does not support the `LIMIT` keyword inside a subquery. Launch the code in Listing 3.5 to see the following output:

```
+------+---------+--------+------+
| id   | name    | height | room |
+------+---------+--------+------+
|    1 | person1 |    150 | 1000 |
|    2 | person2 |    180 | 1000 |
|    3 | person3 |    200 | 1000 |
|    5 | person5 |    130 | 2000 |
|    7 | person7 |    110 | 2000 |
|    8 | person8 |    120 | 2000 |
+------+---------+--------+------+
6 rows in set (0.000 sec)
```

The SQL statement returns 3 rows for room 1000 and 3 rows from room 2000, all of which are the top three tallest students in their respective classrooms. The nice aspect of Listing 3.5 is that you can generalize the result by adding an arbitrary number of departments, or by changing the number of

values that you want from each classroom, or both. Try replacing the number 3 by 1, 2, 4, 5, or any other positive integer and verify that the output of the modified SQL statement is correct.

SQL AND HISTOGRAMS

A *histogram* in SQL refers to a SQL statement that displays the distribution (i.e., frequency) of items in a database table. For example, we can display the contents of the `item_desc` table as follows:

```
select *
from item_desc;
+---------+-------------+------------+
| item_id | item_desc   | item_price |
+---------+-------------+------------+
|     100 | hammer      |      20.00 |
|     200 | screwdriver |       8.00 |
|     300 | wrench      |      10.00 |
+---------+-------------+------------+
3 rows in set (0.001 sec)
```

We display only the values of the `item_price` attribute in the `item_desc` table as follows:

```
select item_price
from item_desc;
+------------+
| item_price |
+------------+
|      20.00 |
|       8.00 |
|      10.00 |
+------------+
3 rows in set (0.000 sec)
```

The next portion of this chapter contains examples of SQL statements that specify each of the clause `ORDER BY`, `GROUP BY`, and `HAVING`, followed by examples that use a combination of these SQL clauses. For simplicity, the SQL queries in the upcoming sections are based on a single table; however, you can generate more sophisticated reports that contain `JOIN` clauses that involve multiple tables.

WHAT ARE GROUP BY, ORDER BY, AND HAVING CLAUSES?

The `GROUP BY` clause enables you to count items that are "grouped together" based on the same attribute value. For example, the following SQL statement counts the number of occurrences of the same `city` value in the `weather` table:

```
SELECT city, COUNT(city)
FROM weather
```

```
GROUP BY city;
+------+-------------+
| city | count(city) |
+------+-------------+
| sf   |           7 |
| se   |           1 |
|      |           2 |
| chi  |           1 |
+------+-------------+
4 rows in set (0.003 sec)
```

The ORDER BY clause enables you to specify the order in which items are displayed. For example, the following SQL statement counts the number of occurrences of the same city name in the weather table and also orders the output alphabetically by city name:

```
SELECT city, COUNT(city)
FROM weather
GROUP BY city
ORDER BY city;
+------+-------------+
| city | count(city) |
+------+-------------+
|      |           2 |
| chi  |           1 |
| se   |           1 |
| sf   |           7 |
+------+-------------+
4 rows in set (0.003 sec)
```

The HAVING clause enables you to specify an additional filter condition for the result set. For example, the following SQL statement extends the previous SQL statement by restricting the result set to cities whose count is greater than 2 in the weather table:

```
SELECT city, COUNT(city)
FROM weather
GROUP BY city
HAVING count(*) > 2
ORDER BY city;
+------+-------------+
| city | count(city) |
+------+-------------+
| sf   |           7 |
+------+-------------+
1 row in set (0.003 sec)
```

A HAVING clause and a WHERE clause *both* filter the data in a result set, but there is a difference. HAVING applies *only* to groups of data, whereas the WHERE clause applies to *individual* rows.

The HAVING clause is executed *before* the SELECT statement, which means that you *cannot* use aliases of aggregated columns in the HAVING clause.

Displaying Duplicate Attribute Values

In a previous section, you learned how to delete duplicate rows, where two rows are considered duplicates if they have the same attribute value. The following SQL statement uses the `HAVING` keyword to display the number of duplicate rows (i.e., rows that contain the same attribute value) in a MySQL table:

```
SELECT city, COUNT(*)
FROM weather
GROUP BY city
HAVING COUNT(*) > 1
+------+----------+
| city | COUNT(*) |
+------+----------+
| sf   |        7 |
|      |        2 |
+------+----------+
2 rows in set (0.006 sec)
```

By contrast, recall that Chapter 2 contains a SQL statement with a subquery to find the *unique* rows in a database table.

EXAMPLES OF THE SQL GROUP BY AND ORDER BY CLAUSE

The following SQL statement displays the values of the `item_price` attribute in the `item_desc` table, as well as their frequency:

```
SELECT item_price, COUNT(1) as frequency
FROM item_desc
GROUP BY 1;
+------------+-----------+
| item_price | frequency |
+------------+-----------+
|      20.00 |         1 |
|       8.00 |         1 |
|      10.00 |         1 |
+------------+-----------+
3 rows in set (0.001 sec)
```

The following SQL statement displays the values of the `item_price` attribute in the `item_desc` table, the frequency of those values, and orders them in decreasing order:

```
SELECT item_price, COUNT(1) as frequency
FROM item_desc
GROUP BY 1
ORDER BY item_price;
+------------+-----------+
| item_price | frequency |
+------------+-----------+
|       8.00 |         1 |
|      10.00 |         1 |
|      20.00 |         1 |
+------------+-----------+
3 rows in set (0.001 sec)
```

Yet another example of a SQL statement with the `ORDER BY` clause is shown here:

```
SELECT item_desc, item_price
FROM new_items
ORDER BY item_price DESC;
+----------------+------------+
| item_desc      | item_price |
+----------------+------------+
| Toolbox L      |      50.00 |
| Toolbox M      |      40.00 |
| Toolbox S      |      30.00 |
| hammer         |      20.00 |
| ballpeen       |      20.00 |
| Handsaw        |      20.00 |
| wrench         |      10.00 |
| pliers         |      10.00 |
| screwdriver    |       8.00 |
| 1/4 inch nails |       8.00 |
+----------------+------------+
10 rows in set (0.000 sec)
```

SQL Histograms on a Table Copy

Suppose that we want to experiment with generating histograms, but without modifying the contents of the `item_desc` table. One approach involves the following steps:

- dynamically create a table `new_items` with the structure of the `item_desc` table
- populate `new_items` with the contents of the `item_desc` table
- insert new rows into the `new_items` table

The first two bullet items are handled by the following SQL statement:

```
CREATE TABLE new_items AS (SELECT * FROM item_desc);
Query OK, 3 rows affected (0.023 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Verify that `new_items` contains the same data as `item_desc` via the following SQL statement:

```
MySQL [mytools]> SELECT * FROM new_items;
+---------+-------------+------------+
| item_id | item_desc   | item_price |
+---------+-------------+------------+
|     100 | hammer      |      20.00 |
|     200 | screwdriver |       8.00 |
|     300 | wrench      |      10.00 |
+---------+-------------+------------+
3 rows in set (0.001 sec)
```

Insert new rows into the `new_items` table by executing the following SQL statements:

```
MySQL [mytools]> INSERT INTO new_items VALUES(400,'pliers',10.00);
Query OK, 1 row affected (0.004 sec)

MySQL [mytools]> INSERT INTO new_items VALUES(500,'ballpeen',20.00);
Query OK, 1 row affected (0.001 sec)

MySQL [mytools]> INSERT INTO new_items VALUES(600,'1/4 inch nails',8.00);
Query OK, 1 row affected (0.001 sec)

MySQL [mytools]> INSERT INTO new_items VALUES(700,'Toolbox S',30.00);
Query OK, 1 row affected (0.002 sec)

MySQL [mytools]> INSERT INTO new_items VALUES(800,'Toolbox M',40.00);
Query OK, 1 row affected (0.001 sec)

MySQL [mytools]> INSERT INTO new_items VALUES(900,'Toolbox L',50.00);
Query OK, 1 row affected (0.001 sec)

MySQL [mytools]> INSERT INTO new_items VALUES(1000,'Handsaw',20.00);
Query OK, 1 row affected (0.004 sec)
```

Now display the new contents of the `new_items` table with the following SQL statement:

```
SELECT * FROM new_items;
+---------+----------------+------------+
| item_id | item_desc      | item_price |
+---------+----------------+------------+
|     100 | hammer         |      20.00 |
|     200 | screwdriver    |       8.00 |
|     100 | wrench         |      10.00 |
|     400 | pliers         |      10.00 |
|     500 | ballpeen       |      20.00 |
|     600 | 1/4 inch nails |       8.00 |
|     700 | Toolbox S      |      30.00 |
|     800 | Toolbox M      |      40.00 |
|     900 | Toolbox L      |      50.00 |
|    1000 | Handsaw        |      20.00 |
+---------+----------------+------------+
9 rows in set (0.000 sec)
```

We can modify and launch the histogram-based SQL statement. Notice the results:

```
SELECT item_price, COUNT(1) as frequency
FROM new_items
GROUP BY 1
ORDER BY item_price;
+------------+-----------+
| item_price | frequency |
+------------+-----------+
|       8.00 |         2 |
|      10.00 |         2 |
```

```
|      20.00 |         3 |
|      30.00 |         1 |
|      40.00 |         1 |
|      50.00 |         1 |
+------------+-----------+
6 rows in set (0.000 sec)
```

Notice that the preceding SQL statement references the `new_items` table instead of the `item_desc` table. We can obtain a similar result with the following SQL statement:

```
SELECT item_price as frequency, COUNT(item_price)
FROM new_items
GROUP BY item_price
ORDER BY item_price;
+-----------+-------------------+
| frequency | COUNT(item_price) |
+-----------+-------------------+
|      8.00 |                 2 |
|     10.00 |                 2 |
|     20.00 |                 3 |
|     30.00 |                 1 |
|     40.00 |                 1 |
|     50.00 |                 1 |
+-----------+-------------------+
6 rows in set (0.000 sec)
```

However, the following SQL statement does *not* work because `GROUP BY` occurs *after* `ORDER BY`, which is invalid SQL syntax:

```
SELECT item_desc, item_price as frequency
FROM new_items
ORDER BY item_price
GROUP BY item_price;
```

If you attempt to execute the preceding SQL statement you will see the following error:

```
ERROR 1064 (42000): You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server
version for the right syntax to use near 'However, the
following SQL statement does not work:
SELECT item_desc, item_price' at line 1
```

COMBINE GROUP BY AND ROLLUP CLAUSE

The term *rollup* refers to the sum of the quantities in sub-accounts to display the combined total for those subaccounts. The general format for SQL statements that include the `ROLLUP` keyword is as follows:

```
SELECT COL1, SUM(COL2)
FROM table name
GROUP BY COL1 WITH ROLLUP;
```

Let's experiment with the `SUM()` and `AVG()` aggregate functions in conjunction with the `ROLLUP` keyword, as shown in the following set of SQL statements.

```
SELECT id, SUM(height)
FROM friends
GROUP BY id WITH ROLLUP;
+------+-------------+
| id   | SUM(height) |
+------+-------------+
|  100 |         170 |
|  200 |         160 |
|  300 |         180 |
| NULL |         510 |
+------+-------------+
4 rows in set (0.008 sec)

SELECT id, SUM(height)
FROM friends
GROUP BY id;
+-----+-------------+
| id  | SUM(height) |
+-----+-------------+
| 100 |         170 |
| 200 |         160 |
| 300 |         180 |
+-----+-------------+
3 rows in set (0.002 sec)

SELECT id, AVG(height)
FROM friends
GROUP BY id WITH ROLLUP;
+------+-------------+
| id   | AVG(height) |
+------+-------------+
|  100 |    170.0000 |
|  200 |    160.0000 |
|  300 |    180.0000 |
| NULL |    170.0000 |
+------+-------------+
4 rows in set (0.002 sec)
```

Although the following SQL statement is similar to the preceding SQL statements, it's actually *invalid*:

```
SELECT id, height
FROM friends
GROUP BY id WITH ROLLUP;
```

If you attempt to execute the preceding SQL statement, you will see the following error:

```
ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP
BY clause and contains nonaggregated column 'mytools.friends.
height' which is not functionally dependent on columns in GROUP
BY clause; this is incompatible with sql_mode=only_full_group_by
```

The 2021 Olympics Medals and the ROLLUP Keyword

The examples in the previous section involved results that you probably anticipated, whereas the next example might produce results that are more meaningful. Specifically, we'll construct a table that contains the number of gold, silver, and bronze medals that the top five countries won during the 2021 Olympics in Japan.

Listing 3.6 shows the content of `OlympicsJAPN2021.csv` that contains the number of medals earned by the top 15 countries in the 2021 Olympics in Japan.

LISTING 3.6: OlympicsJAPN2021.csv

```
Pos,Country,Gold,Silver,Bronze,Total
1,USA,39,41,33,113
2,China,38,32,18,88
3,Japan,27,14,17,58
4,UK,22,21,22,65
5,ROC,20,28,23,71
6,Aust,17,7,22,46
7,Nether,10,12,14,36
8,France,10,12,11,33
9,Germany,10,11,16,37
10,Italy,10,10,20,40
11,Canada,7,6,11,24
12,Brazil,7,6,8,21
13,NZ,7,6,7,20
14,Cuba,7,3,5,15
15,Hungary,6,7,7,20
```

Listing 3.7 shows the content of `olympics.sql` that drops and recreates the table `olympics`, and then populates the table with the number of medals won by the top 5 countries in Listing 3.6.

LISTING 3.7: olympics.sql

```
DROP TABLE IF EXISTS olympics;

CREATE TABLE olympics (pos INTEGER, country VARCHAR(20),
medal VARCHAR(20), count INTEGER);

INSERT INTO olympics VALUES (1,'USA','gold',   39);
INSERT INTO olympics VALUES (1,'USA','silver', 41);
INSERT INTO olympics VALUES (1,'USA','bronze', 33);

INSERT INTO olympics VALUES (2,'CHINA','gold',   38);
INSERT INTO olympics VALUES (2,'CHINA','silver', 32);
INSERT INTO olympics VALUES (2,'CHINA','bronze', 18);

INSERT INTO olympics VALUES (3,'JAPAN','gold',   27);
INSERT INTO olympics VALUES (3,'JAPAN','silver', 14);
INSERT INTO olympics VALUES (3,'JAPAN','bronze', 17);
```

```
INSERT INTO olympics VALUES (4,'UK','gold',   22);
INSERT INTO olympics VALUES (4,'UK','silver', 21);
INSERT INTO olympics VALUES (4,'UK','bronze', 22);

INSERT INTO olympics VALUES (5,'ROC','gold',   20);
INSERT INTO olympics VALUES (5,'ROC','silver', 28);
INSERT INTO olympics VALUES (5,'ROC','bronze', 23);
```

Execute the following SQL statement that contains the ROLLUP keyword and displays the relative ranking of five countries, a column with the number of medals won by each country, and one row for the total number of medals:

```
SELECT pos, SUM(count)
FROM olympics
GROUP BY pos WITH ROLLUP;
+------+------------+
| pos  | SUM(count) |
+------+------------+
|    1 |        113 |
|    2 |         88 |
|    3 |         58 |
|    4 |         65 |
|    5 |         71 |
| NULL |        395 |
+------+------------+
6 rows in set (0.000 sec)
```

The 2021 Olympics Medals and the RANK Operator

Since we've just finished an example that involves the medals for the 2021 Olympics, we'll use that as a segue to ranking the medal counts using the RANK() operator, as shown in the following SQL statement:

```
SELECT count, medal, country,
RANK() OVER (
   ORDER BY count DESC
) my_rank
FROM olympics
LIMIT 10;
+-------+--------+---------+---------+
| count | medal  | country | my_rank |
+-------+--------+---------+---------+
|    41 | silver | USA     |       1 |
|    39 | gold   | USA     |       2 |
|    38 | gold   | CHINA   |       3 |
|    33 | bronze | USA     |       4 |
|    32 | silver | CHINA   |       5 |
|    28 | silver | ROC     |       6 |
|    27 | gold   | JAPAN   |       7 |
|    23 | bronze | ROC     |       8 |
|    22 | gold   | UK      |       9 |
|    22 | bronze | UK      |       9 |
+-------+--------+---------+---------+
10 rows in set (0.001 sec)
```

The RANK() function is shown in bold in the SQL statement: the expression inside the parentheses specifies which attribute to assign a rank, and in this case, rank the attribute values in descending order. Also notice that the last two rows both have 22 medals, which is why both rows have a rank of 9.

THE PARTITION BY CLAUSE

The PARTITION BY clause examines the distinct values of an attribute (which is specified in the SQL statement) in order to partition the rows of a table into subsets such that all the rows in each subset have the same attribute value.

The PARTITION BY clause requires the over() function and can also specify an optional ORDER BY clause as well as an optional window function, such as the RANK(), LEAD(), LAG(), and DENSE_RANK() functions.

The dense_rank() function assigns a rank to each subset: conceptually, the dense_rank() function is a generalization of a row-based ranking (or sorting) of the rows in a table.

The following SQL statement illustrates how to use the PARTITION BY clause to group countries based on their pos value, and then order the rows in each group based on their count value in the olympics table.

```
SELECT pos, country, medal, count,
   DENSE_RANK() OVER (PARTITION BY pos ORDER BY count DESC) AS RANKING
FROM olympics;
+------+---------+--------+-------+---------+
| pos  | country | medal  | count | RANKING |
+------+---------+--------+-------+---------+
|    1 | USA     | silver |    41 |       1 |
|    1 | USA     | gold   |    39 |       2 |
|    1 | USA     | bronze |    33 |       3 |
|    2 | CHINA   | gold   |    38 |       1 |
|    2 | CHINA   | silver |    32 |       2 |
|    2 | CHINA   | bronze |    18 |       3 |
|    3 | JAPAN   | gold   |    27 |       1 |
|    3 | JAPAN   | bronze |    17 |       2 |
|    3 | JAPAN   | silver |    14 |       3 |
|    4 | UK      | gold   |    22 |       1 |
|    4 | UK      | bronze |    22 |       1 |
|    4 | UK      | silver |    21 |       2 |
|    5 | ROC     | silver |    28 |       1 |
|    5 | ROC     | bronze |    23 |       2 |
|    5 | ROC     | gold   |    20 |       3 |
+------+---------+--------+-------+---------+
15 rows in set (0.000 sec)
```

As you can see in the preceding result set, the rows in each partition are displayed in decreasing order with respect to their count value.

GROUP BY, HAVING, AND ORDER BY CLAUSE

Suppose we want to further restrict the result set from the SQL query from an earlier section (i.e., before the PARTITION BY section) to display only the items whose count is less than 2. We can do so with the following SQL query:

```
SELECT item_price, COUNT(*)
FROM new_items
GROUP BY item_price
HAVING COUNT(*) < 2;
+------------+----------+
| item_price | COUNT(*) |
+------------+----------+
|      30.00 |        1 |
|      40.00 |        1 |
|      50.00 |        1 |
+------------+----------+
3 rows in set (0.001 sec)
```

The HAVING keyword will *not* work in the following SQL statement:

```
SELECT frequency, COUNT(*)
FROM new_items
GROUP BY item_price
ORDER BY item_price
HAVING COUNT(*) < 2;
```

The preceding SQL statement generates the following error message:

```
ERROR 1064 (42000): You have an error in your SQL syntax; check
the manual that corresponds to your MySQL server version for
the right syntax to use near 'HAVING COUNT(*) < 2' at line 5
```

Returning to the earlier SQL statement, we can be even more selective with respect to the subtotals by using the IN keyword, as shown in the following SQL statement:

```
SELECT item_price, COUNT(*)
FROM new_items
GROUP BY item_price
HAVING COUNT(*) IN (1,3);
+------------+----------+
| item_price | COUNT(*) |
+------------+----------+
|      20.00 |        3 |
|      30.00 |        1 |
|      40.00 |        1 |
|      50.00 |        1 |
+------------+----------+
4 rows in set (0.000 sec)
```

We can also modify the preceding SQL statement to exclude a specific `item_price` value, as shown in the following SQL statement:

```
SELECT item_price, COUNT(*)
       FROM new_items
       GROUP BY item_price
       HAVING item_price <> 20.00;
+------------+----------+
| item_price | COUNT(*) |
+------------+----------+
|       8.00 |        2 |
|      10.00 |        2 |
|      30.00 |        1 |
|      40.00 |        1 |
|      50.00 |        1 |
+------------+----------+

5 rows in set (0.004 sec)
```

COMBINED GROUP BY, HAVING, AND ORDER BY CLAUSE

This section shows you the order in which these three clauses must appear in a SQL statement in order to execute them correctly. For example, the following SQL statement is *incorrect*:

```
SELECT COUNT(*)
FROM new_items
GROUP BY item_price
ORDER BY item_price
HAVING COUNT(*) > 1;
ERROR 1064 (42000): You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server
version for the right syntax to use near 'HAVING COUNT(*) > 1'
at line 5
```

By contrast, the following similar SQL statement is *correct*:

```
SELECT COUNT(*)
FROM new_items
GROUP BY item_price
HAVING COUNT(*) > 1
ORDER BY item_price;
+----------+
| COUNT(*) |
+----------+
|        2 |
|        2 |
|        3 |
+----------+
3 rows in set (0.000 sec)
```

The difference in the preceding pair of SQL statements involves the placement of the `ORDER BY` clause, which must appear *after* the `HAVING` keyword in these SQL statements.

Updating the item_desc Table from the new_items Table

Suppose that you want to update the original contents of the item_desc table with the contents of the new_items table. One way to do so is to execute the following SQL statements:

```
— the first SQL statement saves the contents of item_desc:
CREATE TABLE orig_item_desc AS (SELECT * FROM item_desc);

— drop the existing table:
DROP TABLE item_desc;

— create item_desc and populate with data from new_item:
CREATE TABLE item_desc AS (SELECT * FROM new_items);
```

The first of the three preceding SQL statements creates a backup of the original contents of the item_desc table. This part of the code is optional and of limited value for a table with only three rows of data. However, it can be useful if the item_desc table contains hundreds (or thousands) of rows of data.

If you need to restore the original contents, this SQL statement will save you the time and effort to locate a backup of the database (which you undoubtedly have somewhere) to restore the original contents of the item_desc table. By contrast, you can restore the contents of the item_desc table with this SQL statement:

```
RENAME TABLE orig_item_desc TO item_desc;
```

This concludes the portion of the chapter pertaining to executing SQL statements with various clauses. One more observation regarding this section: despite the simple SQL queries for a table that contains only 10 rows of data, we have also seen other techniques that can be useful in your own tasks.

Specifically, we saw how to dynamically create the table new_items to replicate an existing table item_desc, populate new_items with new data rows, and then re-create the original table item_desc with the contents of the new_items table.

A SQL QUERY INVOLVING A FOUR-TABLE JOIN

In Chapter 1, you learned about a fictitious Web application that sells tools online to customers, which involves the following four tables:

```
customers
purchase_orders
line_items
item_desc
```

Suppose that you need a SQL query that generates a full report regarding customer activity. Such a report involves a SQL statement that iterates through

the `customers` table and for each customer, lists the purchase orders of that customer, along with the full details of each line item that belongs to each purchase order. The approach outlined in this section involves iterative refinement of SQL statements, which can be summarized as follows:

- Start with a set of tables with a limited number of rows.
- Create a SQL statement with no join conditions (yields duplicate rows).
- Add `JOIN` clauses to reduce the duplicates in the output.
- Repeat the preceding step until the desired report is generated.

Let's perform the steps in the preceding bullet list to issue a series of SQL statements and iteratively refine the SQL code until we arrive at the correct SQL statement. If you prefer to skip the intermediate steps, construct your solution and compare it with the solution at the end of this section.

The first SQL query lists the desired columns from the various tables, along with repeated rows because the SQL statement does not specify any `JOIN` conditions, as shown below:

```
SELECT c.cust_id, p.po_id, l.item_id, d.item_desc
FROM customers c, purchase_orders p, line_items l, item_desc d;
+---------+-------+---------+-------------+
| cust_id | po_id | item_id | item_desc   |
+---------+-------+---------+-------------+
|    1000 | 12500 |    NULL | hammer      |
|    2000 | 12500 |    NULL | hammer      |
|    1000 | 12600 |    NULL | hammer      |
|    2000 | 12600 |    NULL | hammer      |
|    1000 | 12700 |    NULL | hammer      |
|    2000 | 12700 |    NULL | hammer      |
|    1000 | 12500 |    NULL | screwdriver |
|    2000 | 12500 |    NULL | screwdriver |
|    1000 | 12600 |    NULL | screwdriver |
|    2000 | 12600 |    NULL | screwdriver |
|    1000 | 12700 |    NULL | screwdriver |
|    2000 | 12700 |    NULL | screwdriver |
|    1000 | 12500 |    NULL | wrench      |
|    2000 | 12500 |    NULL | wrench      |
|    1000 | 12600 |    NULL | wrench      |
|    2000 | 12600 |    NULL | wrench      |
|    1000 | 12700 |    NULL | wrench      |
|    2000 | 12700 |    NULL | wrench      |
+---------+-------+---------+-------------+
18 rows in set (0.001 sec)
```

Obviously, we want to remove the repeated rows from the preceding result set, so let's try the following SQL statement that also specifies a matching `cust_id` value for the `customers` table *and* the `purchase_orders` table:

```
SELECT c.cust_id, p.po_id, l.item_id, d.item_desc
FROM customers c, purchase_orders p, line_items l, item_desc d
WHERE c.cust_id = p.cust_id;
+---------+-------+---------+-------------+
| cust_id | po_id | item_id | item_desc   |
+---------+-------+---------+-------------+
|    1000 | 12500 |    NULL | hammer      |
|    1000 | 12600 |    NULL | hammer      |
|    1000 | 12700 |    NULL | hammer      |
|    1000 | 12500 |    NULL | screwdriver |
|    1000 | 12600 |    NULL | screwdriver |
|    1000 | 12700 |    NULL | screwdriver |
|    1000 | 12500 |    NULL | wrench      |
|    1000 | 12600 |    NULL | wrench      |
|    1000 | 12700 |    NULL | wrench      |
+---------+-------+---------+-------------+
9 rows in set (0.003 sec)
```

The preceding SQL query has eliminated some rows and *also corrected the erroneous* `JOIN` clause (shown in bold), but we *still* have duplicate rows.

Let's try a third SQL statement that also joins the `purchase_orders` table *and* the `line_items` table, as shown here:

```
SELECT c.cust_id, p.po_id, l.item_id, d.item_desc
FROM customers c, purchase_orders p, line_items l, item_desc d
WHERE c.cust_id = p.cust_id
AND p.po_id = l.po_id
ORDER BY c.cust_id, p.purchase_date, p.po_id;
+---------+-------+---------+-------------+
| cust_id | po_id | item_id | item_desc   |
+---------+-------+---------+-------------+
|    1000 | 12500 |     100 | hammer      |
|    1000 | 12500 |     200 | hammer      |
|    1000 | 12500 |     300 | hammer      |
|    1000 | 12500 |     100 | screwdriver |
|    1000 | 12500 |     200 | screwdriver |
|    1000 | 12500 |     300 | screwdriver |
|    1000 | 12500 |     100 | wrench      |
|    1000 | 12500 |     200 | wrench      |
|    1000 | 12500 |     300 | wrench      |
+---------+-------+---------+-------------+
9 rows in set (0.001 sec)
```

Although we're closer to the correct SQL query, the preceding output contains duplicate rows from the `item_desc` table, so we need to join the `line_items` table and `item_desc` table, as shown here:

```
SELECT c.cust_id, p.po_id, l.item_id, d.item_desc, d.item_price
FROM customers c, purchase_orders p, line_items l, item_desc d
WHERE c.cust_id = p.cust_id
AND p.po_id = l.po_id
AND l.item_id = d.item_id
ORDER BY c.cust_id, p.purchase_date, p.po_id;
```

```
+---------+-------+---------+-------------+------------+
| cust_id | po_id | item_id | item_desc   | item_price |
+---------+-------+---------+-------------+------------+
|    1000 | 12500 |     100 | hammer      |      20.00 |
|    1000 | 12500 |     200 | screwdriver |       8.00 |
|    1000 | 12500 |     300 | wrench      |      10.00 |
+---------+-------+---------+-------------+------------+
3 rows in set (0.001 sec)
```

Success!

The preceding output displays the desired result. Furthermore, we can refine the SQL query by including the number of items that were purchased for each item in the purchase order, as shown here:

```
SELECT c.cust_id, p.po_id, l.item_id, d.item_desc, d.item_price,l.qty
FROM customers c, purchase_orders p, line_items l, item_desc d
WHERE c.cust_id = p.cust_id
AND p.po_id = l.po_id
AND l.item_id = d.item_id
ORDER BY c.cust_id, p.purchase_date, p.po_id;
+---------+-------+---------+-------------+------------+------+
| cust_id | po_id | item_id | item_desc   | item_price | qty  |
+---------+-------+---------+-------------+------------+------+
|    1000 | 12500 |     100 | hammer      |      20.00 |    1 |
|    1000 | 12500 |     200 | screwdriver |       8.00 |    2 |
|    1000 | 12500 |     300 | wrench      |      10.00 |    3 |
+---------+-------+---------+-------------+------------+------+
3 rows in set (0.003 sec)
```

You can also refine the preceding SQL statement to display purchase orders that contain specific items. For example, the following SQL statement displays all the purchase orders that contain a hammer:

```
SELECT c.cust_id, p.po_id, l.item_id, d.item_desc, d.item_price,l.qty
FROM customers c, purchase_orders p, line_items l, item_desc d
WHERE c.cust_id = p.cust_id
AND p.po_id = l.po_id
AND l.item_id = d.item_id
AND d.item_desc = 'hammer'
ORDER BY c.cust_id, p.purchase_date, p.po_id;
+---------+-------+---------+-----------+------------+------+
| cust_id | po_id | item_id | item_desc | item_price | qty  |
+---------+-------+---------+-----------+------------+------+
|    1000 | 12500 |     100 | hammer    |      20.00 |    1 |
+---------+-------+---------+-----------+------------+------+
1 row in set (0.011 sec)
```

The preceding SQL statement will generate more interesting results sets when the `purchase_orders` table contains multiple customers who have made multiple purchases, each of which generates a row in the `purchase_orders` table. Later in this chapter, you will learn how to select the set of purchase orders that are between a pair of dates.

OPERATIONS WITH DATES IN SQL

As a simple starting point for date-related operations, the following SQL statement illustrates how to use the `NOW()` function to display the current date:

```
SELECT NOW() FROM DUAL;
+---------------------+
| NOW()               |
+---------------------+
| 2021-05-11 22:05:03 |
+---------------------+
1 row in set (0.001 sec)
```

The following SQL statement illustrates how to use the `CURRENT_DATE()` function to display the current date, which does not include the `HH::MM:SS` details:

```
SELECT CURRENT_DATE() FROM DUAL;
+----------------+
| current_date() |
+----------------+
| 2021-07-29     |
+----------------+
1 row in set (0.003 sec)
```

In addition, the `SYSDATE` function in SQL is a function that returns the current date as well as the current time, an example of which is shown here:

```
SELECT sysdate() from dual;
+---------------------+
| sysdate()           |
+---------------------+
| 2021-07-15 10:12:59 |
+---------------------+
1 row in set (0.002 sec)
```

The following SQL query returns the same result as the preceding SQL query:

```
SELECT sysdate() from dual;
```

The following SQL statement displays your time zone with an offset from GMT:

```
SELECT TIMEDIFF(NOW(), UTC_TIMESTAMP);
+--------------------------------+
| TIMEDIFF(NOW(), UTC_TIMESTAMP) |
+--------------------------------+
| -07:00:00                      |
+--------------------------------+
1 row in set (0.005 sec)
```

Day and Month Components of Dates in SQL

SQL provides day-related and month-related functions, some of which are listed here:

- MONTHS_BETWEEN
- ADD_MONTHS
- NEXT_DAY
- LAST_DAY
- DAY
- DAYOFMONTH

In fact, SQL supports dozens of date formats, along with functions that enable you to select different elements in date fields. For example, you can select the day, the day of month, or the month of year from a date value. In addition, you can determine the difference between (compatible) dates and also the last date of a period. The default date format is DD-MON-RR.

Let's invoke some SQL queries that illustrate how to use the DAY() date function, which is a synonym for the DAYOFMONTH() date function.

```
select * from weather;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
11 rows in set (0.000 sec)

SELECT DAY(day) from weather;
+----------+
| DAY(day) |
+----------+
|        1 |
|        2 |
|        3 |
|        1 |
|        2 |
|        3 |
|        4 |
```

```
|         6 |
|         1 |
|         2 |
|         3 |
+-----------+
11 rows in set (0.000 sec)
```

The following SQL query returns the same result as the preceding SQL query:

```
SELECT DAYOFMONTH(day) FROM weather;
```

The next SQL query selects all the rows from the `weather` table whose *day* attribute is after 2021-08-01:

```
SELECT *
FROM WEATHER
WHERE date(day) > '2021-08-01'
ORDER BY day;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
5 rows in set (0.001 sec)
```

Rounding Dates in SQL

SQL provides the `date_format()` function that enables you to round a date to a month, day, hour, or minute.

Round to the *month* with this SQL statement:

```
SELECT date_format(now(),'%Y-%m');
+-----------------------------+
| date_format(now(),'%Y-%m')  |
+-----------------------------+
| 2021-08                     |
+-----------------------------+
1 row in set (0.000 sec)
```

Round to the *day* with this SQL statement:

```
SELECT date_format(now(),'%Y-%m-%d');
+--------------------------------+
| date_format(now(),'%Y-%m-%d')  |
+--------------------------------+
| 2021-08-27                     |
+--------------------------------+
1 row in set (0.001 sec)
```

Round to the *hour* with this SQL statement:

```
SELECT date_format(now(),'%Y-%m-%d %H');
+-----------------------------------+
| date_format(now(),'%Y-%m-%d %H') |
+-----------------------------------+
| 2021-08-27 12                     |
+-----------------------------------+
1 row in set (0.000 sec)
```

Round to the *minute* with this SQL statement:

```
SELECT date_format(now(),'%Y-%m-%d %H:%i');
+--------------------------------------+
| date_format(now(),'%Y-%m-%d %H:%i') |
+--------------------------------------+
| 2021-08-27 12:57                     |
+--------------------------------------+
1 row in set (0.000 sec)
```

WORKING WITH DATE RANGES

The next SQL query selects all the rows from the `weather` table whose *day* attribute is between 2021-07-01 and 2021-08-30:

```
SELECT *
FROM WEATHER
WHERE date(day) BETWEEN '2021-07-01' AND '2021-08-30'
ORDER BY day;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
+------------+--------+------+----------+------+-------+
5 rows in set (0.002 sec)
```

The next query lists the purchase orders (and associated details) that were created between a pair of dates:

```
SELECT p.po_id, p.purchase_date, l.item_id, d.item_desc, d.item_price,l.qty
FROM customers c, purchase_orders p, line_items l, item_desc d
WHERE c.cust_id = p.cust_id
AND p.po_id = l.po_id
AND l.item_id = d.item_id
AND p.purchase_date BETWEEN '2021-01-01' AND '2021-01-31'
ORDER BY c.cust_id, p.purchase_date, p.po_id;
```

```
+-------+---------------+---------+-------------+------------+------+
| po_id | purchase_date | item_id | item_desc   | item_price | qty  |
+-------+---------------+---------+-------------+------------+------+
| 12500 | 2021-01-12    |     100 | hammer      |      20.00 |    1 |
| 12500 | 2021-01-12    |     200 | screwdriver |       8.00 |    2 |
| 12500 | 2021-01-12    |     300 | wrench      |      10.00 |    3 |
+-------+---------------+---------+-------------+------------+------+
3 rows in set (0.001 sec)
```

In the preceding query, three rows are returned because the purchase order contains three line items (i.e., one line item for each of the three items that were purchased), and the purchase date is between the specified date values. The following query will return 0 rows because there are no purchase orders that were placed prior to 2021-01-01:

```
SELECT p.po_id, p.purchase_date, l.item_id, d.item_desc, d.item_price,l.qty
FROM customers c, purchase_orders p, line_items l, item_desc d
WHERE c.cust_id = p.cust_id
AND p.po_id = l.po_id
AND l.item_id = d.item_id
AND p.purchase_date < '2021-01-01'
ORDER BY c.cust_id, p.purchase_date, p.po_id;
Empty set (0.001 sec)
```

TABLES CONTAINING MODIFICATION TIMES

A table with a date-based attribute is obviously useful for keeping track of the creation date of a row in a table. Depending on your application requirements, you might also need an attribute whose value equals the time stamp whenever the associated rows is updated.

Fortunately, MySQL supports this functionality. The table `users_dates` contains the `updated_at` attribute that is updated to the *current* timestamp whenever the contents of the associated row are modified.

```
CREATE TABLE users_dates (
    id INT(6)  NOT NULL PRIMARY KEY AUTO_INCREMENT,
    name       VARCHAR(40) NOT NULL UNIQUE,
    birth_date DATE NOT NULL,
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
    TIMESTAMP
);
```

Let's execute some SQL statements that perform the following changes to the `users_dates` table:

- Insert a row into the `users_dates` table.
- Display the contents of the `users_dates` table.
- Update the single row in the `users_dates` table.
- Display the contents of the `users_dates`.

```
— insert a single row:
INSERT INTO users_dates values(1000, 'jane jones', '2001-07-07',
'2021-03-03', '2021-03-03');
Query OK, 1 row affected (0.001 sec)
MySQL [mytools]> select name,birth_date,updated_at from users_dates;
+------------+------------+---------------------+
| name       | birth_date | updated_at          |
+------------+------------+---------------------+
| jane jones | 2001-07-07 | 2021-03-03 00:00:00 |
+------------+------------+---------------------+
1 row in set (0.002 sec)
— update the name:
UPDATE users_dates SET name='JANE Q JONES' WHERE name='jane jones';
Query OK, 1 row affected (0.021 sec)
Rows matched: 1  Changed: 1  Warnings: 0

— check the contents of the row:
MySQL [mytools]> select name,birth_date,updated_at from users_dates;
+--------------+------------+---------------------+
| name         | birth_date | updated_at          |
+--------------+------------+---------------------+
| JANE Q JONES | 2001-07-07 | 2021-09-02 17:01:35 |
+--------------+------------+---------------------+
1 row in set (0.000 sec)
```

As you can see, the `updated_at` value in the preceding row has been set equal to the current time stamp.

ARITHMETIC OPERATIONS WITH DATES

SQL enables you to subtract two dates (which returns a number) and also add or subtract a number from a date. You can also perform these operations with hours instead of days. The following two SQL queries show you how to calculate the difference between two dates:

```
SELECT DATEDIFF("2021-11-25", "2021-12-17");
+--------------------------------------+
| DATEDIFF("2021-11-25", "2021-12-17") |
+--------------------------------------+
|                                  -22 |
+--------------------------------------+
1 row in set (0.001 sec)

SELECT DATEDIFF("2021-12-25", "2021-11-17");
+--------------------------------------+
| DATEDIFF("2021-12-25", "2021-11-17") |
+--------------------------------------+
|                                   38 |
+--------------------------------------+
1 row in set (0.000 sec)
```

The following SQL query shows you how to add a number to a date (use a negative number to subtract from a date):

```
SELECT ADDDATE("2021-11-15", INTERVAL 20 DAY);
+---------------------------------------+
| ADDDATE("2021-11-15", INTERVAL 20 DAY) |
+---------------------------------------+
| 2021-12-05                            |
+---------------------------------------+
1 row in set (0.003 sec)
```

There are literally dozens of different date formats, along with SQL functions that can convert between character strings and dates. Here are some additional date functions in SQL:

- ADDTIME
- CURDATE
- CURRENT_DATE
- CURRENT_TIME
- CURRENT_TIMESTAMP
- CURTIME
- DATE_ADD
- DATE_FORMAT
- DATE_SUB
- DAYNAME
- DAYOFMONTH
- DAYOFWEEK
- DAYOFYEAR

DATE COMPONENTS AND DATE FORMATS

A date field in a database table contains the year, month, and day for a given date. However, you might need to access the individual components of a date, or perhaps change the format of a given date field.

The SQL file `date-fields-formats.sql` shows you how to extract the year, month, and day of the `purchase_date` attribute of the `purchase_orders` table and also how to display the date values with different date formats.

```
SELECT YEAR(purchase_date),MONTH(purchase_
date),DAY(purchase_date)
FROM purchase_orders;

SELECT YEAR(purchase_date) as year,
MONTH(purchase_date) as month,
DAY(purchase_date) as day
FROM purchase_orders;

SELECT cust_id,
YEAR(purchase_date) as year,
MONTH(purchase_date) as month,
DAY(purchase_date) as day
```

```
FROM purchase_orders
WHERE MONTH(purchase_date) > 1
AND DAY(purchase_date) < 5;

SELECT date_format(purchase_date, '%m-%d-%Y')
FROM purchase_orders;

SELECT date_format(purchase_date, '%d-%m-%y')
FROM purchase_orders;
```

Invoke the SQL file `date-field-formats.sql` from the MySQL prompt, as shown below:

```
MySQL [mytools] > source date-fields-formats.sql;
+---------------------+----------------------+--------------------+
| YEAR(purchase_date) | MONTH(purchase_date) | DAY(purchase_date) |
+---------------------+----------------------+--------------------+
|                2021 |                    1 |                 12 |
|                2021 |                    2 |                  3 |
|                2021 |                    7 |                  4 |
+---------------------+----------------------+--------------------+
3 rows in set (0.000 sec)

+------+-------+------+
| year | month | day  |
+------+-------+------+
| 2021 |     1 |   12 |
| 2021 |     2 |    3 |
| 2021 |     7 |    4 |
+------+-------+------+
3 rows in set (0.000 sec)

+---------+------+-------+------+
| cust_id | year | month | day  |
+---------+------+-------+------+
|    1000 | 2021 |     2 |    3 |
|    1000 | 2021 |     7 |    4 |
+---------+------+-------+------+
2 rows in set (0.000 sec)

+----------------------------------------+
| date_format(purchase_date, '%m-%d-%Y') |
+----------------------------------------+
| 01-12-2021                             |
| 02-03-2021                             |
| 07-04-2021                             |
+----------------------------------------+
3 rows in set (0.000 sec)

+----------------------------------------+
| date_format(purchase_date, '%d-%m-%y') |
+----------------------------------------+
| 12-01-21                               |
| 03-02-21                               |
| 04-07-21                               |
+----------------------------------------+
3 rows in set (0.000 sec)
```

SQL also enables you to perform conversions between numbers and characters, as well as conversions between dates and characters, such as the following:

```
NUMBER            to VARCHAR2
VARCHAR2 or CHAR to NUMBER
VARCHAR2 or CHAR to DATE
DATE              to VARCHAR2
```

FINDING THE WEEK IN DATE VALUES

MySQL makes it easy to determine the week of a given date, examples of which are as follows:

```
SELECT WEEK("2021-02-14") AS week;
+------+
| week |
+------+
|    7 |
+------+
1 row in set (0.000 sec)

SELECT WEEK("2021-12-30 14:25:16") AS week;
+------+
| week |
+------+
|   52 |
+------+
1 row in set (0.000 sec)

— CURDATE() is spelled with one "R":
SELECT WEEK(CURDATE());
+-----------------+
| WEEK(CURDATE()) |
+-----------------+
|              35 |
+-----------------+
1 row in set (0.000 sec)
```

Displaying Weekly Revenue

Listing 3.8 shows the content of `revenue.sql` that illustrates how to create a `revenue` table and then populate the table with some simulated data.

LISTING 3.8: revenue.sql

```
use mytools;

DROP TABLE IF EXISTS revenue;

CREATE TABLE revenue ( rev_date DATE, revenue INT(8),
location CHAR(20));

INSERT INTO revenue VALUES('2021-08-13', 1200,'Chicago');
INSERT INTO revenue VALUES('2021-08-15', 1000,'SF');
```

```
INSERT INTO revenue VALUES('2021-09-17', 1300,'LA');
INSERT INTO revenue VALUES('2021-09-18', 1800,'LA');
INSERT INTO revenue VALUES('2021-09-18', 1400,'Miami');
INSERT INTO revenue VALUES('2021-09-19', 2000,'Miami');
```

The following SQL statement displays the total revenue based on each location and also orders by the location:

```
SELECT location, SUM(revenue)
FROM revenue
GROUP BY location
ORDER BY location;
+----------+--------------+
| location | SUM(revenue) |
+----------+--------------+
| Chicago  |         1200 |
| LA       |         3100 |
| Miami    |         3400 |
| SF       |         1000 |
+----------+--------------+
4 rows in set (0.001 sec)
```

The following SQL statement displays the number of revenue rows based on each location and also orders the results by the location:

```
SELECT location, COUNT(revenue) AS ItemCount
FROM revenue
GROUP BY location
ORDER BY location;
+----------+-----------+
| location | ItemCount |
+----------+-----------+
| Chicago  |         1 |
| LA       |         2 |
| Miami    |         2 |
| SF       |         1 |
+----------+-----------+
4 rows in set (0.002 sec)
```

However, if we want to display the revenue *per week* and also order by both the revenue and the location, we can do so with the following SQL statement:

```
SELECT revenue, location, WEEK(rev_date) AS weekly_revenue
FROM revenue
GROUP BY revenue, location, WEEK(rev_date)
ORDER BY revenue, location, WEEK(rev_date);
+---------+----------+----------------+
| revenue | location | weekly_revenue |
+---------+----------+----------------+
|    1000 | SF       |             33 |
|    1200 | Chicago  |             32 |
|    1300 | LA       |             37 |
|    1400 | Miami    |             37 |
|    1800 | LA       |             37 |
|    2000 | Miami    |             38 |
+---------+----------+----------------+
6 rows in set (0.000 sec)
```

If we want to display the *cumulative* revenue for each week, and also order by the location, we can do so with this SQL statement:

```
SELECT location, WEEK(rev_date) AS weekly_revenue,
       SUM(revenue) AS total_sales
FROM revenue
GROUP BY location, WEEK(rev_date)
ORDER BY location, WEEK(rev_date);
+----------+----------------+-------------+
| location | weekly_revenue | total_sales |
+----------+----------------+-------------+
| Chicago  |             32 |        1200 |
| LA       |             37 |        3100 |
| Miami    |             37 |        1400 |
| Miami    |             38 |        2000 |
| SF       |             33 |        1000 |
+----------+----------------+-------------+
5 rows in set (0.001 sec)
```

ASSORTED SQL OPERATORS

SQL enables you to display a result set in descending or ascending order. For example, the following query displays the list of items in alphabetically descending order based on the `DESC` keyword:

```
SELECT item_desc DESC
FROM new_items;
```

As you can probably surmise, the following SQL query displays the list of items in alphabetically ascending order based on the `ASC` keyword:

```
SELECT item_desc ASC
FROM new_items;
```

Working with Column Aliases

A *column alias* in SQL statements is a way of representing a column heading, typically using a much shorter string. For example, you can specify the strings "c" and "p" as aliases for the `customers` and `purchase_orders` tables. If you have two tables that start with the same letter, then you can select a pair of letters to differentiate between those two tables. In general, select aliases that are short, unique, and related to the name of the table that is represented by the chosen aliases. Here is a summary of the features of column aliases:

- Renames a column heading
- Is useful with calculations
- Immediately follows the column name
- Requires double quotation marks for text that contains spaces, special characters, or case sensitive data

You can also specify the optional `AS` keyword between the column name and alias. Here is a very simple example of specifying the string `name` and **`comm`** as column aliases:

```
SELECT last_name AS name, commission_pct comm
FROM employees;
```

SQL Variables

SQL enables you to define variables, as shown in the following example that initializes an integer-valued variable and two string variables:

```
MySQL [mytools]> SET @counter := 10;
Query OK, 0 rows affected (0.002 sec)

MySQL [mytools]> SELECT @counter := 10;
+----------------+
| @counter := 10 |
+----------------+
|             10 |
+----------------+
1 row in set, 1 warning (0.000 sec)

MySQL [mytools]> SET @student_name := "Jane Smith";
Query OK, 0 rows affected (0.000 sec)

MySQL [mytools]> SET @email := "johndoe@yahoo.com";
Query OK, 0 rows affected (0.000 sec)
```

The following example illustrates how to set the value of the variable `maxp` equal to the maximum value in the `item_price` attribute in the `item_desc` table:

```
SELECT @maxp := MAX(item_desc.item_price)
FROM item_desc;
+------------------------------------+
| @maxp := MAX(item_desc.item_price) |
+------------------------------------+
|                              20.00 |
+------------------------------------+
1 row in set, 1 warning (0.001 sec)
```

The next task involves finding the *average* price of items in several stores and displaying the results in decreasing order of the average price:

```
+---------+--------------+-------+----------+
| item_id | description  | price | store_id |
+---------+--------------+-------+----------+
|       1 | apple        |  2.45 |        1 |
|       2 | banana       |  3.45 |        1 |
|       3 | cereal       |  4.20 |        2 |
|       4 | milk 1 liter |  3.80 |        2 |
|       5 | lettuce      |  1.80 |        1 |
+---------+--------------+-------+----------+
```

We can solve this task by performing the following steps:

- apply the `avg()` function to the price column
- group the values by `store_id`
- sort via the `ORDER` clause

Here is the SQL query that is based on the three preceding bullet items:

```
select avg(price), store_id
from items
group by store_id
order by avg(price);
+------------+----------+
| avg(price) | store_id |
+------------+----------+
|   1.833333 |        1 |
|   3.650000 |        3 |
|   3.820000 |        2 |
+------------+----------+
```

Earlier in this chapter, you learned how to define a SQL subquery to find the rows in the `weather` table that have the maximum temperature. Alternatively, you can initialize a variable with the maximum value and specify that variable in the following SQL statement:

```
SELECT @max1 := MAX(temper) FROM weather;
SELECT day, forecast
FROM weather
WHERE temper = @max1;
```

SQL SUMMARY REPORTS

A *summary report* is an informal term that refers to any SQL query whose output can be a tabular display of summarized data, such as a list of employees in a particular department. In general, a summary report can contain multiple subsections of summarized data, such as an alphabetical list of employees for each department, which in turn could belong to a specific region of the country. Other examples of detailed reports include quarterly business reports, machine utilization reports, network activity reports, or user activity reports.

For example, the following reports vary from basic to complex, each of which can be generated by defining suitable SQL statements:

- An alphabetical listing of employees in each division of a company
- A summary of customer purchase orders that are grouped by customer
- A summary of student grades on a quarterly basis, alphabetized by courses

- A quarterly sales report by region, division, and sales people
- A company-wide summary of quarterly revenue and expenditures by region

Enterprise-level financial systems typically contain a report-related section that provides standard reports that are based on various options, including a start date and an end date for a report. Those systems often provide support for so-called *ad hoc* reports; i.e., custom reports that are not available as standard reports.

Simple SQL Reports

The table in this section contains simplified details so that it's easier to understand the SQL statements. However, you can enhance by the inclusion of other relevant information, such as location-related information for each item sold (including state and city) and the name of the sales person who sold each item. In addition to the monthly reports, you can generate weekly reports (week-based intervals are discussed earlier in this chapter) and daily reports.

Listing 3.9 shows the content of `sold_items.sql` that illustrates how to create a `sold_items` table that contains information about sold items, such as the region where an item was sold, the quantity, the sold price, and the date when the item was sold.

LISTING 3.9: sold_items.sql

```
use mytools;

DROP TABLE IF EXISTS sold_items;
CREATE TABLE sold_items (region CHAR(20), qty INTEGER, sold_price
DECIMAL(8,2), sold_date DATE);

INSERT INTO sold_items VALUES ('branch1', 1, 15.00,'2021-12-03');
INSERT INTO sold_items VALUES ('branch1', 2, 10.00,'2021-12-03');
INSERT INTO sold_items VALUES ('branch1', 3, 10.00,'2021-12-03');

INSERT INTO sold_items VALUES ('branch2', 3, 10.00,'2021-12-01');
INSERT INTO sold_items VALUES ('branch2', 2, 10.00,'2021-12-01');
INSERT INTO sold_items VALUES ('branch2', 1, 10.00,'2021-12-01');

INSERT INTO sold_items VALUES ('branch1', 5, 10.00,'2021-11-15');
INSERT INTO sold_items VALUES ('branch1', 6, 10.00,'2021-11-15');
INSERT INTO sold_items VALUES ('branch1', 8, 15.00,'2021-11-15');

INSERT INTO sold_items VALUES ('branch1', 5, 15.00,'2021-11-10');
INSERT INTO sold_items VALUES ('branch2', 5, 10.00,'2021-11-10');
INSERT INTO sold_items VALUES ('branch3', 5, 10.00,'2021-11-10');

INSERT INTO sold_items VALUES ('branch1', 5, 15.00,'2021-11-05');
INSERT INTO sold_items VALUES ('branch2', 6, 10.00,'2021-11-05');
INSERT INTO sold_items VALUES ('branch3', 8, 10.00,'2021-11-05');
```

```
-- sold_price-based list:
\! echo '=> sold_price list:';
SELECT sold_price
FROM sold_items
ORDER BY sold_price;

-- list based on region and sold_price:
\! echo '=> region and sold_price list:';
SELECT region, sum(sold_price)
FROM sold_items
GROUP BY region
ORDER BY region, sum(sold_price) DESC;

-- list based on region and revenue:
\! echo '=> region and revenue list:';
SELECT region, sum(sold_price*qty)
FROM sold_items
GROUP BY region
ORDER BY region, sum(sold_price*qty) DESC;

-- list based on revenue and region:
\! echo '=> region and revenue list:';
SELECT sum(sold_price*qty), region
FROM sold_items
GROUP BY region
ORDER BY sum(sold_price*qty) DESC, region;

-- list based on date, revenue, and region:
\! echo '=> date, region, and revenue list:';
SELECT sold_date, sum(sold_price*qty), region
FROM sold_items
GROUP BY sold_date, region
ORDER BY sold_date, sum(sold_price*qty) DESC, region;
```

Listing 3.9 contains SQL statements that are based on the material that was covered earlier in this chapter. Now launch the code in Listing 3.9 to see the following output:

```
=> sold_price list:
+------------+
| sold_price |
+------------+
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
|      10.00 |
```

```
|       15.00 |
|       15.00 |
|       15.00 |
|       15.00 |
+-------------+
15 rows in set (0.000 sec)

=> region and sold_price list:
+---------+-----------------+
| region  | sum(sold_price) |
+---------+-----------------+
| branch1 |          100.00 |
| branch2 |           50.00 |
| branch3 |           20.00 |
+---------+-----------------+
3 rows in set (0.001 sec)

=> region and revenue list:
+---------+---------------------+
| region  | sum(sold_price*qty) |
+---------+---------------------+
| branch1 |              445.00 |
| branch2 |              170.00 |
| branch3 |              130.00 |
+---------+---------------------+
3 rows in set (0.002 sec)

=> region and revenue list:
+---------------------+---------+
| sum(sold_price*qty) | region  |
+---------------------+---------+
|              445.00 | branch1 |
|              170.00 | branch2 |
|              130.00 | branch3 |
+---------------------+---------+
3 rows in set (0.000 sec)

=> date, region, and revenue list:
+------------+---------------------+---------+
| sold_date  | sum(sold_price*qty) | region  |
+------------+---------------------+---------+
| 2021-11-05 |               80.00 | branch3 |
| 2021-11-05 |               75.00 | branch1 |
| 2021-11-05 |               60.00 | branch2 |
| 2021-11-10 |               75.00 | branch1 |
| 2021-11-10 |               50.00 | branch2 |
| 2021-11-10 |               50.00 | branch3 |
| 2021-11-15 |              230.00 | branch1 |
| 2021-12-01 |               60.00 | branch2 |
| 2021-12-03 |               65.00 | branch1 |
+------------+---------------------+---------+
9 rows in set (0.000 sec)
```

Calculating SubTotals

Listing 3.10 shows the content of `sub_totals.sql` that illustrates how to calculate subtotals for data in the numeric column `amount` that represents a fictitious set of revenue figures.

LISTING 3.10: sub_totals.sql

```
use mytools;
DROP TABLE IF EXISTS invoices;
CREATE TABLE invoices (id INTEGER, amount INTEGER, the_date
date);

INSERT INTO invoices VALUES (1000,1000,'2021-10-01');
INSERT INTO invoices VALUES (1000,300, '2022-11-03');
INSERT INTO invoices VALUES (1000,400, '2022-12-07');

INSERT INTO invoices VALUES (2000,2500,'2021-01-08');
INSERT INTO invoices VALUES (3000,3600,'2022-02-09');
INSERT INTO invoices VALUES (4000,4700,'2022-03-10');

SELECT  id, SUM(amount) AS total_amount
FROM invoices
GROUP BY id WITH ROLLUP
```

Listing 3.10 creates the invoices table with a set of rows, some of which have the same `id` value of 1000. The subtotal for the three rows equals 1700, which is displayed in the output below. As you can see, there is only *one* row for each `id` value, whereas the code sample in the next section generates an output row for *every* row in the table.

The remaining three rows have distinct `id` values, and therefore the subtotal for each of those rows. Launch the code in Listing 3.10 to see the following output:

```
+------+--------------+
| id   | total_amount |
+------+--------------+
| 1000 |         1700 |
| 2000 |         2500 |
| 3000 |         3600 |
| 4000 |         4700 |
| NULL |        12500 |
+------+--------------+
5 rows in set (0.000 sec)
```

Calculating "Running" (Cumulative) Totals

Listing 3.11 shows the content of `running_totals.sql` that illustrates how to calculate cumulative totals for data in the numeric column `amount` that represents a fictitious set of revenue figures.

LISTING 3.11: running_totals.sql

```
use mytools;
DROP TABLE IF EXISTS invoices;
CREATE TABLE invoices (id INTEGER, amount INTEGER, the_date date);

INSERT INTO invoices VALUES (1000,1000,'2021-10-01');
INSERT INTO invoices VALUES (1000,300, '2022-11-03');
INSERT INTO invoices VALUES (1000,400, '2022-12-07');

INSERT INTO invoices VALUES (2000,2500,'2021-01-08');
INSERT INTO invoices VALUES (3000,3600,'2022-02-09');
INSERT INTO invoices VALUES (4000,4700,'2022-03-10');

SELECT id, the_date, amount,
       SUM(amount) OVER (ORDER BY id) as total_sum
FROM invoices;
```

Listing 3.11 differs from Listing 3.10 only in the SQL statement, which in this example generates "running" totals instead of subtotals. The key difference in this SQL statement is shown in bold in Listing 3.11. As you will see in the output, an output row is generated for each row in the table. Launch the code in Listing 3.11 to see the following output:

```
+------+------------+--------+-----------+
| id   | the_date   | amount | total_sum |
+------+------------+--------+-----------+
| 1000 | 2021-10-01 |   1000 |      1700 |
| 1000 | 2022-11-03 |    300 |      1700 |
| 1000 | 2022-12-07 |    400 |      1700 |
| 2000 | 2021-01-08 |   2500 |      4200 |
| 3000 | 2022-02-09 |   3600 |      7800 |
| 4000 | 2022-03-10 |   4700 |     12500 |
+------+------------+--------+-----------+
6 rows in set (0.000 sec)
```

The choice of Listing 3.10 versus Listing 3.11 depends on the output that you want to display in your report.

SUMMARY

This chapter introduced you to the SQL `JOIN` keyword on two tables, along with examples of different types of `JOIN` statements. which can be extended to multiple tables. In addition, you learned how to create views, and the advantages they provide over tables.

Next, you learned about primary keys, unique keys, and foreign keys, along with an example of defining a foreign key in one table (`child_tbl`) that references a primary key in another table (`parent_tbl`).

In addition, you saw examples of SQL statements that contain `GROUP BY`, `HAVING`, and `ORDER BY` clauses, as well as well as how to use the `ROLLUP` keyword in a SQL statement. Finally, you learned how to generate SQL-based reports based on sold items in a database table.

CHAPTER 4

ASSORTED SQL FUNCTIONS

This chapter contains a variety of SQL topics, such as aggregate functions, scalar functions, and string functions in SQL. You will also learn how to work with dates in SQL, date ranges, date components, and the SQL `CASE` statement.

The first section introduces numeric functions in SQL, such as `LENGTH()`, `MOD()`, and `ROUND()`. You will also learn about logarithmic, exponential, and trigonometric functions in SQL.

The second section contains SQL statements that illustrate how to use aggregate functions and scalar functions in SQL, such as the `max()` and `min()` functions. You will see SQL statements that use the `LIMIT` and `OFFSET` keywords that enable you to find the `kth` largest value in a column and a range of values in a sorted set of numeric values. Moreover, you will learn about string functions in SQL and how to use the `substring()` function.

The third section contains examples of Boolean operators and set operators, and how to use the `AND`, `OR`, and `NOT` operators in SQL statements. The fourth section introduces the `ORDER BY` clause that is illustrated in various SQL statements. This section also discusses the `MATCH()` function, along with CTEs (common table expressions), which were introduced in MySQL 8.0.

The final portion of this chapter contains an example of linear regression in SQL, a section about window functions, the SQL `CASE` statement, and how to work with `NULL` values in SQL.

In some cases, the initial `MySQL [mytools]>` string has been omitted in the output listings to improve readability.

NUMERIC FUNCTIONS IN SQL

SQL provides various built-in functions that return numeric values or provide formatting features for numeric values, some of which are listed here:

- FORMAT()
- LEN() or LENGTH()
- MOD()
- ROUND()
- POSITION()

The SQL FORMAT() function enables you to format numeric values in various ways. For example, the following SQL statement displays the closest integer value to the decimal number 123.456:

```
SELECT FORMAT(123.456, 0);
+--------------------+
| FORMAT(123.456, 0) |
+--------------------+
| 123                |
+--------------------+
1 row in set (0.000 sec)
```

The following SQL statement shows you how to use the FORMAT() function to round a number to the nearest integer:

```
SELECT FORMAT(123.789, 0);
+--------------------+
| FORMAT(123.789, 0) |
+--------------------+
| 124                |
+--------------------+
1 row in set (0.000 sec)
```

If you work with decimal values that represent currency, you can round numbers to two decimal places with this SQL statement:

```
SELECT FORMAT(123.789, 2);
+--------------------+
| FORMAT(123.789, 2) |
+--------------------+
| 123.79             |
+--------------------+
1 row in set (0.000 sec)
```

The following SQL statement illustrates how to use the LEN() function to find the length of the strings in the item_desc attribute of the new_items table:

```
SELECT LENGTH(item_desc), item_desc
FROM new_items;
```

Launch the preceding SQL statement to see the following output:

```
+-------------------+----------------+
| LENGTH(item_desc) | item_desc      |
+-------------------+----------------+
|                 6 | hammer         |
|                11 | screwdriver    |
|                 6 | wrench         |
|                 6 | pliers         |
|                 8 | ballpeen       |
|                14 | 1/4 inch nails |
|                 9 | Toolbox S      |
|                 9 | Toolbox M      |
|                 9 | Toolbox L      |
|                 7 | Handsaw        |
+-------------------+----------------+
10 rows in set (0.000 sec)
```

As a variant of the preceding example, the following SQL statement selects the rows in which the length of the description is between 6 and 14:

```
SELECT LENGTH(item_desc), item_desc
FROM new_items
WHERE LENGTH(item_desc) > 6 AND LENGTH(item_desc) < 14;
+-------------------+-------------+
| LENGTH(item_desc) | item_desc   |
+-------------------+-------------+
|                11 | screwdriver |
|                 8 | ballpeen    |
|                 9 | Toolbox S   |
|                 9 | Toolbox M   |
|                 9 | Toolbox L   |
|                 7 | Handsaw     |
+-------------------+-------------+
6 rows in set (0.000 sec)
```

The `MOD()` function returns the integer remainder of dividing an integer (positive or negative) by a non-zero integer, as shown here:

```
MySQL [mytools]> SELECT MOD(7,3);
+----------+
| MOD(7,3) |
+----------+
|        1 |
+----------+
1 row in set (0.004 sec)

MySQL [mytools]> SELECT MOD(-7,3);
+-----------+
| MOD(-7,3) |
+-----------+
|        -1 |
+-----------+
1 row in set (0.003 sec)
```

```
SELECT MOD(7,0);
+-----------+
| MOD(7,0)  |
+-----------+
|      NULL |
+-----------+
1 row in set, 1 warning (0.002 sec)
```

The following SQL statement illustrates how to use the `POSITION()` function to find the index position of the first space character in a text string (which is 0 if the string does not contain any spaces):

```
SELECT emp_id, POSITION(" " in title) space_index
FROM employees;
+--------+-------------+
| emp_id | space_index |
+--------+-------------+
|   1000 |           0 |
|   2000 |           8 |
|   3000 |           4 |
|   4000 |           7 |
+--------+-------------+
4 rows in set (0.000 sec)
```

The `ROUND()` function calculates the rounded integer value for a numeric field (or decimal point values), an example of which is shown here:

```
SELECT ROUND(123.789, 2);
+-------------------+
| ROUND(123.789, 2) |
+-------------------+
|            123.79 |
+-------------------+
1 row in set (0.003 sec)
```

Calculated Columns

The SQL statements in this section show you how to calculate a percentage of a numeric column, which is useful when you need to display tax-related values. Note that the SQL statements illustrate the `ORDER BY` clause (which has an intuitive purpose) that we'll explore in greater detail later in this chapter. The following `SQL` statement calculates a tax of 8% for each item:

```
SELECT item_price, item_price*0.08 AS TAX
FROM item_desc
ORDER BY item_price;
+------------+--------+
| item_price | TAX    |
+------------+--------+
|       8.00 | 0.6400 |
|      10.00 | 0.8000 |
|      20.00 | 1.6000 |
+------------+--------+
3 rows in set (0.000 sec)
```

We can calculate the `item_price`, the `tax`, and the total price for each item in the `item_desc` table by creating a view over the `item_desc` table and then selecting everything from the view, as shown here:

```
CREATE OR REPLACE VIEW v_item_desc AS
SELECT item_id, item_price, item_price*0.08, item_price*(1.08) AS TOTAL
FROM item_desc
ORDER BY item_id;
Query OK, 0 rows affected (0.004 sec)
```

Now select everything from the view:

```
select * from v_item_desc;
+---------+------------+-----------------+---------+
| item_id | item_price | item_price*0.08 | TOTAL   |
+---------+------------+-----------------+---------+
|     100 |      20.00 |          1.6000 | 21.6000 |
|     100 |      10.00 |          0.8000 | 10.8000 |
|     200 |       8.00 |          0.6400 |  8.6400 |
+---------+------------+-----------------+---------+
3 rows in set (0.002 sec)
```

The following SQL statement displays a "$" currency symbol on the left side of each item price:

```
SELECT CONCAT('$', item_price)
FROM item_desc
ORDER BY item_price;
+-------------------------+
| CONCAT('$', item_price) |
+-------------------------+
| $8.00                   |
| $10.00                  |
| $20.00                  |
+-------------------------+
3 rows in set (0.003 sec)
```

THE ROUND(), CEIL(), AND FLOOR() FUNCTIONS

This section contains examples of rounding a number, calculating the ceiling, and calculating the floor of a number using the functions `round()`, `ceil()`, and `floor()`, respectively.

Listing 4.1 shows the content of `round_values.sql` that illustrates the result of invoking the `ROUND()` function on various decimal values.

LISTING 4.1: round_values.sql

```
SELECT ROUND(7.51);  -- 8
SELECT ROUND(7.49);  -- 7
SELECT ROUND(-7.51); -- -8

SELECT ROUND(25e-1); -- 2  The nearest even value = 2
SELECT ROUND(35e-1); -- 4  The nearest even
```

```
-- Round to two decimal places:
SELECT ROUND(234.567, 2);  -- 234.57
SELECT ROUND(234.567, -2); -- 200
```

Listing 4.1 contains SQL statements that involve the `ROUND()` function. For approximate numeric values, the result of the `ROUND()` function depends on the C library. In fact, the ROUND() function often uses the "round to the nearest even" rule, which means that 2.5 rounds to 2 whereas 3.5 rounds to 4. Launch the code in Listing 4.1 to see the following output:

```
+-------------+
| ROUND(7.51) |
+-------------+
|           8 |
+-------------+
1 row in set (0.000 sec)

+-------------+
| ROUND(7.49) |
+-------------+
|           7 |
+-------------+
1 row in set (0.000 sec)

+--------------+
| ROUND(-7.51) |
+--------------+
|           -8 |
+--------------+
1 row in set (0.002 sec)

+--------------+
| ROUND(25e-1) |
+--------------+
|            2 |
+--------------+
1 row in set (0.000 sec)

+--------------+
| ROUND(35e-1) |
+--------------+
|            4 |
+--------------+
1 row in set (0.000 sec)

+-------------------+
| ROUND(234.567, 2) |
+-------------------+
|            234.57 |
+-------------------+
1 row in set (0.000 sec)

+--------------------+
| ROUND(234.567, -2) |
+--------------------+
|                200 |
+--------------------+
1 row in set (0.000 sec)
```

Listing 4.2 shows the content of `ceil_floor.sql` that illustrates the result of invoking the `ceil()` function and the `floor()` function on various decimal values.

LISTING 4.2: ceil_floor.sql

```
-- round up:
SELECT CEIL(4.56);     -- 5
SELECT CEILING(7.83); -- 8
SELECT CEIL(-3.01);    -- -4

-- round down:
SELECT FLOOR(3.99);    -- 3
SELECT FLOOR(-3.01);   -- -4
```

Listing 4.2 contains two occurrences of `CEIL()` to show you that the first `CEIL()` function rounds up to the nearest integer, whereas the other `CEIL()` function rounds down to the nearest integer. Launch the code in Listing 4.2 to see the following output:

```
+------------+
| CEIL(4.56) |
+------------+
|          5 |
+------------+
1 row in set (0.000 sec)

+---------------+
| CEILING(7.83) |
+---------------+
|             8 |
+---------------+
1 row in set (0.000 sec)

+-------------+
| CEIL(-3.01) |
+-------------+
|          -3 |
+-------------+
1 row in set (0.000 sec)

+-------------+
| FLOOR(3.99) |
+-------------+
|           3 |
+-------------+
1 row in set (0.000 sec)

+--------------+
| FLOOR(-3.01) |
+--------------+
|           -4 |
+--------------+
1 row in set (0.000 sec)
```

SQL Queries with the rand() Function

The `RAND()` function generates a random number between 0 and 1, an example of which is here (and invoked twice):

```
SELECT RAND();
+--------------------+
| RAND()             |
+--------------------+
| 0.4952851277732152 |
+--------------------+
1 row in set (0.002 sec)

SELECT RAND();
+---------------------+
| RAND()              |
+---------------------+
| 0.13495801774315352 |
+---------------------+
1 row in set (0.000 sec)
```

The `RAND()` function enables you to select a random set of rows from a table, as shown here:

```
SELECT *
FROM weather
ORDER BY RAND()
LIMIT 3;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
+------------+--------+------+----------+------+-------+
3 rows in set (0.002 sec)
```

The preceding SQL statement retrieves a set of three random rows, and obviously you can specify a different number or omit the `LIMIT` clause.

LOG, EXPONENTIAL, AND TRIG FUNCTIONS IN SQL

SQL supports logarithmic, exponential functions, and several trigonometric functions. If you are familiar with such functions, then the SQL statements in this section are straightforward. If you need to use these functions and you are unfamiliar with the underlying mathematical concepts, perform an online search for articles that provide the necessary details. With the preceding details in mind, here is a list of SQL statements involving mathematical functions.

```
SELECT LN(2), LN(5), LN(-5);
+--------------------+--------------------+--------+
| LN(2)              | LN(5)              | LN(-5) |
+--------------------+--------------------+--------+
| 0.6931471805599453 | 1.6094379124341003 |   NULL |
+--------------------+--------------------+--------+
1 row in set, 1 warning (0.002 sec)

SELECT LOG(2), LOG(2, 250), LOG(10, 250);
+--------------------+-------------------+-------------------+
| LOG(2)             | LOG(2, 250)       | LOG(10, 250)      |
+--------------------+-------------------+-------------------+
| 0.6931471805599453 | 7.965784284662087 | 2.397940008672037 |
+--------------------+-------------------+-------------------+
1 row in set (0.001 sec)

SELECT LOG2(250), LOG2(24567), LOG2(-23234);
+-------------------+-------------------+--------------+
| LOG2(250)         | LOG2(24567)       | LOG2(-23234) |
+-------------------+-------------------+--------------+
| 7.965784284662087 | 14.58443407325384 |         NULL |
+-------------------+-------------------+--------------+
1 row in set, 1 warning (0.000 sec)

SELECT EXP(0), EXP(2), EXP(-2);
+--------+------------------+--------------------+
| EXP(0) | EXP(2)           | EXP(-2)            |
+--------+------------------+--------------------+
|      1 | 7.38905609893065 | 0.1353352832366127 |
+--------+------------------+--------------------+
1 row in set (0.001 sec)

SELECT ATAN(4), ATAN(24), ATAN(-32);
+--------------------+--------------------+---------------------+
| ATAN(4)            | ATAN(24)           | ATAN(-32)           |
+--------------------+--------------------+---------------------+
| 1.3258176636680326 | 1.5291537476963082 | -1.5395564933646284 |
+--------------------+--------------------+---------------------+
1 row in set (0.000 sec)

SELECT ATAN2(1, 5), ATAN2(-2, 3), ATAN(3.5, 0);
+---------------------+---------------------+--------------------+
| ATAN2(1, 5)         | ATAN2(-2, 3)        | ATAN(3.5, 0)       |
+---------------------+---------------------+--------------------+
| 0.19739555984988075 | -0.5880026035475675 | 1.5707963267948966 |
+---------------------+---------------------+--------------------+
1 row in set (0.000 sec)

SELECT COS(0), COS(1), COS(2.5);
+--------+--------------------+---------------------+
| COS(0) | COS(1)             | COS(2.5)            |
+--------+--------------------+---------------------+
|      1 | 0.5403023058681398 | -0.8011436155469337 |
+--------+--------------------+---------------------+
1 row in set (0.000 sec)
```

```
SELECT POW(3, 2), POW(25, 5), POW(16, 2);
+-----------+-----------+------------+
| POW(3, 2) | POW(25, 5) | POW(16, 2) |
+-----------+-----------+------------+
|         9 |    9765625 |        256 |
+-----------+-----------+------------+
1 row in set (0.000 sec)

SELECT POWER(3, 2), POWER(25, 5), POWER(16, 2);
+-------------+-------------+--------------+
| POWER(3, 2) | POWER(25, 5) | POWER(16, 2) |
+-------------+-------------+--------------+
|           9 |      9765625 |          256 |
+-------------+-------------+--------------+
1 row in set (0.000 sec)
```

Calculate the harmonic mean as follows:

```
SELECT COUNT(temper) / SUM( 1/temper ) AS harmonic
FROM weather;
+----------+
| harmonic |
+----------+
|  40.7391 |
+----------+
1 row in set (0.000 sec)
```

Calculate the geometric mean as follows:

```
SELECT EXP(SUM(LOG(temper)) / COUNT(temper)) AS geometricmean
FROM weather;
+--------------------+
| geometricmean      |
+--------------------+
| 45.102493300236915 |
+--------------------+
1 row in set (0.000 sec)

SELECT RADIANS(90), RADIANS(180), RADIANS(360);
+--------------------+-------------------+-------------------+
| RADIANS(90)        | RADIANS(180)      | RADIANS(360)      |
+--------------------+-------------------+-------------------+
| 1.5707963267948966 | 3.141592653589793 | 6.283185307179586 |
+--------------------+-------------------+-------------------+
1 row in set (0.000 sec)

SELECT CONV('A', 16, 2), CONV('G', 18, 8);
+------------------+------------------+
| CONV('A', 16, 2) | CONV('G', 18, 8) |
+------------------+------------------+
| 1010             | 20               |
+------------------+------------------+
1 row in set (0.000 sec)
```

SCALAR FUNCTIONS IN SQL

A *scalar* function returns a single value based on the input value. The following list contains some commonly used SQL scalar functions:

- `LENGTH()`: Calculates the total length of the given field (column)
- `UCASE()`: Converts a collection of string values to uppercase characters
- `LCASE()`: Converts a collection of string values to lowercase characters
- `MID()`: Extracts substrings from a collection of string values in a table
- `SUBSTRING()`: Extracts substrings from a collection of string values in a table
- `CONCAT()`: Concatenates two or more strings
- `RAND()`: Generates a random collection of numbers of given length
- `ROUND()`: Calculates the rounded integer value for a number (or decimal values)
- `NOW()`: Returns the current data and time
- `FORMAT()`: Sets the format to display a collection of values

The following SQL statement selects the first five characters of the `title` attribute of the `employees` table:

```
SELECT SUBSTR(title,1,5)
FROM employees;
+-------------------+
| SUBSTR(title,1,5) |
+-------------------+
| Devel             |
| Proje             |
| Dev M             |
| Senio             |
+-------------------+
4 rows in set (0.001 sec)
```

The following SQL statement selects the characters in columns 3 through 9 of the `title` attribute of the `employees` table:

```
SELECT SUBSTR(title,3,9)
FROM employees;
+-------------------+
| SUBSTR(title,3,9) |
+-------------------+
| veloper           |
| oject Lea         |
| v Manager         |
| nior Dev          |
+-------------------+
4 rows in set (0.000 sec)
```

AGGREGATE FUNCTIONS IN SQL

An *aggregate* function performs operations on a collection of values to return a single scalar value. Aggregate functions are often used with the GROUP BY and HAVING clauses of the SELECT statement.

The following list contains some commonly used SQL aggregate functions (followed by simple examples):

- AVG(): Calculates the mean of a collection of values
- COUNT(): Counts the total number of records in a specific table or view
- MAX(): Calculates the maximum of a collection of values
- MIN(): Calculates the minimum of a collection of values
- SUM(): Calculates the sum of a collection of values

One other detail to keep in mind is that except for the COUNT() function, all the aggregate functions in the preceding list ignore NULL values.

Let's look at examples of SQL statements that contain the aggregate functions in the preceding bullet list. First, let's review the contents of the new_items table that was created and populated with data in Chapter 3:

```
SELECT * FROM new_items;
+---------+----------------+------------+
| item_id | item_desc      | item_price |
+---------+----------------+------------+
|     100 | hammer         |      20.00 |
|     200 | screwdriver    |       8.00 |
|     100 | wrench         |      10.00 |
|     400 | pliers         |      10.00 |
|     500 | ballpeen       |      20.00 |
|     600 | 1/4 inch nails |       8.00 |
|     700 | Toolbox S      |      30.00 |
|     800 | Toolbox M      |      40.00 |
|     900 | Toolbox L      |      50.00 |
|    1000 | Handsaw        |      20.00 |
+---------+----------------+------------+
10 rows in set (0.002 sec)
```

The following SQL statement illustrates how to use the MAX() function to find the maximum value in the item_price field of the new_items table:

```
SELECT MAX(item_price) FROM new_items;
```

Launch the preceding SQL statement to see the following output:

```
+-----------------+
| MAX(item_price) |
+-----------------+
|           50.00 |
+-----------------+
1 row in set (0.002 sec)
```

The following SQL statement illustrates how to use the `MIN()` function to find the minimum value in the `item_price` field in the `new_items` table:

```
SELECT MIN(item_price) FROM new_items;
```

Invoke the preceding SQL statement to see the following output:

```
+-----------------+
| MIN(item_price) |
+-----------------+
|            8.00 |
+-----------------+
1 row in set (0.002 sec)
```

The following SQL statement illustrates how to use the `AVG()` function to find the average value in the `item_price` field in the `new_items` table:

```
SELECT AVG(item_price) FROM new_items;
```

Launch the preceding SQL statement to see the following output:

```
+-----------------+
| AVG(item_price) |
+-----------------+
|       21.600000 |
+-----------------+
1 row in set (0.002 sec)
```

The following SQL statement illustrates how to use the `COUNT()` function to find the number of rows in the `item_price` field in the `new_items` table:

```
SELECT COUNT(*) FROM new_items;
```

Launch the preceding SQL statement to see the following output:

```
+----------+
| COUNT(*) |
+----------+
|       10 |
+----------+
1 row in set (0.002 sec)
```

The following SQL statement illustrates how to use the `SUM()` function to find the sum of the values in the `item_price` field in the `new_items` table:

```
SELECT SUM(item_price) FROM new_items;
```

Launch the preceding SQL statement to see the following output:

```
+-----------------+
| SUM(item_price) |
+-----------------+
|          216.00 |
+-----------------+
1 row in set (0.002 sec)
```

Now that you have seen a few examples of SQL statements that contain aggregate functions in SQL, the next section discusses scalar functions in SQL, along with some SQL statements that use those functions.

SQL QUERIES WITH THE MAX() AND MIN() FUNCTIONS

The following SQL statement retrieves the maximum `student_id` and the minimum `student_id` from the `schedule` table:

```
SELECT max(student_id), min(student_id)
FROM schedule;
+-----------------+-----------------+
| max(student_id) | min(student_id) |
+-----------------+-----------------+
| 1060            | 1010            |
+-----------------+-----------------+
1 row in set (0.001 sec)
```

The following SQL statement retrieves the maximum `student_id` and the minimum `student_id` using the `GROUP BY` keywords for the term from the `schedule` table:

```
SELECT max(student_id), min(student_id)
FROM schedule
GROUP BY term;
+-----------------+-----------------+
| max(student_id) | min(student_id) |
+-----------------+-----------------+
| 1020            | 1010            |
| 1020            | 1020            |
| 1060            | 1030            |
+-----------------+-----------------+
3 rows in set (0.002 sec)
```

The following SQL statement retrieves the maximum `student_id` and the minimum `student_id` using the `GROUP BY` clause for the `term` as well as the `ORDER BY` clause for the `term` from the `schedule` table:

```
SELECT max(student_id), min(student_id)
FROM schedule
GROUP BY term
ORDER BY term;
+-----------------+-----------------+
| max(student_id) | min(student_id) |
+-----------------+-----------------+
| 1060            | 1030            |
| 1020            | 1010            |
| 1020            | 1020            |
+-----------------+-----------------+
3 rows in set (0.002 sec)
```

Notice that the following SQL statement generates an error without the `GROUP BY` keywords in the SQL statement:

```
SELECT max(student_id), min(student_id), term FROM
schedule;
ERROR 1140 (42000): In aggregated query without GROUP BY,
expression #3 of SELECT list contains nonaggregated column
'mytools.schedule.term'; this is incompatible with sql_
mode=only_full_group_by
1 row in set (0.001 sec)
```

FIND MAXIMUM VALUES WITH SQL SUBQUERIES

This section contains examples of SQL statements that involve the `MAX()` function and SQL subqueries that contain the `MAX()` function. If need be, you can replace the occurrences of `MAX()` with `MIN()` in the following SQL queries.

As a starting point, let's look at an incorrect SQL statement that might look as though it returns the maximum temperature in the `weather` table:

```
SELECT temper
FROM weather
WHERE temper = MAX(temper);
```

The output from the preceding SQL query is shown here:

```
ERROR 1111 (HY000): Invalid use of group function
```

Fortunately, we can find the maximum temperature in the `weather` table with the following SQL query:

```
SELECT MAX(temper)
FROM weather;
+--------------+
| MAX(temper) |
+--------------+
|           78 |
+--------------+
1 row in set (0.000 sec)
```

Another way to find the maximum temperature is with this SQL query that does not contain a `WHERE` keyword:

```
SELECT temper
FROM weather
ORDER BY temper DESC
LIMIT 1;
```

The output from the preceding SQL query displays a single value, as shown below:

```
+--------+
| temper |
+--------+
|     78 |
+--------+
1 row in set (0.000 sec)
```

Modify the preceding SQL query to display the *top two* temperatures, as shown here:

```
SELECT temper
FROM weather
ORDER BY temper
DESC LIMIT 2;
```

The output from the preceding SQL query displays two values, as follows:

```
+---------+
| temper |
+---------+
|      78 |
|      78 |
+---------+
2 rows in set (0.001 sec)
```

Notice that the previous output consists of two occurrences of the value 78. Modify the preceding SQL query to display the top two *distinct* temperatures, as shown here:

```
SELECT DISTINCT temper
FROM weather
ORDER BY temper
DESC LIMIT 2;
```

The output from the preceding SQL query displays the two largest distinct values, as shown below:

```
+---------+
| temper |
+---------+
|      78 |
|      51 |
+---------+
2 rows in set (0.001 sec)
```

The following SQL query displays the maximum temperature for each day:

```
SELECT day, MAX(temper)
FROM weather
GROUP BY day;
```

The output from the preceding SQL query displays a single value:

```
+------------+-------------+
| day        | MAX(temper) |
+------------+-------------+
| 2021-04-01 |          42 |
| 2021-04-02 |          45 |
| 2021-04-03 |          78 |
| 2021-07-01 |          42 |
| 2021-07-02 |          45 |
```

```
| 2021-07-03 |          78 |
| 2021-08-04 |          50 |
| 2021-08-06 |          51 |
| 2021-09-01 |          42 |
| 2021-09-02 |          45 |
| 2021-09-03 |          15 |
+------------+------------+
11 rows in set (0.000 sec)
```

Since the preceding SQL query returns all the rows in the `weather` table, so how do we know for certain that the maximum temperature is returned for each day? One way to convince ourselves is to create the table `weather2` as a copy of the table `weather`, and insert rows with different temperatures for the same day.

Listing 4.3 shows the content of `weather2.sql` that performs the steps described in the preceding paragraph.

LISTING 4.3: weather2.sql

```
use mytools;

DROP TABLE IF EXISTS weather2;
CREATE TABLE weather2 AS (SELECT * FROM weather);

INSERT INTO weather2 VALUES( '2021-04-01',62, 16,  'Rain', 'sf', 'ca');
INSERT INTO weather2 VALUES( '2021-04-02',65, 3,   'Sunny','sf', 'ca');
INSERT INTO weather2 VALUES( '2021-04-03',98, -12, NULL,   'se', 'wa');

SELECT COUNT(*) FROM weather2;

SELECT day, MAX(temper)
FROM weather2
GROUP BY day;
```

Launch the code in Listing 4.3 to see that `weather2` contains 14 rows, whereas the final SQL query in Listing 4.3 returns only 11 rows:

```
source weather2.sql;
Database changed
Query OK, 0 rows affected (0.011 sec)

Query OK, 11 rows affected (0.010 sec)
Records: 11  Duplicates: 0  Warnings: 0

Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)
Query OK, 1 row affected (0.001 sec)

+----------+
| COUNT(*) |
+----------+
|       14 |
+----------+
1 row in set (0.001 sec)
```

```
+------------+-------------+
| day        | MAX(temper) |
+------------+-------------+
| 2021-04-01 |          62 |
| 2021-04-02 |          65 |
| 2021-04-03 |          98 |
| 2021-07-01 |          42 |
| 2021-07-02 |          45 |
| 2021-07-03 |          78 |
| 2021-08-04 |          50 |
| 2021-08-06 |          51 |
| 2021-09-01 |          42 |
| 2021-09-02 |          45 |
| 2021-09-03 |          15 |
+------------+-------------+
11 rows in set (0.000 sec)
```

As you can see, the value shown in bold is the new maximum temperature for the date `2021-04-01`, which is greater than the temperature of 42 for the same day.

Simplify SQL Queries Containing Subqueries

In the previous section, you saw examples of the capability of subqueries in SQL statements, and it's important to avoid defining SQL statements with unnecessary complexity.

For example, suppose we want to display the rows in the `weather` table on a day that has the maximum temperature. We can do so with the following query:

```
SELECT * FROM weather
  WHERE day = (
    SELECT day FROM weather
    WHERE temper = (
      SELECT MAX(temper) FROM weather limit 1)
    limit 1
    );
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
+------------+--------+------+----------+------+-------+
1 row in set (0.000 sec)
```

We can simplify the preceding SQL statement with the following statement:

```
SELECT * FROM weather
    WHERE temper = (
      SELECT MAX(temper) FROM weather LIMIT 1)
    LIMIT 1;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
+------------+--------+------+----------+------+-------+
1 row in set (0.005 sec)
```

However, there are some details to keep in mind. First, the two preceding SQL statements contain the code snippet `LIMIT 1` in the subqueries. This is necessary because the `WHERE temper =` code snippet must be assigned a unique value. An error occurs without the preceding code snippet. You can confirm this detail by removing the `LIMIT 1` code snippet from the SQL statements.

The second point is that there are *two* rows that have the maximum temperature. To find all such rows, and *only* rows with the maximum temperature, the solution is shown later in this chapter in the section that discusses the `IN` keyword.

FIND TOP-RANKED NUMERIC VALUES

The previous section showed you how to find the largest value in a column of a table, whereas "top-ranked" refers to values such as the second largest or third largest value in a column of a table, both of which are illustrated in the next subsection.

Find the Second and Third Largest Values in a Column

The second largest temperature in the `weather` table is easy to find via a SQL subquery:

```
SELECT MAX(temper)
FROM weather
WHERE temper < ( SELECT MAX(temper) FROM weather);
```

The output from the preceding SQL query is here:

```
+-------------+
| MAX(temper) |
+-------------+
|          51 |
+-------------+
1 row in set (0.001 sec)
```

Incidentally, this task is sometimes given as an interview question, and now you know how easy it is to solve this task if you understand SQL subqueries.

You can easily modify the preceding SQL query to find the third largest temperature with this SQL query:

```
SELECT MAX(temper)
FROM weather
WHERE temper < (
      SELECT MAX(temper)
      FROM weather
      WHERE temper < ( SELECT MAX(temper) FROM weather));
```

The output from the preceding SQL query is shown below, which returns the value 50:

```
+--------------+
| MAX(temper)  |
+--------------+
|           50 |
+--------------+
1 row in set (0.001 sec)
```

Another way to find the second largest temperature in the `weather` table is shown here:

```
SELECT MAX(temper)
FROM weather
WHERE temper NOT IN (SELECT MAX(temper) FROM weather);
```

The output from the preceding SQL query is here:

```
+--------------+
| MAX(temper)  |
+--------------+
|           51 |
+--------------+
1 row in set (0.001 sec)
```

Find the Top Three Values in a Column

The largest three temperatures in the `weather` table are easy to find by means of a simple SQL query that does not involve a subquery, as shown here:

```
SELECT temper
FROM weather
ORDER BY temper DESC
LIMIT 3;
```

The output from the preceding SQL query is here. Notice that 78 appears twice as the largest value:

```
+---------+
| temper |
+---------+
|      78 |
|      78 |
|      51 |
+---------+
3 rows in set (0.001 sec)
```

If you want to display the top `n` temperatures, simply replace the integer 3 in the preceding query with the (positive integer) `n`.

One other detail: The preceding SQL query returns the three largest temperature values with duplicates as well. This result is correct: since the two temperatures of 78 are "tied for first," the temperature of 51 is the third largest value.

FIND VALUES WITH THE OFFSET KEYWORD

The previous section showed you how to find the largest, second largest, and third largest values using a SQL subquery, which can be cumbersome when you're trying to find values that are further from the maximum value. A better solution involves the `LIMIT` keyword to find the top `k` values in a column. This section shows you how to find the following without SQL subqueries:

- The kth largest value (and only the kth value) in a column
- Any contiguous range of values in a sort set of numbers

For example, the following SQL statement finds the fifth largest value in the `weather` table:

```
SELECT temper
FROM weather
ORDER BY temper DESC
LIMIT 1 OFFSET 4;
+--------+
| temper |
+--------+
|     45 |
+--------+
1 row in set (0.174 sec)
```

The preceding SQL query specifies an offset of 4, which means that the four largest values are skipped, and then the fifth largest value is selected because the `LIMIT` keyword specifies the value 1.

We can modify the preceding SQL query to find any range of values, starting from any position in a numerically sorted set of values. For example, the following SQL query finds the sixth, seventh, and eighth largest values in the `weather` table:

```
SELECT temper
FROM weather
ORDER BY temper DESC
LIMIT 3 OFFSET 5;
+--------+
| temper |
+--------+
|     45 |
|     45 |
|     42 |
+--------+
3 rows in set (0.000 sec)
```

If need be, you can manually confirm that the preceding SQL query does return the correct set of values.

STRING FUNCTIONS IN SQL

The following SQL statement illustrates how to use the `UCASE()` function to convert the `item_desc` values to uppercase in the `item_desc` field of the table `new_items`:

```
SELECT UCASE(item_desc), item_desc
FROM new_items;
```

Execute the preceding SQL statement to see the following output:

```
+------------------+----------------+
| UCASE(item_desc) | item_desc      |
+------------------+----------------+
| HAMMER           | hammer         |
| SCREWDRIVER      | screwdriver    |
| WRENCH           | wrench         |
| PLIERS           | pliers         |
| BALLPEEN         | ballpeen       |
| 1/4 INCH NAILS   | 1/4 inch nails |
| TOOLBOX S        | Toolbox S      |
| TOOLBOX M        | Toolbox M      |
| TOOLBOX L        | Toolbox L      |
| HANDSAW          | Handsaw        |
+------------------+----------------+
10 rows in set (0.004 sec)
```

The following SQL statement illustrates how to use the `LCASE()` function to convert the `item_desc` values to lowercase in the `item_desc` field of the `new_items` table:

```
SELECT LCASE(item_desc), item_desc
FROM new_items;
```

Launch the preceding SQL statement to see the following output:

```
+------------------+----------------+
| LCASE(item_desc) | item_desc      |
+------------------+----------------+
| hammer           | hammer         |
| screwdriver      | screwdriver    |
| wrench           | wrench         |
| pliers           | pliers         |
| ballpeen         | ballpeen       |
| 1/4 inch nails   | 1/4 inch nails |
| toolbox s        | Toolbox S      |
| toolbox m        | Toolbox M      |
| toolbox l        | Toolbox L      |
| handsaw          | Handsaw        |
+------------------+----------------+
10 rows in set (0.001 sec)
```

The `MID()` function extracts substrings from string values in a table. Specify the attribute name, the start column, and an optional length:

```
SELECT MID(item_desc,3,4) AS short_desc
FROM new_items;
+------------+
| short_desc |
+------------+
| mmer       |
| rewd       |
| ench       |
| iers       |
| llpe       |
| 4 in       |
| olbo       |
| olbo       |
| olbo       |
| ndsa       |
+------------+
10 rows in set (0.000 sec)
```

The `SUBSTR()` function is similar to the `MID()` function:

```
SELECT SUBSTRING(item_desc,2,4) AS short_desc
FROM new_items
WHERE item_price > 10;
+------------+
| short_desc |
+------------+
| amme       |
| allp       |
| oolb       |
| oolb       |
| oolb       |
| ands       |
+------------+
6 rows in set (0.000 sec)
```

The following SQL statement illustrates how to use the `CONCAT()` function to concatenate two strings:

```
SELECT CONCAT("I ", "Love ", "Pizza") AS PizzaLine;
+--------------+
| PizzaLine    |
+--------------+
| I Love Pizza |
+--------------+
1 row in set (0.002 sec)
```

A more useful example of the `CONCAT()` function is shown here:

```
SELECT CONCAT(first_name, " ", last_name, " ", home_address ) AS Address
FROM customers;
+------------------------+
| Address                |
+------------------------+
| John Smith 123 Main St |
+------------------------+
1 row in set (0.001 sec)
```

One common task involves capitalizing the first letter of a string, which you can accomplish by a combination of `CONCAT()`, `UCASE()`, and `SUBSTRING()`, as shown here:

```
SELECT item_desc, CONCAT(UCASE(LEFT(item_desc, 1)),
SUBSTRING(item_desc, 2)) AS UPPERFIRST
FROM new_items;
+----------------+----------------+
| item_desc      | UPPERFIRST     |
+----------------+----------------+
| hammer         | Hammer         |
| screwdriver    | Screwdriver    |
| wrench         | Wrench         |
| pliers         | Pliers         |
| ballpeen       | Ballpeen       |
| 1/4 inch nails | 1/4 inch nails |
| Toolbox S      | Toolbox S      |
| Toolbox M      | Toolbox M      |
| Toolbox L      | Toolbox L      |
| Handsaw        | Handsaw        |
+----------------+----------------+
10 rows in set (0.000 sec)
```

Alternatively, you can use the `MID()` function:

```
SELECT CONCAT(UCASE(MID(item_desc,1,1)),MID(item_desc,2))
AS descr
FROM new_items;
+----------------+
| descr          |
+----------------+
| Hammer         |
| Screwdriver    |
| Wrench         |
| Pliers         |
| Ballpeen       |
| 1/4 inch nails |
| Toolbox S      |
| Toolbox M      |
| Toolbox L      |
| Handsaw        |
+----------------+
10 rows in set (0.001 sec)
```

SQL QUERIES WITH THE SUBSTRING() FUNCTION

This section shows you an assortment of SQL queries that involve either the `substr()` function, the `ROWID`, or both. The first step involves launching the `schedule.sql` script that creates the `schedule` table, which is displayed in Listing 4.4.

LISTING 4.4: schedule.sql

```
USE DATABASE mytools;

DROP TABLE IF EXISTS schedule;
CREATE TABLE schedule (year VARCHAR(4), term VARCHAR(10),
student_id VARCHAR(20), course_id VARCHAR(20));

INSERT INTO schedule VALUES ('2020','SPRING','1010','5010');
INSERT INTO schedule VALUES ('2020','SPRING','1020','5010');
INSERT INTO schedule VALUES ('2020','SUMMER','1020','5020');
INSERT INTO schedule VALUES ('2020','SUMMER','1020','5030');
INSERT INTO schedule VALUES ('2020','FALL',  '1030','5040');
INSERT INTO schedule VALUES ('2020','FALL',  '1030','5050');
INSERT INTO schedule VALUES ('2020','FALL',  '1040','6000');
INSERT INTO schedule VALUES ('2020','FALL',  '1040','7000');
INSERT INTO schedule VALUES ('2020','FALL',  '1050','6000');
INSERT INTO schedule VALUES ('2020','FALL',  '1050','7000');
INSERT INTO schedule VALUES ('2020','FALL',  '1060','6000');
INSERT INTO schedule VALUES ('2020','FALL',  '1060','7000');
```

Listing 4.4 shows the `schedule` table is dropped (if it already exists) and then re-created with the `CREATE TABLE` statement. The next portion of Listing 4.4 inserts multiple rows of data into the `schedule` table by invoking a set of `INSERT` statements.

Next, display all the rows in the `schedule` table by invoking the SQL statement shown in bold below:

```
MySQL [mytools]> select * from schedule;
+------+--------+------------+-----------+
| year | term   | student_id | course_id |
+------+--------+------------+-----------+
| 2020 | SPRING | 1010       | 5010      |
| 2020 | SPRING | 1020       | 5010      |
| 2020 | SUMMER | 1020       | 5020      |
| 2020 | SUMMER | 1020       | 5030      |
| 2020 | FALL   | 1030       | 5040      |
| 2020 | FALL   | 1030       | 5050      |
| 2020 | FALL   | 1040       | 6000      |
| 2020 | FALL   | 1040       | 7000      |
| 2020 | FALL   | 1050       | 6000      |
| 2020 | FALL   | 1050       | 7000      |
| 2020 | FALL   | 1060       | 6000      |
| 2020 | FALL   | 1060       | 7000      |
+------+--------+------------+-----------+
12 rows in set (0.001 sec)
```

The SUBSTRING() Function in SQL

The following SQL statement shows you how to use the `substring()` function to return the left-most three characters of the `term` attribute:

```
select substring(term,1,3) from schedule;
+---------------------+
| substring(term,1,3) |
+---------------------+
| SPR                 |
| SPR                 |
| SUM                 |
| SUM                 |
| FAL                 |
| FAL                 |
| FAL                 |
| FAL                 |
| FAL                 |
| FAL                 |
| FAL                 |
| FAL                 |
+---------------------+
12 rows in set (0.000 sec)
```

The following SQL statement returns the left-most three characters of the `term` attribute for the student whose `student_id` is 1020:

```
select substring(term,1,3)
from schedule
where student_id = 1020;
+---------------------+
| substring(term,1,3) |
+---------------------+
| SPR                 |
| SUM                 |
| SUM                 |
+---------------------+
3 rows in set (0.002 sec)
```

BOOLEAN OPERATORS IN SQL

This section contains Boolean operations in SQL, some of which you have already seen earlier in this chapter (and perhaps in other programming languages as well):

- `AND` combines Boolean expressions for filtering data
- `OR` combines Boolean expressions for filtering data
- `IN` determines if a value matches any value in a list or a subquery
- `BETWEEN` queries data based on a range
- `LIKE` queries data based on a pattern
- `LIMIT` constrains the number of rows returned by `SELECT` statement
- `IS NULL` checks whether a value is `NULL`

Here is an example of a SQL statement that contains a `BETWEEN` condition:

```
SELECT last_name, salary
FROM employees
WHERE salary BETWEEN 5000 AND 6000;
```

Here is an example of a SQL statement that contains a `LIKE` condition:

```
SELECT emp_id, title
FROM employees
WHERE title LIKE 'D%';
+--------+-------------+
| emp_id | title       |
+--------+-------------+
|   1000 | Developer   |
|   3000 | Dev Manager |
+--------+-------------+
2 rows in set (0.000 sec)
```

Here is an example of a SQL statement that checks for `NULL` values:

```
=> test for nulls with the IS NULL operator:
SELECT last_name, manager_id
FROM employees
WHERE manager_id IS NULL;
```

The IN Keyword

Here is an example of a SQL statement that contains an `IN` condition in order to find rows whose `manager_id` is in a list of values:

```
SELECT employee_id, last_name, salary, manager_id
FROM employees
WHERE manager_id IN (100, 101, 201);
```

Earlier in this chapter, you saw an example of finding the maximum temperature in the `weather` table:

```
SELECT MAX(temper)
FROM weather;
+-------------+
| MAX(temper) |
+-------------+
|          78 |
+-------------+
1 row in set (0.003 sec)
```

However, the preceding SQL query only returns the maximum temperature: it does not tell us *how many rows* have the maximum temperature. The solution involves the `IN` keyword, as shown here:

```
SELECT * FROM weather
    WHERE temper IN (
      SELECT MAX(temper) FROM weather);
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
+------------+--------+------+----------+------+-------+
2 rows in set (0.011 sec)
```

SET OPERATORS IN SQL

SQL supports the following set-related operators, each of which is illustrated later in this section via a SQL statement:

- `INTERSECT`
- `MINUS`
- `UNION`
- `UNION ALL`

Conceptually these operators work the same way as sets in mathematics. The *intersection* of sets `S1` and `S2` is the (possibly empty) subset of elements that are common to *both* `S1` and `S2`. The *difference* `S1 - S2` is the set of elements that are in `S1` that are *not* in the set `S2`.

Similarly, the *union* of sets `S1` and `S2` is the set of elements that belong to *either* `S1` or `S2`. The `UNION` keyword combines rows from multiple queries (which can involve tables or views) and the result set contains unique rows. If you want to include duplicate rows in the result set, use `UNION ALL` instead of `UNION`.

Before we look at SQL statements that contain these keywords, let's create two tables, `t1` and `t2`, and populate them with data, as shown below:

```
DROP TABLE IF EXISTS t1;
DROP TABLE IF EXISTS t2;

CREATE TABLE t1 (id INT PRIMARY KEY);
CREATE TABLE t2 (id INT PRIMARY KEY);

INSERT INTO t1 VALUES (1),(2),(3);
INSERT INTO t2 VALUES (2),(3),(4);
```

The following SQL statement returns the *intersection* of the rows in `t1` and `t2`:

```
(SELECT id FROM t1)
INTERSECT
(SELECT id FROM t2);
```

The following SQL statement returns the *union* of `t1` and `t2`:

```
(SELECT id FROM t1)
UNION
(SELECT id FROM t2);
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
+----+
4 rows in set (0.001 sec)
```

The following SQL statement returns the *difference* (via the MINUS keyword) between table t1 and table t2: (i.e., t1 - t2)

```
SELECT id FROM t1
MINUS
SELECT id FROM t2;
```

AND, OR, AND NOT OPERATORS IN SQL

SQL supports the AND, OR, and NOT operators that operate in the same fashion as those operators in programming languages. Specifically, the AND operator requires both conditions to be true, an example of which is shown here:

```
SELECT *
FROM employees
WHERE emp_id > 1000
AND emp_id = mgr_id;
+--------+--------+--------------------+
| emp_id | mgr_id | title              |
+--------+--------+--------------------+
|   4000 |   4000 | Senior Dev Manager |
+--------+--------+--------------------+
1 row in set (0.001 sec)
```

The OR operator returns the rows that satisfy *any* condition(s) in the OR portion of the SQL statement, an example of which is shown here:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE salary >= 10000
OR    job_id LIKE '%MAN%'
```

You can combine the OR operator with an AND operator, as shown in the following example:

```
SELECT *
FROM employees
WHERE title = 'Developer'
OR emp_id = 2000
OR emp_id = 4000 and mgr_id = 4000;
+--------+--------+--------------------+
| emp_id | mgr_id | title              |
+--------+--------+--------------------+
|   1000 |   2000 | Developer          |
|   2000 |   3000 | Project Lead       |
|   4000 |   4000 | Senior Dev Manager |
+--------+--------+--------------------+
3 rows in set (0.000 sec)
```

The NOT operator requires the opposite condition to be true, as shown here:

```
SELECT *
FROM employees
WHERE title
```

```
NOT IN ('SALES', 'MKTG')
AND emp_id >= 2000;
+--------+--------+--------------------+
| emp_id | mgr_id | title              |
+--------+--------+--------------------+
|   2000 |   3000 | Project Lead       |
|   3000 |   4000 | Dev Manager        |
|   4000 |   4000 | Senior Dev Manager |
+--------+--------+--------------------+
3 rows in set (0.000 sec)
```

The preceding SQL statements contain >= to indicate "greater than or equal to," which is one type of inequality. A more extensive list of inequalities is shown here, each of which can be used in the earlier SQL statements:

- `>= specifies "greater than or equal to"`
- `>  specifies "greater than"`
- `=  specifies "equal to"`
- `<= specifies "less than or equal to"`
- `<  specifies "less than"`
- `<> specifies "not equal to"`

WORKING WITH ARITHMETIC OPERATORS

SQL allows you to use the addition (+) operator to calculate the sum of two or more numeric values, an example of which is shown here:

```
SELECT 7 + 13  as my_sum;
+--------+
| my_sum |
+--------+
|     20 |
+--------+
1 row in set (0.000 sec)
```

An example of adding three numbers is here:

```
SELECT 7 + 13 + 25.123 as my_sum2;
+---------+
| my_sum2 |
+---------+
|  45.123 |
+---------+
1 row in set (0.000 sec)
```

Update an integer-valued attribute in a table, as shown here:

```
SELECT SALARY+10000 as new_salary FROM EMPLOYEES;
```

Add a numeric value to the `emp_id` column using the addition operator, as shown in this query:

```
SELECT emp_id+10000 as new_emp_id
FROM employees;
+------------+
| new_emp_id |
+------------+
|      11000 |
|      12000 |
|      13000 |
|      14000 |
+------------+
4 rows in set (0.000 sec)
```

SQL supports the arithmetic operator "-" for subtraction, as shown here:

```
SELECT 260-99 as Subtract;
+----------+
| Subtract |
+----------+
|      161 |
+----------+
1 row in set (0.000 sec)
```

```
SELECT emp_id-100 as Subtracted_id FROM EMPLOYEES;
+---------------+
| Subtracted_id |
+---------------+
|           900 |
|          1900 |
|          2900 |
|          3900 |
+---------------+
4 rows in set (0.001 sec)
```

SQL supports the arithmetic operator "*" for multiplication, as shown here:

```
SELECT 100*77 as Multiplication;
+----------------+
| Multiplication |
+----------------+
|           7700 |
+----------------+
1 row in set (0.000 sec)
```

SQL supports the arithmetic operator "/" for division, as shown here:

```
SELECT 15/6 as Division;
+----------+
| Division |
+----------+
|   2.5000 |
+----------+
1 row in set (0.000 sec)
```

SQL supports the arithmetic operator "%" for modulus, as shown here:

```
SELECT 23%4 as result;
+---------+
| result |
+---------+
|      3 |
+---------+
1 row in set (0.000 sec)

SELECT emp_id, emp_id%3 as result FROM EMPLOYEES;
+--------+--------+
| emp_id | result |
+--------+--------+
|   1000 |      1 |
|   2000 |      2 |
|   3000 |      0 |
|   4000 |      1 |
+--------+--------+
4 rows in set (0.000 sec)
```

ARITHMETIC AGGREGATE OPERATORS IN SQL

SQL supports aggregate arithmetic functions such as `max()`, `min()`, and `avg()` for finding the maximum, minimum, and average, respectively, of the values in a numeric column. The next set of SQL queries displays the rows in the `item_desc` table followed by SQL statements that contain the above-mentioned arithmetic aggregate functions.

```
DESC item_desc;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| item_id    | int          | YES  |     | NULL    |       |
| item_desc  | varchar(80)  | YES  |     | NULL    |       |
| item_price | decimal(8,2) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
3 rows in set (0.017 sec)

SELECT *
FROM item_desc;
+---------+-------------+------------+
| item_id | item_desc   | item_price |
+---------+-------------+------------+
|     100 | hammer      |      20.00 |
|     200 | screwdriver |       8.00 |
|     100 | wrench      |      10.00 |
+---------+-------------+------------+
3 rows in set (0.001 sec)

SELECT max(item_price) as item_price
FROM item_desc;
```

```
+------------+
| item_price |
+------------+
|      20.00 |
+------------+
1 row in set (0.007 sec)

SELECT max(item_price) maxp, min(item_price) as minp
FROM item_desc;
+-------+------+
| maxp  | minp |
+-------+------+
| 20.00 | 8.00 |
+-------+------+
1 row in set (0.000 sec)
```

As you can see, the preceding SQL statements retrieve a single value for the maximum and minimum of the `price` column of the `item_desc` table.

However, the next set of SQL statements return the full details of the row that contains the maximum or minimum value in the `price` column of the `item_desc` table.

```
SELECT *
FROM item_desc
WHERE item_price = (select max(item_price)
                    FROM item_desc);
+---------+-----------+------------+
| item_id | item_desc | item_price |
+---------+-----------+------------+
|     100 | hammer    |      20.00 |
+---------+-----------+------------+
1 row in set (0.003 sec)

SELECT *
FROM item_desc
WHERE item_price = (SELECT min(item_price)
                    FROM item_desc);
+---------+-------------+------------+
| item_id | item_desc   | item_price |
+---------+-------------+------------+
|     200 | screwdriver |       8.00 |
+---------+-------------+------------+
1 row in set (0.001 sec)
```

Finding Average Values

The following SQL statement determines the average price of the items in the `item_desc` table:

```
SELECT max(item_price) maxp,
       avg(item_price) as avgp,
       min(item_price) as minp
FROM item_desc;
```

```
+-------+-----------+------+
| maxp  | avgp      | minp |
+-------+-----------+------+
| 20.00 | 12.666667 | 8.00 |
+-------+-----------+------+
1 row in set (0.000 sec)
```

Although you might be tempted to replace the `min()` or `max()` function with the `avg()` function, the result will typically be the empty set. Indeed, how often will the average value appear as a row? Let's see what happens in our case:

```
MySQL [mytools]> SELECT * FROM item_desc
     WHERE item_price = (SELECT avg(item_price) FROM item_desc);
Empty set (0.001 sec)
```

Note that you can also replace `SELECT * FROM item_desc` with a sublist of columns from the `item_desc` table.

The following SQL statement calculates the average monthly temperature of the rows in the weather table:

```
SELECT YEAR(day) year, MONTH(day) month, AVG(temper) average
FROM weather
WHERE MONTH(day) IN (1,2,3,4,5,6,7,8,9,10,11,12)
GROUP BY YEAR(day), MONTH(day)
ORDER BY YEAR(day), MONTH(day), AVG(temper);
+------+-------+---------+
| year | month | average |
+------+-------+---------+
| 2021 |     4 | 55.0000 |
| 2021 |     7 | 55.0000 |
| 2021 |     8 | 50.5000 |
| 2021 |     9 | 34.0000 |
+------+-------+---------+
4 rows in set (0.001 sec)
```

The first column in the preceding result set contains only the value 2021 because all the rows in the weather table consist of data from the year 2021. However, the preceding SQL statement will work correctly with data from multiple years.

SELECT Clauses with Multiple Aggregate Functions

This section contains SQL statements that contain the `max()` and `min()` functions in the `SELECT` clause. For example, the following SQL statement displays the maximum difference in temperature in the `weather` table:

```
SELECT MAX(temper) - MIN(temper) as delta
FROM weather;
+-------+
| delta |
+-------+
|    63 |
+-------+
1 row in set (0.000 sec)
```

The following SQL statement displays the maximum difference in temperature during the month of April in the `weather` table:

```
SELECT MAX(temper) - MIN(temper) as delta
FROM weather
WHERE MONTH(day) = 04;
+-------+
| delta |
+-------+
|    36 |
+-------+
1 row in set (0.001 sec)
```

THE ORDER BY CLAUSE IN SQL

You have already seen SQL statements in this chapter that specify the `ORDER BY` clause in order to specify the order in which the result set is displayed. The two options are ascending order or descending order, which can be performed with alphabetic values or numeric values.

This section contains an assortment of SQL statements that also specify the `ORDER BY` clause. For example, the following SQL statements order the output in increasing order (the default) and then in decreasing order.

```
SELECT *
FROM employees
ORDER BY title;
+--------+--------+--------------------+
| emp_id | mgr_id | title              |
+--------+--------+--------------------+
|   3000 |   4000 | Dev Manager        |
|   1000 |   2000 | Developer          |
|   2000 |   3000 | Project Lead       |
|   4000 |   4000 | Senior Dev Manager |
+--------+--------+--------------------+
4 rows in set (0.000 sec)

SELECT *
FROM employees
ORDER BY title DESC;
+--------+--------+--------------------+
| emp_id | mgr_id | title              |
+--------+--------+--------------------+
|   4000 |   4000 | Senior Dev Manager |
|   2000 |   3000 | Project Lead       |
|   1000 |   2000 | Developer          |
|   3000 |   4000 | Dev Manager        |
+--------+--------+--------------------+
4 rows in set (0.000 sec)
```

Sort a table by specifying multiple columns in the `ORDER BY` clause, as shown here:

```
SELECT *
FROM employees
```

```
ORDER BY title, mgr_id;
+--------+--------+--------------------+
| emp_id | mgr_id | title              |
+--------+--------+--------------------+
|   3000 |   4000 | Dev Manager        |
|   1000 |   2000 | Developer          |
|   2000 |   3000 | Project Lead       |
|   4000 |   4000 | Senior Dev Manager |
+--------+--------+--------------------+
4 rows in set (0.000 sec)
```

In the preceding SQL statement, the inclusion of the `mgr_id` has no effect because the rows have unique values for `mgr_id` and `title`. However, the following SQL statements show you that the order of the attributes can make a difference.

```
SELECT *
FROM weather
WHERE forecast != '' AND city != ''
ORDER BY forecast,city;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
+------------+--------+------+----------+------+-------+
5 rows in set (0.001 sec)

SELECT *
FROM weather
WHERE forecast != '' AND city != ''
ORDER BY city,forecast;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
+------------+--------+------+----------+------+-------+
5 rows in set (0.001 sec)
Note: the ORDER BY clause must be the last in a SELECT
statement
```

ORDER BY with Aggregate Functions

You can also define SQL statements that combine aggregate functions with the `ORDER BY` clause, as shown here:

```
SELECT day, temper, AVG(temper)
FROM weather
```

```
GROUP BY day, temper
ORDER BY AVG(temper) DESC;
+------------+--------+-------------+
| day        | temper | AVG(temper) |
+------------+--------+-------------+
| 2021-04-03 |     78 |     78.0000 |
| 2021-07-03 |     78 |     78.0000 |
| 2021-08-06 |     51 |     51.0000 |
| 2021-08-04 |     50 |     50.0000 |
| 2021-04-02 |     45 |     45.0000 |
| 2021-07-02 |     45 |     45.0000 |
| 2021-09-02 |     45 |     45.0000 |
| 2021-04-01 |     42 |     42.0000 |
| 2021-07-01 |     42 |     42.0000 |
| 2021-09-01 |     42 |     42.0000 |
| 2021-09-03 |     15 |     15.0000 |
+------------+--------+-------------+
11 rows in set (0.003 sec)
```

The following SQL statement displays the maximum difference in temperature during the months of August and September in the `weather` table, using the `GROUP BY` clause and the `ORDER BY` clause for the month value:

```
SELECT MONTH(day), MAX(temper) - MIN(temper) as delta
FROM weather
WHERE MONTH(day) IN (08,09)
GROUP BY MONTH(day)
ORDER BY MONTH(day);
+------------+-------+
| MONTH(day) | delta |
+------------+-------+
|          8 |     1 |
|          9 |    30 |
+------------+-------+
2 rows in set (0.003 sec)
```

LARGEST DISTINCT VALUES AND FREQUENCY OF VALUES

In Chapter 3, you learned how to find the second largest and third largest distinct values in the `weather` table. In this section, you will learn how to select the three largest values in the `weather` table using the `ORDER BY` clause instead of a subquery.

As a quick reminder, there is a difference between “select the largest three values” and “select the largest three *distinct* values.” By default, SQL statements that select the largest values allow for duplicate values. In this section, you will see how to write SQL statements that select distinct maximum values.

Let’s look at some SQL statements that might seem to be the solution, but they do not produce the desired results (i.e., distinct values). For example, the following SQL statement is incorrect because the selected temperature values are not selected from a list of temperatures in *descending* order:

```
SELECT temper
FROM weather
LIMIT 3;
+--------+
| temper |
+--------+
|     42 |
|     45 |
|     78 |
+--------+
3 rows in set (0.001 sec)
```

The following SQL statement is incorrect because the selected temperatures contain *duplicate* values:

```
SELECT temper
FROM weather
ORDER BY temper DESC
LIMIT 3;
+--------+
| temper |
+--------+
|     78 |
|     78 |
|     51 |
+--------+
3 rows in set (0.000 sec)
```

The following SQL statement is correct because the selected temperatures are distinct and they are selected from a descending list of temperatures:

```
SELECT DISTINCT(temper)
FROM weather
ORDER BY temper DESC
LIMIT 3;
+--------+
| temper |
+--------+
|     78 |
|     51 |
|     50 |
+--------+
3 rows in set (0.001 sec)
```

As an additional observation, the following SQL queries return only the largest value instead of the top two values:

```
SELECT MAX(temper)
FROM weather
LIMIT 2;

SELECT MAX(DISTINCT(temper))
FROM weather
LIMIT 2;
```

The following SQL statement displays the most frequently occurring value for `temper` in the `weather` table:

```
SELECT temper, COUNT(*)
FROM weather
GROUP BY temper
ORDER BY COUNT(*) DESC
LIMIT 1;
+--------+------------+
| temper | COUNT( * ) |
+--------+------------+
|     42 |          3 |
+--------+------------+
1 row in set (0.000 sec)
```

The following SQL statement displays the frequency of the values in the `state` attribute in the `weather` table:

```
SELECT state, occurrences
FROM (SELECT state,count(*) as occurrences
      FROM weather
      GROUP BY state
      ) T1;
+-------+-------------+
| state | occurrences |
+-------+-------------+
| ca    |           7 |
| wa    |           1 |
| mn    |           2 |
| il    |           1 |
+-------+-------------+
4 rows in set (0.002 sec)
```

The following SQL statement is a variation of the preceding SQL statement that also includes the `LIMIT 1` clause in order to display the most frequently occurring value in the `state` in attribute in the `weather` table:

```
SELECT state, occurrences
FROM (SELECT state,count(*) as occurrences
      FROM weather
      GROUP BY state
      LIMIT 1
      ) T1;
+-------+-------------+
| state | occurrences |
+-------+-------------+
| ca    |           7 |
+-------+-------------+
1 row in set (0.003 sec)
```

CHARACTER FUNCTIONS AND STRING OPERATORS

There are two main types of SQL functions: *single-row functions* that return one result per row and *multiple-row functions* that return one result per set of rows. Specifically, single-row functions in SQL will

- manipulate data items
- accept arguments and return one value
- act on each row that is returned
- return one result per row
- may modify the data type
- can be nested
- accept arguments that can be a column or an expression

SQL Character Functions

There two types of character functions: case-manipulation functions and character manipulation functions. Case manipulation functions in SQL include the following:

- `LOWER`
- `UPPER`
- `INITCAP`

Character manipulation functions in SQL include the following built-in functions:

- `SUBSTRING`
- `LENGTH`
- `INSTR`
- `LPAD | RPAD`
- `TRIM`
- `REPLACE`

Following this section are some one-line examples of some of the preceding built-in functions, where you need to replace `my_table` with a suitable table name and replace `fname` and `phone_number` with attributes from your table in your database:

Remove leading spaces:

```
SELECT LTRIM(fname) from my_table;
```

Remove leading *and* trailing spaces:

```
SELECT TRIM(fname) from my_table;
```

Replace "-" with a space (" "):

```
SELECT fname, REPLACE(phone_number, '-', ' ') as p_number
FROM my_table;
```

SQL supports built-in number functions, include the following functions:

- `ROUND`
- `TRUNC`
- `MOD`

An example of the built-in `truncate()` function (which is *different* from the `TRUNCATE` keyword) is as follows:

-- the value 12.345 is replaced with 12:

```
SELECT TRUNCATE (average, 0) from my_table;
```

String Operators in SQL

SQL supports the following string operators that perform the concatenation of strings and partial matches of strings against meta characters:

- CONCAT (concatenation)
- LIKE operator

```
SELECT 'Hello' + ' ' + 'World!' AS StringConcatenated;
SELECT FIRSTNAME + ' ' + LASTNAME AS ConcatenatedName FROM STUDENTS;
```

SQL provides a concatenation operator that does the following:

- links columns or character strings to other columns
- is represented by two vertical bars (||)
- creates a resultant column that is a character expression

For example, the following SQL statement concatenates the `last_name` field with the `job_id` field for each row in the `employees` table:

```
SELECT last_name||job_id AS "Employees"
FROM   employees;
```

Literal character strings can be a character, a number, or a date, and they have the following properties:

- A literal is included in the `SELECT` statement.
- Dates and characters must be enclosed by single quotation marks.
- Each character string is output once for each row returned.

You can also specify an alternative quote (q) operator:

- choose any delimiter
- useful for increasing readability and usability

The `LIKE` keyword supports the percent `(%)` meta character as well as the underscore `(_)` meta character, where the latter matches any single character.

THE MATCH() FUNCTION AND TEXT SEARCH

Listing 4.5 shows the content of `nlp_terms.sql` that illustrates how to use the `MATCH()` function to search for text in a database table.

LISTING 4.5: nlp_terms.sql

```
use mytools;
DROP TABLE IF EXISTS nlp_terms;

-- create table:
CREATE TABLE nlp_terms (
          id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
          nlp_term VARCHAR(200),
          definition TEXT,
          FULLTEXT (nlp_term,definition)
        ) ENGINE=InnoDB;

-- insert data into table:
INSERT INTO nlp_terms (nlp_term,definition) VALUES
        ('lemmatization','Word Root Words'),
        ('nltk',         'NLP Toolkit From Stanford'),
        ('SpaCy',        'Very Good NLP toolkit'),
        ('stemming',     'Truncates Word Suffixes'),
        ('stopwords',    'Common Words'),
        ('word2vec',     'CBoW and Skip Grams');

-- select data:
SELECT * FROM nlp_terms
        WHERE MATCH (nlp_term,definition)
        AGAINST ('NLP' IN NATURAL LANGUAGE MODE);
```

Listing 4.5 starts by creating and populating the table `nlp_terms` with a set of rows containing text, followed by a SQL statement that uses the `MATCH()` function to search for the term `NLP` in the `nlp_terms` table. Launch the code in Listing 4.5 to see the following output:

```
+----+----------+---------------------------------+
| id | nlp_term | definition                      |
+----+----------+---------------------------------+
| 1  | nltk     | NLP Toolkit From Stanford       |
| 2  | SpaCy    | Very Good NLP Toolkit           |
+----+----------+---------------------------------+
2 rows in set (0.000 sec)
```

CTES AND THE "WITH" KEYWORD IN MYSQL (VERSION 8)

A *common table expression* (CTE) is a temporary named result set. A CTE is defined within the execution scope of a single `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `CREATE VIEW` statement.

To define a CTE, you need to specify a `with` statement, which is available in MySQL 8 (but not earlier versions of MySQL). In fact, you can specify multiple blocks of SQL statements that can include various SQL keywords, such as `GROUP BY`, and aggregate functions such as `MIN()` and `MAX()`. Interestingly, the `with` statement can be used to define recursive SQL queries (discussed later). The definition of a CTE has three parts:

- the `with` keyword
- the name the CTE
- the body of the CTE

Here is a sample syntax for constructing a single CTE, followed by a SQL statement that references the CTE:

```
WITH simple_name(column-list) AS (
  YOUR-SQL-QUERY
)
SELECT * FROM simple_name;
```

The following example illustrates how to define a single CTE that specifies the `emp_id` attribute of the `employees` table, and notice that the inner SQL statement does *not* contain a semi-colon:

```
WITH emps(emp_id) AS
(
  SELECT emp_id FROM employees
)
SELECT * FROM emps;
+--------+
| emp_id |
+--------+
|   1000 |
|   2000 |
|   3000 |
|   4000 |
+--------+
4 rows in set (0.001 sec)
```

The next example shows you how to specify multiple attributes in a CTE using a syntax that is slightly different from the preceding CTE:

```
WITH emps AS
(
  SELECT emp_id, mgr_id FROM employees
)
SELECT emp_id FROM emps
WHERE emp_id > 1000;
+--------+
| emp_id |
+--------+
|   2000 |
|   3000 |
|   4000 |
+--------+
3 rows in set (0.001 sec)
```

The next sample shows you how to construct a CTE that contains a `JOIN` keyword:

```
WITH purch_orders AS (
  SELECT cust_id, po_id FROM purchase_orders
)
SELECT cust_id
FROM customers
JOIN purch_orders USING(cust_id);
+---------+
| cust_id |
+---------+
|    1000 |
|    1000 |
|    1000 |
+---------+
3 rows in set (0.002 sec)
```

You can also define a CTE that specifies multiple `WITH` code blocks using the following syntax:

```
WITH simple_name1 AS (
  YOUR-SQL-QUERY1
),
WITH simple_name2 AS (
  YOUR-SQL-QUERY2
)
SELECT * FROM simple_name1 JOIN simple_name2 ON some-condition;
```

The CTE examples in this section contain a `SELECT` keyword, and you can define CTE expressions with other keywords, as outlined here:

```
WITH ... INSERT ...
WITH ... UPDATE ...
WITH ... DELETE ...
```

Consult the online documentation for MySQL 8 for additional information regarding CTEs.

The with Keyword and a Recursive SQL Query

The SQL file `recursive.sql` defines a recursive SQL statement that displays the integers from 1 to 6 inclusive, as follows:

```
WITH RECURSIVE arith_seq AS
(
  SELECT 1 AS x
  UNION ALL
  SELECT 1+x FROM arith_seq WHERE x<6
)
SELECT * FROM arith_seq;
+------+
| x    |
+------+
|    1 |
|    2 |
|    3 |
```

```
|     4 |
|     5 |
|     6 |
+-------+
6 rows in set (0.000 sec)
```

CTES AND THE MEAN, STDDEV, AND Z-SCORES

Listing 4.6 shows the content of `my_stats_data.sql` that creates and populates the table `my_stats_data` with numeric values, followed by SQL statements to calculate the mean, standard deviation, and z-scores of the rows in this table.

LISTING 4.6: my_stats_data.sql

```
use mytools;

DROP TABLE IF EXISTS my_stats_data;
CREATE TABLE my_stats_data ( num_val INT(4));

INSERT INTO my_stats_data VALUES (2);
INSERT INTO my_stats_data VALUES (5);
INSERT INTO my_stats_data VALUES (7);
INSERT INTO my_stats_data VALUES (9);
INSERT INTO my_stats_data VALUES (9);
INSERT INTO my_stats_data VALUES (37);

-- Find the mean with this SQL statement:
\! echo '=> Calculate the mean:';
SELECT AVG(num_val)
FROM my_stats_data;

-- Find the standard deviation with this SQL statement:
\! echo '=> Calculate the standard deviation:';
SELECT STD(num_val)
FROM my_stats_data;

-- Find the z-score with this SQL statement:
\! echo '=> Calculate the z-scores:';
WITH simple_stats as
    (SELECT AVG(num_val) as mean,
            STDDEV(num_val) as sd
    FROM my_stats_data)
SELECT num_val, (num_val-simple_stats.mean) / simple_stats.
sd as z_score
FROM my_stats_data, simple_stats;

-- Find z-scores greater than 2 with this SQL statement:
\! echo '=> Find the z-scores greater than 2:';
WITH simple_stats as
    (SELECT AVG(num_val) as mean,
            STDDEV(num_val) as sd
    FROM my_stats_data)
```

```
SELECT num_val, (num_val-simple_stats.mean) / simple_stats.
sd as z_score
FROM my_stats_data, simple_stats
HAVING z_score > 2;
```

Listing 4.6 starts by creating and then populating the `my_stats_data` table with data. The next portion of Listing 4.6 contains a SQL statement for calculating the mean of the value, followed by a SQL statement that calculates the standard deviation.

The third SQL statement defines a CTE with the mean and standard deviation in order to calculate the standardized values of the numbers in the `my_stats_data` table. The fourth and final SQL statement modifies the third `SQL` statement by adding the following code snippet to detect (potential) outliers:

```
HAVING z_score > 2;
```

You can replace the value 2 with whatever value is appropriate for detecting outliers in a database table. Note that only the fourth SQL statement is required for detecting outliers. The other three SQL statements are included for your convenience. Launch the code in Listing 4.6 to see the following output:

```
=> Calculate the mean:
+--------------+
| AVG(num_val) |
+--------------+
|      11.5000 |
+--------------+
1 row in set (0.000 sec)

=> Calculate the standard deviation:
+--------------------+
| STD(num_val)       |
+--------------------+
| 11.658330355015108 |
+--------------------+
1 row in set (0.000 sec)

=> Calculate the z-scores:
+---------+----------------------+
| num_val | z_score              |
+---------+----------------------+
|       2 |  -0.8148679708594249 |
|       5 |  -0.557541243219606 5 |
|       7 | -0.38599009145972757 |
|       9 | -0.21443893969984865 |
|       9 | -0.21443893969984865 |
|      37 |   2.1872771849384565 |
+---------+----------------------+
6 rows in set (0.001 sec)
```

```
=> Find the z-scores greater than 2:
+---------+--------------------+
| num_val | z_score            |
+---------+--------------------+
|      37 | 2.1872771849384565 |
+---------+--------------------+
1 row in set (0.000 sec)
```

LINEAR REGRESSION IN SQL

Linear regression is a standard task in statistics, and if you are a data scientist or machine learning engineer, you are most likely already familiar with the calculations to determine the slope and y-intercept of the best fitting line. If you have forgotten some of those details, you can review them by reading the code in this section.

Listing 4.7 shows the content of `linear_regression.sql` that finds the best fitting line for the data in the `pasta_prices` table.

LISTING 4.7: linear_regression.sql

```
use mytools;

DROP TABLE IF EXISTS pasta_prices ;
CREATE TABLE pasta_prices ( kilos INT(3), dollars INT(3));

-- approximate line: dollars = 2*kilos+3
INSERT INTO pasta_prices VALUES (5,12);
INSERT INTO pasta_prices VALUES (6,16);
INSERT INTO pasta_prices VALUES (7,17);
INSERT INTO pasta_prices VALUES (8,20);
INSERT INTO pasta_prices VALUES (9,22);
INSERT INTO pasta_prices VALUES (10,23);
INSERT INTO pasta_prices VALUES (11,25);

SELECT
 @num   := COUNT(dollars)         AS Num,
 @meanX := format(AVG(kilos),3)   AS "XMean",
 @sumX  := SUM(kilos)             AS "XSum",
 @sumXS := SUM(kilos*kilos)       AS "XSumOfSquares",
 @meanY := format(AVG(dollars),3) AS "YMean",
 @sumY  := SUM(dollars)           AS "SumOfY",
 @sumYS := SUM(dollars*dollars)   AS "YSumOfSquares",
 @sumXY := SUM(kilos*dollars)     AS "SumOfX*Y"
 FROM pasta_prices;

SELECT
 @m := format((@n*@sumXY - @sumX*@sumY) / (@num*@sumXS - @sumX*@sumX),3)
 AS slope;

SELECT @b := format((@meanY - @m*@meanX),3) AS intercept;

SELECT CONCAT('Y = ',@m,'X + ',@b) AS 'Least Squares Regression';

SELECT
 format((@num*@sumXY - @sumX*@sumY)
 / SQRT((@num*@sumXS - @sumX*@sumX) * (@num*@sumYS - @sumY*@sumY)),4)
 AS correlation;
```

Listing 4.7 starts by creating (and populating) the table `pasta_prices` with two columns, where the first column contains the number of kilograms and the second column contains the corresponding price for that number of kilograms of pasta.

The next portion of Listing 4.7 contains a SQL statement that initializes some standard quantities that are required for finding the best fitting line. The second and third SQL statements calculate the slope `m` and intercept `b`, respectively, of the best fitting line. The next SQL statement is for display purposes: it displays the best fitting line in the form `Y = m*X + b`.

The final SQL statement uses the quantities from the first SQL statement in order to calculate the correlation of the values in the `my_stats_data` table. Launch the code in Listing 4.7 to see the following output:

```
+-----+-------+------+--------------+--------+--------+--------------+----------+
| Num | XMean | XSum | XSumOfSquares | YMean  | SumOfY | YSumOfSquares | SumOfX*Y |
+-----+-------+------+--------------+--------+--------+--------------+----------+
|   7 | 8.000 |   56 |          476 | 19.286 |    135 |          2727 |     1138 |
+-----+-------+------+--------------+--------+--------+--------------+----------+
1 row in set, 8 warnings (0.000 sec)

+-------+
| slope |
+-------+
| 2.071 |
+-------+
1 row in set, 1 warning (0.000 sec)

+-----------+
| intercept |
+-----------+
| 2.718     |
+-----------+
1 row in set, 1 warning (0.000 sec)

+--------------------------+
| Least Squares Regression |
+--------------------------+
| Y = 2.071X + 2.718       |
+--------------------------+
1 row in set (0.000 sec)

+-------------+
| correlation |
+-------------+
| 0.9866      |
+-------------+
1 row in set (0.000 sec)
```

WINDOW FUNCTIONS

Window functions are functions that can rank data over a specific window or generate ranking indexes within groups. Different relational databases support different functions. Check the documentation to determine whether your database supports the functions listed in this section. If a specific function is not available, consider writing a stored function that implements the functionality that you need for your requirements.

Types of Window Functions in SQL

One way to categorize different types of window functions is as follows:

- Aggregate Functions
- Rank-related Functions
- Statistical Functions
- Functions for Time Series

Aggregate functions include the SQL functions `AVG`, `MIN`, `MAX`, `COUNT`, and `SUM`, all of which specify a table column and then aggregate data based on that column.

Rank-related functions include `ROW_NUMBER`, `RANK`, and `RANK_DENSE` whose purpose is to rank data based on columns in a table or the full dataset.

Statistical functions include `NTILE` (to calculate percentiles, quartiles, and medians) that can be applied to a column or the full dataset. You can think of the `NTILE` function as a "binning" function that partitions data into a set of bins (or buckets). `NTILE` takes an integer as an argument that represents the number of desired bins.

Functions for Time Series include `LAG` and `LEAD` to calculate a month-over-month rolling average.

Recall that Chapter 3 contains an example of the `RANK()` function regarding various countries that won medals in the 2021 Olympics in Japan.

The `RANK` and `DENSE_RANK` functions in MySQL both return sequential numbers (starting from 1) based on the order of the rows that is returned by the `ORDER BY` clause. When you have two records with the same data, then both functions give the same rank to both the rows.

However, only `RANK()` skips the number of positions after records with the same rank number. For example, suppose that `DENSE_RANK()` returns the following values that contains duplicate values with no gaps:

```
1
2
2
2
3
4
5
```

By contrast, the `RANK()` function returns the following list that contains gaps that take into account duplicate values:

```
1
2
2
2
```

```
5
6
7
```

The outcome of sports races uses `RANK`-based values instead of `DENSE_RANK` values. Perform an online search for detailed examples involving the `RANK()`, `DENSE_RANK()`, and `ROW_NUMBER()` functions.

If you want to learn more about window functions, a partial list of MySQL 8.x window functions is available online:

https://dev.mysql.com/doc/refman/8.0/en/window-function-descriptions.html

Some examples of window functions in MySQL 8.x are available online:

https://dev.mysql.com/doc/refman/8.0/en/window-functions-usage.html

THE SQL CASE CLAUSE

This section shows you how to write SQL statements that contain the `CASE` keyword. The SQL `CASE` keyword superficially resembles a `switch` statement in programming languages, such as C and Java, and has the following general structure:

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  WHEN conditionN THEN resultN
  ELSE result
END
```

Each `WHEN` condition is evaluated, and the first one that is `TRUE` will execute its corresponding code that appears after the `THEN` keyword. However, if no `WHEN` condition is `TRUE`, then the code in the `ELSE` keyword is executed.

Listing 4.8 shows the content of `case_weather.sql` that uses a `CASE` statement to modify the values that are returned from the `wind` attribute of the `weather` table.

LISTING 4.8: case_weather.sql

```
SELECT CASE WHEN wind < 0 THEN 0
            WHEN wind > 100 THEN 100
            ELSE wind END
AS wind FROM weather;
```

Log into MySQL and execute the following statements:

```
use mytools;
source case_weather.sql;
```

The output from the preceding code snippet looks similar to the following:

```
+-------+
| wind  |
+-------+
|    16 |
|     3 |
|     0 |
|    16 |
|     0 |
|    12 |
|    12 |
|    32 |
|    16 |
|    99 |
|    12 |
+-------+
11 rows in set (0.000 sec)
```

As another example of the `CASE` statement, Listing 4.9 shows the contents of `create_movies.sql` that first creates a `movie_ratings` table with a single row.

LISTING 4.9: create_movies.sql

```
SELECT @stars = 3;

USE DATABASE mytools;
DROP TABLE IF EXISTS movie_ratings;

CREATE TABLE movie_ratings (movie_id INTEGER, stars
INTEGER, movie_desc VARCHAR(20));

INSERT INTO movie_ratings VALUES(1000, 3, 'unrated');
```

Listing 4.9 involves the usual sequence of SQL statements to create the table `movie_ratings` and then insert a single row of data. Log into MySQL and execute the `movie_ratings.sql` file:

```
MySQL [mytools]>
source movie_ratings.sql;
Database changed
Query OK, 0 rows affected (0.010 sec)
Query OK, 0 rows affected (0.009 sec)
Query OK, 1 row affected (0.001 sec)
```

Verify the contents of the `movie_ratings` table:

```
select * from movie_ratings;
+----------+-------+------------+
| movie_id | stars | movie_desc |
+----------+-------+------------+
|     1000 |     3 | unrated    |
+----------+-------+------------+
1 row in set (0.000 sec)
```

Note the value of the movie_desc attribute is unrated, which will be updated via a CASE statement in the next code sample.

Listing 4.10 shows the content of case_movies.sql that uses a CASE statement that updates the value of the movie_desc attribute of a row in the movie_ratings table.

LISTING 4.10: case_movies.sql

```
SELECT @stars = 3;

UPDATE movie_ratings
SET movie_desc = CASE
               WHEN stars = 1  THEN 'poor'
               WHEN stars = 2  THEN 'minimal'
               WHEN stars = 3  THEN 'decent'
               WHEN stars = 4  THEN 'great'
               WHEN stars = 5  THEN 'fantastic'
               ELSE 'unknown star value' END
WHERE movie_id = 1000;
```

Log into MySQL and execute the following statements that execute the SQL file case_movies.sql to update the lone row in the movie_ratings table:

```
MySQL [mytools]> use mytools;
source case_movies.sql;
Database changed
Query OK, 0 rows affected (0.010 sec)
MySQL [mytools]> select * from movie_ratings;
+----------+-------+------------+
| movie_id | stars | movie_desc |
+----------+-------+------------+
|     1000 |     3 | decent     |
+----------+-------+------------+
```

As you can see in the previous output, the movie_desc attribute has been updated to the value decent.

The final example of a CASE statement shows you that NULL does not equal NULL:

```
SELECT CASE WHEN NULL=NULL THEN "Chicago" ELSE "New York" END;
+-------------------------------------------------------+
| case when null=null then "Chicago" Else "New York" end |
+-------------------------------------------------------+
| New York                                              |
+-------------------------------------------------------+
1 row in set (0.000 sec)
```

WORKING WITH NULL VALUES IN SQL

This section shows you the difference between checking for NULL values versus empty string (‘’) values. Note the following definition of a null in SQL: it’s a value that is unavailable, unassigned, unknown, or inapplicable. Hence, a null is *not* the same as a zero or a blank space.

SQL supports the `IFNull()` function, which is the counterpart of the `NVL()` function that's available in Oracle databases.

```
SELECT IFNULL(1,0); -- returns 1
SELECT IFNULL('',1); -- returns ''
SELECT IFNULL(NULL,'IFNULL function');
-- returns 'IFNULL function'
```

The following SQL statement contains the `IFNULL()` function that returns the value of `workphone` if it's not null; otherwise, it returns the value of `homephone`.

```
SELECT contactname, IFNULL(workphone, homephone) phone
FROM contacts;
```

Listing 4.11 shows the content of `not_null.sql` that uses a `CASE` statement to modify the values that are returned from the `wind` attribute of the `weather` table.

LISTING 4.11: not_null.sql

```
— select rows where forecast is not NULL:
SELECT forecast FROM weather WHERE forecast IS NOT NULL;

— select rows where forecast is not empty string '':
SELECT forecast FROM weather WHERE forecast <> '';
```

Log into MySQL and execute the following statements:

```
use mytools;
source not_null.sql;
```

The output from the preceding code snippet is similar to the following:

```
+----------+
| forecast |
+----------+
| Rain     |
| Sunny    |
| Rain     |
| Sunny    |
| Snow     |
|          |
| Rain     |
|          |
| Snow     |
+----------+
9 rows in set (0.000 sec)

+----------+
| forecast |
+----------+
| Rain     |
| Sunny    |
| Rain     |
```

```
| Sunny     |
| Snow      |
| Rain      |
| Snow      |
+-----------+
7 rows in set (0.001 sec)
```

Listing 4.12 shows the content of `is_null.sql` that illustrates how to select `NULL` values and `''` values.

LISTING 4.12: is_null.sql

```
— the opposite of the queries in not_null.sql:
SELECT * FROM weather WHERE weather.forecast IS NULL;
SELECT * FROM weather WHERE weather.forecast = '';
```

Log into MySQL and execute the following statements:

```
use mytools;
source not_null.sql;
```

The output from the preceding code snippet looks similar to the following:

```
MySQL [mytools]> source is_null.sql;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
+------------+--------+------+----------+------+-------+
2 rows in set (0.000 sec)

+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
+------------+--------+------+----------+------+-------+
2 rows in set (0.000 sec)
```

Listing 4.13 shows the content of `Null_If.sql` that updates the `city` attribute in the `weather` table to `NULL` if the city is `sf`.

LISTING 4.13: Null_If.sql

```
UPDATE WEATHER
SET city = NULLIF(city, 'sf');
```

Listing 4.13 contains a simple SQL `UPDATE` statement that invokes the `NULLIF` statement to set the `city` equal to `NULL` if the city value is `sf`. Log into MySQL and invoke the following command to display the current contents of the `weather` table:

```
MySQL [mytools]> select * from weather;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | sf   | ca    |
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | sf   | ca    |
| 2021-07-03 |     78 |   12 | NULL     | sf   | mn    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-08-06 |     51 |   32 |          | sf   | ca    |
| 2021-09-01 |     42 |   16 | Rain     | sf   | ca    |
| 2021-09-02 |     45 |   99 |          | sf   | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
11 rows in set (0.000 sec)
```

Now launch `Null_If.sql` to update the values in the `weather` table:

```
MySQL [mytools]> source Null_If.sql;
Query OK, 7 rows affected (0.003 sec)
Rows matched: 11  Changed: 7  Warnings: 0
```

Display the rows in the `weather` table and compare the following list with the preceding list:

```
MySQL [mytools]> select * from weather;
+------------+--------+------+----------+------+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+------+-------+
| 2021-04-01 |     42 |   16 | Rain     | NULL | ca    |
| 2021-04-02 |     45 |    3 | Sunny    | NULL | ca    |
| 2021-04-03 |     78 |  -12 | NULL     | se   | wa    |
| 2021-07-01 |     42 |   16 | Rain     |      | ca    |
| 2021-07-02 |     45 |   -3 | Sunny    | NULL | ca    |
| 2021-07-03 |     78 |   12 | NULL     | NULL | mn    |
| 2021-08-04 |     50 |   12 | Snow     |      | mn    |
| 2021-08-06 |     51 |   32 |          | NULL | ca    |
| 2021-09-01 |     42 |   16 | Rain     | NULL | ca    |
| 2021-09-02 |     45 |   99 |          | NULL | ca    |
| 2021-09-03 |     15 |   12 | Snow     | chi  | il    |
+------------+--------+------+----------+------+-------+
11 rows in set (0.001 sec)
```

MISCELLANEOUS ONE-LINERS

This section contains an eclectic collection of functions that are available in MySQL, along with short descriptions of the purpose of the functions. In most cases, the names of the functions have intuitive names, and for those that are not intuitive, the samples make their purpose clear.

```
SELECT SUM(temper) AS total_temp FROM weather;
```

The `LEAST()` function returns the smallest value in a list of values, which can be a list of numeric values or a list of string values, as shown here:

```
SELECT LEAST(3, 12, 34, 8, 25);
+-------------------------+
| LEAST(3, 12, 34, 8, 25) |
+-------------------------+
|                       3 |
+-------------------------+
1 row in set (0.000 sec)

SELECT LEAST("abc", "def", "ghi");
+----------------------------+
| LEAST("abc", "def", "ghi") |
+----------------------------+
| abc                        |
+----------------------------+
1 row in set (0.000 sec)
```

The `GREATEST()` function is the counterpart to the `LEAST()` function that returns the smallest value in a list of values, which can be a list of numeric values or a list of string values, as shown here:

```
SELECT GREATEST(3, 12, 34, 8, 25);
+----------------------------+
| GREATEST(3, 12, 34, 8, 25) |
+----------------------------+
|                         34 |
+----------------------------+
1 row in set (0.000 sec)

SELECT GREATEST("abc", "def", "ghi");
+-------------------------------+
| GREATEST("abc", "def", "ghi") |
+-------------------------------+
| ghi                           |
+-------------------------------+
1 row in set (0.000 sec)
```

The `BIN()` function converts a base 10 integer to a base 2 numbers, as shown here:

```
SELECT BIN(15);
+---------+
| BIN(15) |
+---------+
| 1111    |
+---------+
1 row in set (0.000 sec)
```

The `CONV()` function is more general than the `BIN()` function because it converts an integer from one base to another base, where the two bases are positive integers (i.e., not necessarily 10 and 2), as shown here:

```
SELECT CONV(15, 10, 2);
+-----------------+
| CONV(15, 10, 2) |
+-----------------+
| 1111            |
+-----------------+
1 row in set (0.000 sec)

SELECT CONV(15, 10, 3);
+-----------------+
| CONV(15, 10, 3) |
+-----------------+
| 120             |
+-----------------+
1 row in set (0.000 sec)
```

The `COALESCE()` function processes a list of values that may be `NULL` and returns the first non-null value; if all values are null, then the result is `NULL`. The `COALESCE()` function and the `NULLIF()` are essentially a shortened form of a `CASE` expression. Here is a simple example:

```
SELECT COALESCE(NULL, NULL, NULL, 'abc', NULL, 'def');
+-------------------------------------------------+
| COALESCE(NULL, NULL, NULL, 'abc', NULL, 'def') |
+-------------------------------------------------+
| abc                                             |
+-------------------------------------------------+
```

The `CONVERT()` function converts a value into the specified datatype, an example of which is here:

```
SELECT CONVERT("2021-12-30", DATE);
+-----------------------------+
| CONVERT("2021-12-30", DATE) |
+-----------------------------+
| 2021-12-30                  |
+-----------------------------+
1 row in set (0.000 sec)
```

The `SESSION()` function displays the name of the current MySQL user, an example of which is here:

```
SELECT SESSION_USER();
+----------------+
| SESSION_USER() |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.000 sec)
```

WORKING WITH THE CAST() FUNCTION IN SQL

The `CAST()` function converts a value (of any type) into the specified datatype, an example of which is here:

```
SELECT CAST("2021-12-30" AS DATE);
+----------------------------+
| CAST("2021-12-30" AS DATE) |
+----------------------------+
| 2021-12-30                 |
+----------------------------+
1 row in set (0.000 sec)

SELECT CAST(777 AS CHAR);
+-------------------+
| CAST(777 AS CHAR) |
+-------------------+
| 777               |
+-------------------+
1 row in set (0.000 sec)
```

Listing 4.14 shows the content of `split_float.sql` that splits a floating point number, stored as a string in a table, into its integer portion and its decimal portion.

LISTING 4.14: split_float.sql

```
USE mytools;
DROP TABLE IF EXISTS split_float;
CREATE TABLE split_float (height CHAR(10));

INSERT INTO split_float VALUES ("12.3");
INSERT INTO split_float VALUES ("45.6");
INSERT INTO split_float VALUES ("78.9");
INSERT INTO split_float VALUES ("-3.4");
SELECT * FROM split_float;

--this prevents the huge number in the next SQL statement:
--DELETE
--FROM split_float
--WHERE height < 0;

SELECT CAST(SUBSTRING_INDEX(height, '.', 1) AS UNSIGNED) AS whole_value,
   CAST(SUBSTRING_INDEX(height, '.', -1) AS UNSIGNED) AS decimal_value
FROM split_float;
```

Listing 4.14 creates and populates the table `split_float` with string values that contain decimal numbers. The last portion of Listing 4.14 contains a SQL statement that involves the built-in `CAST()` function and the built-in `SUBSTRING_INDEX()` function in order to extract the integer portion and the decimal portion of the strings in the `split_float` table.

Launch the code in Listing 4.14 to see the following output:

```
MySQL [mytools]> select * from weather;
+--------+
| height |
+--------+
| 12.3   |
| 45.6   |
```

```
| 78.9   |
| -3.4   |
+--------+
4 rows in set (0.000 sec)

+----------------------+---------------+
| whole_value          | decimal_value |
+----------------------+---------------+
|                   12 |             3 |
|                   45 |             6 |
|                   78 |             9 |
| 18446744073709551613 |             4 |
+----------------------+---------------+
4 rows in set, 1 warning (0.000 sec)
+------------+--------+------+----------+-----+-------+
| day        | temper | wind | forecast | city | state |
+------------+--------+------+----------+-----+-------+
```

Notice the enormous number in the bottom row: this is due to specifying `unsigned` in the associated SQL statement. One solution involves deleting rows whose height value is less than 0, which is obviously true if the height value is for humans. Another solution involves finding the substring after the negative sign, and then proceed with splitting the string as above. However, it's logical and much simpler to delete the rows whose height value is negative.

SUMMARY

This chapter started with examples of SQL statements that illustrate numeric functions as well as logarithmic, exponential, and trigonometric functions in SQL. You then learned about aggregate functions and scalar functions in SQL, along with additional examples of the `GROUP BY` clause in a SQL statement.

Next, you learned about Boolean operators and set operators, and how to use the `AND`, `OR`, and `NOT` operators in SQL statements. In addition, you saw examples of SQL statements that use the `ORDER BY` clause and the `MATCH()` function. Next, you learned about CTEs (common table expressions) that were introduced in MySQL 8.0.

Finally, you learned how to perform linear regression in SQL, followed by an overview of window functions, the SQL `CASE` statement, and how to work with `NULL` values in SQL.

CHAPTER 5

NoSQL, SQLite, and Python

This chapter introduces non-relational databases whose feature sets are well-suited to certain types of applications. Specifically, you will learn about NoSQL and MongoDB, which is a popular NoSQL database. Then you will see some of the features of SQLite, SQLAlchemy, and how to access both of them through Python scripts, followed by Python code that accesses MySQL.

The first section (which is roughly half of this chapter) introduces NoSQL, along with Python code samples to manage data in a MongoDB collection. To some extent, this section shows you how to perform operations in MongoDB that are counterparts to SQL commands. You will learn how to create a database in MongoDB, how to create a collection, and how to populate the collection with documents.

The second section shows you the NoSQL command for querying data from a NoSQL collection, as well as deleting document from a collection. You will also learn about Compass (a GUI tool for MongoDB) and PyMongo, which is a Python distribution for working with MongoDB.

The third section returns to MySQL, where you will see how to read MySQL data into a Pandas data frame and then save the data frame as an Excel spreadsheet. Although we won't discuss the details of Pandas and its rich functionality, the Pandas-related code is straightforward. If need be, you can also find online tutorials that discuss various features of Pandas.

The fourth section provides a short description of SQLite, which is a database that is available on mobile devices, such as Android and iOS. As you can probably surmise from its name, SQLite supports a subset of SQL. You can invoke SQL commands in SQLite in the various ways (such as SQLiteStudio) that are discussed in this section.

The final section provides an overview of SQLite, which is a command line tool for managing databases that is available on mobile devices. This section also introduces related tools, such as SQLiteStudio (an IDE for sqlite), DB Browser, and SQLiteDict.

NON-RELATIONAL DATABASE SYSTEMS

There are several types of non-relational data stores, some of which are listed here:

- key-value store
- document store
- wide-column stores
- graph database

The following paragraphs contain a high-level description of the data stores in the preceding list of bullet items. Note that the details of NoSQL are deferred until later sections in this chapter.

A *key/value store* is analogous to a Python dictionary or a hash table (hash map) in Java. In abstract terms, a *key* can unlock a door to give you access to the contents on the other side of that door. In the case of key/value pairs of a key/value store, the *value* is the contents. The value can be anything, including a scalar, a data structure, or a concrete instance of a class. Although key/value stores provide limited functionality, they do provide high performance and are convenient as an in-memory cache.

A *document store* focuses on managing the storage of documents, which can involve XML, JSON, or binary formats. You can also view a document store as a generalization of a key/value store, where the values in these pairs are documents. In general, document stores also provide APIs to perform various operations, such as save, delete, update, and find documents.

A *wide document store* provides column-based storage of name/value pairs, which includes documents. A single column can consist of *multiple* columns, somewhat analogous to a table. Row keys provide access to individual columns, and columns with the same row key belong to the same row in the store. Examples of wide document stores include Bigtable (Google) and Cassandra (Facebook). Distributed databases include GCP (Google), Bigtable (Google), DynamoDB (Amazon), and Azure Storage (Microsoft).

Graph databases are well-suited for more complex data models, such as those that exist in social networks. Each node in a graph database is a record and each edge between two nodes is a relationship between those two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships. Unsurprisingly, the complexity of their structure makes it correspondingly more difficult to easily access their contents.

Advantages of Non-Relational Databases

A NoSQL database is designed to provide fast access to data that may be stored in multiple locations (nodes). Important considerations include

- good scalability
- support for structured and non-structured data
- simpler updates to schemas
- Shared Nothing Architecture

Due to the last point in the preceding bullet list, a DDB (distributed database) is a loosely coupled system in which each node operates on its own physical resources.

In some cases, a DDB will provide strong query abilities, whereas others focus on key-value data representation. A homogeneous DDB involves multiple databases with the same underlying DBMS, whereas a heterogeneous DDB involves multiple databases with different underlying DBMSs.

While distributed databases provide multiple advantages, they can also be more complex than a centralized DBMS.

WHAT IS NOSQL?

Let's start with a clarification: in the early days, NoSQL usually meant "not SQL." More recently, NoSQL has evolved to mean "not only SQL." Moreover, RDBMSs such as Oracle have adapted their database to include support for non-structured data as well as semi-structured data. Nevertheless, RDBMSs are primarily about structured data, and NoSQL databases were designed for data types that are less structured (more about this later). In fact, some RDBMSs, such as Oracle, added support for NoSQL to the Oracle database.

NoSQL includes the data stores and graph databases that are discussed in the previous section. Recall that RDBMSs include the normalization of database tables, whereas NoSQL data is denormalized, and JOIN operations are typically performed in the application code. NoSQL databases enable you to store and retrieve documents (often based on JSON) of variable length, and you can do so without defining a schema or even a table structure. In general, NoSQL databases do not support ACID (they lean toward eventual consistency), so they tend to have high speed transactions.

SQL stores data in tabular form with labelled rows and columns. By contrast, NoSQL databases have a "collection" that is analogous to an RDBMS table. A *collection* can contain multiple documents, where a document is analogous to a row in an RDBMS table.

Collections are not required to conform to a schema, which means that a collection can contain unrelated documents. Although you will probably populate each collection with documents that are logically related, the key point is that you have a great deal of flexibility when making this decision.

Moreover, it's easy to add new fields to one or more documents in a collection without updating a formal schema. Of course, if the documents in a collection have a highly similar (or identical) structure, then it's easier to insert, update, select, or delete such documents.

What is NewSQL?

NewSQL refers to databases that provide the scalability of NoSQL databases and the transactional support of relational databases. Such databases can offer decentralized SQL support and often will provide support for dynamic JSON. Several examples of NewSQL databases include Snowflake, CockroachDB, and Spark SQL. More details regarding NewSQL and additional databases are available online:

https://en.wikipedia.org/wiki/NewSQL

RDBMSS VERSUS NOSQL: WHICH ONE TO USE?

Although RDBMSs and NoSQL databases can support the same types of data (and there are many types), they have different strengths. RDBMSs are suitable for structured data, and NoSQL databases excel in their support for unstructured data. An important advantage of NoSQL and key/value stores is their unlimited horizontal scalability, whereas RDBMSs have vertical expansion.

As you learned in previous chapters, RDBMSs are well-suited for data that can be stored in a tabular form. In addition, the structure of tables (i.e., their attributes along with their types) must be defined in advance. Furthermore, data is accessed through SQL queries, many of which are discussed in Chapter 2 and Chapter 4.

Some simple examples of "suitable data" include the details for customers and purchase orders (one-to-many relationship), along with purchase orders and line items (also a one-to-many relationship).

Another example involves `students` and `classes` whose many-to-many relationship is replaced by a "join" table whose key is the union of 1) the attributes from the primary key for the `students` table and 2) the primary key for the `classes` table. As a result, both the `students` table and the `classes` table have a one-to-many relationship with the join table.

Good Data Types for NoSQL

A NoSQL database is well-suited for documents, images, audio, and video, all of which have variable lengths (and different formats) and can be stored as single entities without adhering to the normalization process that you will see in Chapter 6. Although it's certainly possible for RDBMSs to manage these types of entities, write and read operations might require accessing different parts of an entity from multiple tables. Recall that normalization requires a `JOIN` keyword in SQL clauses that retrieve data from multiple tables, which in turn can adversely affect performance.

Moreover, the document model for NoSQL allows for fields to vary from document to document, all of which can belong to the same collection. In addition, more recent versions of MongoDB provide ACID compliance, and in conjunction with transaction support, this functionality can give MongoDB the look-and-feel of RDBMSs.

It is important to select the system (whenever possible) that best fits the requirements for your application.

Some Guidelines for Selecting a Database

MongoDB might be a better choice under the following conditions:

- you need high data availability and fast, automatic, and instant data recovery
- you work with an unstable schema
- your services are mostly cloud-based, so the native scale-out architecture that MongoDB comes with will be suitable for your business
- the architecture provides sharding, which aligns with horizontal scaling offered through cloud computing.

MySQL could be a better option under the following conditions:

- you are starting a business and the database is not going to scale
- the schema is fixed and its data structure will not change over time
- you want high-performance ability on a low budget
- you need a high transaction rate
- data security is the foremost priority

Of course, the preceding lists of bullet items only provide guidelines rather than a definitive list of criteria. Before you make a decision, make sure you perform a thorough evaluation of two types of databases based on a prioritized list of requirements.

NoSQL Databases

The following list contains several NoSQL databases that are available for free:

- CockroachDB
- FaunaDB
- HarperDB
- RethinkDB

Before you decide to adopt one of the preceding databases, compare your list of requirements with each of these databases (and you might decide to adopt MySQL). If two of them are viable candidates, check for blog posts that contain a detailed comparison. If you decide to utilize an application that uses each of

those two databases, remember that performance-related issues generally arise when there is a high volume of data and/or many simultaneous transactions.

Now that you have an overview of some of the differences between RDBMSs and NoSQL databases, let's take a closer look at MongoDB, which is the topic of the next section.

WHAT IS MONGODB?

MongoDB is a popular NoSQL database that supports NoSQL operations on data. As a quick reminder, in an earlier chapter, you learned that an RDBMS allows you to create databases and tables and then insert data into those tables. By contrast, MongoDB supports the creation of databases and collections, after which you can insert documents into the collections (discussed in more detail shortly).

Features of MongoDB

In addition to support for many standard query types, MongoDB offers the following features:

- sharding
- load balancing
- scalability
- schemas are optional
- support for indexes

Installing MongoDB

There are two versions of MongoDB that you can install on your machine. The MongoDB community edition is downloadable:

https://docs.mongodb.com/manual/installation/#mongodb-community-edition-installation-tutorials

Note that on MacOS, you can use brew to install MongoDB. The MongoDB Enterprise edition is downloadable:

https://docs.mongodb.com/manual/administration/install-enterprise/

In addition, you can use MongoDB with Docker (search online for tutorials and instructions).

Launching MongoDB

Launch the command `mongo` without arguments, which then launches a command shell and also connects to the URL `mongod://127.0.0.1:27017`.

The preceding URL is the default local server, and you're connected to the local host through port 27017. Type the following command to find the location of the mongo executable:

```
$ which mongo
/usr/local/bin/mongo
```

Type `mongo` from a command shell in order to enter the mongo shell:

```
$ mongo
```

If everything has been set up correctly, you will see the following (or something similar):

```
MongoDB shell version v4.4.3
connecting to: mongodb://127.0.0.1:27017/?compressors=disab
led&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("2c254ca6-adf4-
466f-8a87-a182d801ee0e") }
MongoDB server version: 4.4.3
// other details omitted for brevity
>
```

The `mongo` shell makes a connection to the `test` database, and you can verify the latter by typing the following in the MongoDB command shell:

```
> db
Test
```

Display the existing MongoDB databases with this command:

```
> show databases
admin   0.000GB
config  0.000GB
local   0.000GB
```

You can also replace `databases` with `dbs` in the preceding command. Note that `admin` and `local` are databases that are part of every MongoDB cluster.

USEFUL MONGO APIS

With MongoDB, you can create one or more databases, where each database can contain one or more collections, and each collection can contain one or more JSON-based documents.

MongoDB supports CRUD operations that include `find()`, `insert()`, `update()`, and `delete()`, where these keywords can be suffixed with "One" or "Many" (e.g., `findOne()` or `findMany()`).

You can find data in a Mongo database with the following APIs:

- `db.collection.find()` lists all the documents in the collection.
- `db.collection.findOne()` lists only the first document in the collection.

Insert data with these MongoDB APIs:

- `db.collection.insert()` creates a new document in a collection.
- `db.collection.insertOne()` inserts a new document in a collection.
- `db.collection.insertMany()` inserts several new documents in a collection.

Update data with these MongoDB APIs:

- `db.collection.update()` modifies a document in a collection.
- `db.collection.updateOne()` modifies a single document in a collection.
- `db.collection.updateMany()` modifies multiple documents in a collection.
- `db.collection.replaceOne()` modifies a single document in a collection.

Delete data with these MongoDB APIs:

- `db.collection.remove()`: Delete a single document or all documents that match a specified filter.
- `db.collection.deleteOne()`: Delete, at most, a single document that matches a specified filter even though multiple documents may match the specified filter.
- `db.collection.deleteMany()`: Delete all documents that match a specified filter.

Meta Characters in Mongo Queries

MongoDB supports these two meta characters and a lowercase switch that you can use when you want to find substrings of a text string:

- `$` matches the end of a line
- `^` matches the beginning of a line
- `i` means "ignore case"

Consider the following list of names that we will use with the preceding bullet items:

```
Smith1
Smith2
Smith3
Smith4
Smith5
```

The following expression does not match anything in the list (it starts with a lowercase "s" instead of an uppercase "S"):

```
/smith1/
```

The following expression matches `Smith1` in the list because of the `i` switch:

```
/smith1/i
```

The following expression matches `Smith1` through `Smith5` because `^S` matches any string that starts with a capital `S`:

```
/^S/
```

The following expression matches `Smith5` because `/^5/` matches any string that ends in the digit 5:

```
/5$/
```

MONGODB COLLECTIONS AND DOCUMENTS

In simplified terms, think of a collection as a container-like entity that enables you to store documents. In addition, you can think of a document as a set of name/value pairs, where the values can be simple data types (e.g., numbers or strings) as well as arrays. Thus, MongoDB has a document-oriented data model instead of a table-oriented data model.

MongoDB's document-oriented model means that documents can be managed in their entirety instead of splitting them into components that are stored in different tables whose relationship must be defined, such as a one-to-many relationship that involves a foreign key.

Instead, you create a collection and simply insert documents in that collection. MongoDB provides APIs for managing the documents in a given collection. the MongoDB performs a *lazy* creation of databases and collections, which means that databases and collections are created after you insert the first document.

Document Format in MongoDB

The documents in MongoDB are composed of field-and-value pairs and have the following structure:

```
{
   field1 → value1,
   field2 → value2,
   field3 → value3,
   ...
   fieldN → valueN
}
```

The value of a field can be any BSON datatype, including other documents, arrays, and arrays of documents. In practice, you'll specify your documents using the JSON format.

CREATE A MONGODB COLLECTION

Unlike an RDBMS, MongoDB does not have a `CREATE` command. Instead, you need to invoke the `use` command to create a database, and then the `INSERT` command to insert a document, after which a database is created:

```
use temp
Insert a document in temp
```

Enter the following commands from the command line:

```
> use temp
switched to db temp
> db.temp.insertOne({"fname": "John", "lname": "Smith"})
{
   "acknowledged" : true,
   "insertedId" : ObjectId("603dad30876da25aabd36d5f")
}
```

You can also insert multiple documents, as shown here:

```
> doc1 = {"fname": "John", "lname": "Smith"}
{ "fname" : "John", "lname" : "Smith" }

> doc2 = {"fname": "Jane", "lname": "Jones"}
{ "fname" : "Jane", "lname" : "Jones" }

> doc3 = {"fname": "Dave", "lname": "Stone"}
{ "fname" : "Dave", "lname" : "Stone" }

> db.temp.insertMany([doc1,doc2,doc3])
{
   "acknowledged" : true,
   "insertedIds" : [
         ObjectId("603daee5876da25aabd36d60"),
         ObjectId("603daee5876da25aabd36d61"),
         ObjectId("603daee5876da25aabd36d62")
   ]
}
>
> db.temp.find()
{ "_id" : ObjectId("603dad30876da25aabd36d5f"), "fname" :
"John", "lname" : "Smith" }
{ "_id" : ObjectId("603daee5876da25aabd36d60"), "fname" :
"John", "lname" : "Smith" }
{ "_id" : ObjectId("603daee5876da25aabd36d61"), "fname" :
"Jane", "lname" : "Jones" }
{ "_id" : ObjectId("603daee5876da25aabd36d62"), "fname" :
"Dave", "lname" : "Stone" }
>
> db.temp.find({fname : "Jane"})
{ "_id" : ObjectId("603daee5876da25aabd36d61"), "fname" :
"Jane", "lname" : "Jones" }
>
```

Let's create a MongoDB collection called `cellphones` whose documents contain the attributes `year`, `os`, `model`, `color`, and `price`. Unlike an RDBMS that requires you to define the attributes of a database table, you can create a collection simply by inserting a document in that collection. Here is an example of inserting a row in the `cellphones` collection, which will create the `cellphones` collection if it does not already exist:

```
> use cellphones
switched to cellphones
> db
cellphones
> db.cellphones.insert({"year":"2017","os":"android","model":
"pixel2","color":"black","price":320})
```

The following document contains data for a specific cell phone in the `cellphones` collection:

```
{
 "_id"   : ObjectId("600c626932e0e6419cee81a7"),
 "year"  : "2017",
 "os"    : "android",
 "model" : "pixel2",
 "color" : "black",
 "price" : 320
}
```

Invoke the following command if you want to delete the `cellphones` collection:

```
> db.cellphones.drop()
```

WORKING WITH MONGODB COLLECTIONS

This section contains a set of examples that illustrate how to use various MongoDB APIs for managing the data in the `cellphones` collection that was created in the previous section. The following sections contain code blocks that illustrate how to use the following APIs.

```
find()
insertOne()
insertMany()
aggregate()
```

Find all Android Phones

NoSQL (and MongoDB) provide the `find()` function to query a collection for documents. The following query finds the `Android` cell phones in the `cellphones` collection:

```
> db.cellphones.find( {os: "android"} ).limit(1).pretty(){
 "_id"   : ObjectId("600c63cf32e0e6419cee81ab"),
 "year"  : "2017",
 "os"    : "android",
 "model" : "pixel2",
 "color" : "black",
 "price" : 28000
}
```

In addition to the `find()` function, the preceding query contains two additional functions. First, the `limit()` function is invoked to limit the number of rows that are returned in a query. Second, the `pretty()` function is invoked to display the output in a more aesthetically pleasing manner.

If you decide to omit the `pretty()` function in the preceding query, the output looks like this:

```
> db.cellphones.find( {os: "android"} ).limit(1){ "_id" :
ObjectId("600c63cf32e0e6419cee81ab"), "year" : "2017", "os"
: "android", "model" : "pixel2", "color"  "black", "price"
: 320 }
```

Find All Android Phones in 2018

NoSQL (and MongoDB) allow you to list multiple comma-separated conditions in order to specify Boolean AND logic on the specified conditions:

```
> db.cellphones.find( {os: "android", year: "2018"} ).pretty(){
 "_id"   : ObjectId("600c63cf32e0e6419cee81af"),
 "year"  : "2018",
 "os"    : "android",
 "model" : "pixel3",
 "color" : "white",
 "price" : 700
}
```

Insert a New Item (document)

MongoDB provides the `insertOne()` function to insert a single document in a collection. An example of inserting (creating) a new document is as follows:

```
> db.cellphones.insertOne(
... {year: "2017", os: "bmw", color: "silver",
...   km: 28000, price: 39000}
... ){
 "acknowledged" : true,
 "insertedId" : ObjectId("600c6bc79445b834692e3b91")
}
```

Update an Existing Item (document)

MongoDB provides the `update()` function to update an existing document in a collection. For example, the following query specifies the condition that indicates the documents to be updated, and then passes the updated values as well as the `set` keyword:

```
> db.cellphones.update(
... { os: "bmw" },
... { $set: { os: "ios" }},
... { multi: true }
... )WriteResult({ "nMatched" : 5, "nUpserted" : 0, "nModified" : 5 })
```

We need to use the `multi` parameter to update all the documents that meet the given condition. Otherwise, only one document will be updated.

Calculate the Average Price for Each Brand

MongoDB provides the `aggregate()` function to assemble documents into similar groups. The subsequent query does the following:

1. groups the documents based on brands by selecting `$os` as `id`
2. specifies the aggregation function, which is `$avg`
3. specifies the field to be aggregated
4. inserts a single document in a collection

Here is the query that performs the preceding list of steps:

```
> db.cellphones.aggregate([
... { $group: { _id: "$make", avg_price: { $avg: "$price" }}}
... ]){ "_id" : "hyundai", "avg_price" : 36333.333333333336 }
{ "_id" : "ios", "avg_price" : 47400 }
{ "_id" : "android", "avg_price" : 35333.333333333336 }
```

If you are familiar with Pandas, the syntax is similar to the `groupby` function.

Calculate the Average Price for Each Brand in 2019

This task is to start with the query in the preceding section and add another condition that specifies a value of 2019 for the year:

```
> db.cellphones.aggregate([
... { $match: { year: "2019" }},
... { $group: { _id: "$os", avg_price: { $avg: "$price" }}}
... ]){ "_id" : "ios", "avg_price" : 53000 }
{ "_id" : "android", "avg_price" : 42000 }
{ "_id" : "ios",     "avg_price" : 41000 }
```

Import Data with mongoimport

The `mongoimport` utility is a command line utility that enables you to import JSON, CSV, or TSV files into a MongoDB database. For example, you can import the CSV file `data.csv` into the `mytools` database with the following command:

```
mongoimport —db mytools —file /tmp/data.csv
```

WHAT IS FUGUE?

Fugue a Python-based library that enables you to invoke SQL-like queries against Pandas data frames via FugueSQL. Install Fugue with the following command (specify a different version if you need to do so):

```
pip3.7 install fugue
```

Listing 5.1 shows the content of `fugue1.py` that illustrates how to populate a Pandas data frame and then invoke various SQL commands to retrieve a subset of the data from the Pandas data frame.

LISTING 5.1: fugue1.py

```
import pandas as pd
from fugue_sql import fsql

df1 = pd.DataFrame({'fnames': ['john', 'dave', 'sara', 'eddy'],
                    'lnames': ['smith','stone','stein','bower'],
                    'ages':   [30,33,34,35],
                    'gender': ['m','m','f','m']})

print("=> data frame:")
print(df1)
print()

# Example #1: select users who are older than 33:
query_1 = """
SELECT fnames, lnames, ages, gender FROM df1
WHERE ages > 33
PRINT
"""

# display the extracted data:
fsql(query_1).run()
```

Listing 5.1 starts with `import` statements and then initializes the Pandas data frame `df1` with a set of data values. The next portion of Listing 5.1 constructs a query that retrieves the data values of all users who are older than 33. Launch the code in Listing 5.1 to see the following output:

```
collection:
=> data frame:
  fnames lnames  ages gender
0   john  smith    30      m
1   dave  stone    33      m
2   sara  stein    34      f
3   eddy  bower    35      m

ANTLR runtime and generated code versions disagree: 4.8!=4.9
ANTLR runtime and generated code versions disagree: 4.8!=4.9
ANTLR runtime and generated code versions disagree: 4.8!=4.9
ANTLR runtime and generated code versions disagree: 4.8!=4.9
PandasDataFrame
fnames:str|lnames:str|ages:long|gender:str
----------+----------+---------+----------
sara      |stein     |34       |f
eddy      |bower     |35       |m
Total count: 2
```

WHAT IS COMPASS?

MongoDB Compass is a free GUI tool for MongoDB that enables you to manage data in a MongoDB database. In addition, you can use this GUI to visually explore data and execute ad hoc queries. Compass is available for multiple platforms, such as Mac, Linux, and Windows. The instructions for downloading Compass are available online:

https://docs.mongodb.com/compass/master/install/

After completing the installation, launch Compass and in the "Connect to Host" page, enter the following information:

```
Hostname: localhost
Port: 27107
Favorite Name: You Decide
```

WHAT IS PYMONGO?

PyMongo is a Python distribution for working with MongoDB via Python. Install PyMongo on your machine with the following command:

```
pip3 install pymongo==3.11.2
```

The following website contains thorough documentation for learning how to use PyMongo:

https://pymongo.readthedocs.io/en/stable/index.html

Listing 5.2 shows the content of `pymongo1.py` that connects to the `mytools` MongoDB database.

LISTING 5.2: pymongo1.py

```
import pymongo

# a client instance:
myclient = MongoClient("localhost",27017)

# connect to mytools:
db = myclient['mytools']

coll = db['weather']
print("collection:")
print(coll)
```

Listing 5.2 starts with an `import` statement and then initializes the variable my client as an instance of the `MongoClient` class. Next, the variable `db` is initialized as the database connection to the `mytools` database. The remaining code involves the variable `coll`, which is a reference to the `weather` collection, whose contents are then displayed. Launch the code in Listing 5.2 to see the following output:

```
collection:
Collection(Database(MongoClient(host=['localhost:27017'],
document_class=dict, tz_aware=False, connect=True),
'mytools'), 'weather')
```

In addition, PyMongoArrow enables you to load MongoDB result sets in several ways: as NumPy arrays, as Pandas data frames, or as Apache Arrow tables. Install PyMongoArrow with this command:

```
pip3 install pymongoarrow
```

This concludes the portion of the chapter regarding MongoDB. The next section returns to MySQL and discusses how to access a MySQL database via SQLAlchemy and Pandas.

MYSQL, SQLALCHEMY, AND PANDAS

There are several ways to interact with a MySQL database, one of which is via SQLAlchemy. The Python code samples in subsequent sections rely on SQLAlchemy (which is briefly described in the next section) and Pandas.

What is SQLAlchemy?

SQLAlchemy is an ORM (Object Relational Mapping), which serves as a "bridge" between Python code and a database. Install SQLAlchemy with this command:

```
pip3 install sqlalchemy
```

SQLAlchemy handles the task of converting Python function invocations into the appropriate SQL statements, as well as providing support for custom SQL statements. In addition, SQLAlchemy supports multiple databases, including MySQL, Oracle, PostgreSQL, and SQLite.

Read MySQL Data via SQLAlchemy

The previous section showed you how to install SQLAlchemy. Install Pandas (if you haven't done so already) with this command:

```
pip3 install pandas
```

The Pandas functionality in the code samples involve the intuitively named `read_sql()` method and the related `read_sql_query()` method, both of which read the contents of a MySQL table.

Listing 5.3 shows the content of `read_sql_data.py` that reads the contents of the `people` table.

LISTING 5.3: read_sql_table.py

```
from sqlalchemy import create_engine
import pymysql
import pandas as pd

engine = create_engine('mysql+pymysql://root:yourpassword@1
27.0.0.1',pool_recycle=3600)
dbConn = engine.connect()
frame  = pd.read_sql("select * from mytools.people", dbConn);

pd.set_option('display.expand_frame_repr', False)
print(frame)
dbConn.close()
```

Listing 5.3 starts with several `import` statements that are required to access a MySQL database. The next portion of code initializes the variable `engine` as a reference to MySQL, followed by `dbConn`, which is a database connection. Next, the variable `frame` is initialized with the rows in the `people` table. Launch the following command in a command shell:

```
python3 read_sql_table.py
```

You will see the following output:

```
  fname  lname age gender  country
0  john  smith  30      m      usa
1  jane  smith  31      f   france
2  jack  jones  32      m   france
3  dave  stone  33      m    italy
4  sara  stein  34      f  germany
5  eddy  bower  35      m    spain
```

Listing 5.4 shows the content of `sql_query.py` that reads the contents of the `people` table.

LISTING 5.4: sql_query.py

```
from sqlalchemy import create_engine
import pymysql
import pandas as pd
engine = create_engine('mysql+pymysql://root:yourpassword@1
27.0.0.1',pool_recycle=3600)

query_1 = '''
select * from mytools.people
'''

print("create dataframe from table:")
df_2 = pd.read_sql_query(query_1, engine)

print("dataframe:")
print(df_2)
```

Listing 5.4 starts with several `import` statements followed by initializing the variable `engine` as a reference to a MySQL instance. Next, the variable `query_1` is defined as a string variable that specifies a SQL statement that selects all the rows of the `people` table, followed by the variable `df_2` (a data frame) that returns the result of executing the SQL statement specified in the variable `query_1`. The final code snippet displays the contents of the `people` table. Launch the following command in a command shell:

```
python3 sql_query.py
```

You will see the following output:

```
  fname  lname age gender  country
0  john  smith  30      m      usa
```

```
1  jane  smith  31      f   france
2  jack  jones  32      m   france
3  dave  stone  33      m    italy
4  sara  stein  34      f  germany
5  eddy  bower  35      m    spain
```

Launch the following Python script in a command shell:

```
python3 sql_query.py
```

You will see the following output:

```
  fname  lname age gender  country
0  john  smith  30      m      usa
1  jane  smith  31      f   france
2  jack  jones  32      m   france
3  dave  stone  33      m    italy
4  sara  stein  34      f  germany
5  eddy  bower  35      m    spain
```

EXPORT SQL DATA FROM PANDAS TO EXCEL

Listing 5.5 shows the content of `sql_query_excel.py` that reads the contents of the `people` table into a Pandas data frame and then exports the latter to an Excel file.

LISTING 5.5: sql_query_excel.py

```
from sqlalchemy import create_engine
import pymysql
import pandas as pd

engine = create_engine('mysql+pymysql://root:yourpassword@1
27.0.0.1',pool_recycle=3600)

query_1 = '''
select * from mytools.people
'''

print("create dataframe from table:")
df_2 = pd.read_sql_query(query_1, engine)

print("Contents of Pandas dataframe:")
print(df_2)

import openpyxl
print("saving dataframe to people.xlsx")
df_2.to_excel('people.xlsx', index=False)
```

Listing 5.5 contains several `import` statements followed by the variable `engine` that is initialized to an "endpoint" from which a MySQL database can be accessed. The next code snippet initializes the variable `query_1` as a string that contains a simple SQL `SELECT` statement.

Next, the variable `df_2` is a Pandas data frame that initialized to the result of invoking the SQL statement defined in the variable `query_1`, after which the contents of `df_2` are displayed. The final portion of code in Listing 5.5 saves the contents of `df_2` to an Excel document called `people.xlsx`. Launch the following command in a command shell:

```
python3 sql_query_excel.py
```

The preceding command generates the following output:

```
Creating dataframe from table people
Contents of Pandas dataframe:
   fname  lname age gender  country
0   john  smith  30      m      usa
1   jane  smith  31      f   france
2   jack  jones  32      m   france
3   dave  stone  33      m    italy
4   sara  stein  34      f  germany
..   ...    ...  ..    ...      ...
73  jane  smith  31      f   france
74  jack  jones  32      m   france
75  dave  stone  33      m    italy
76  sara  stein  34      f  germany
77  eddy  bower  35      m    spain

[78 rows x 5 columns]
saving dataframe to people.xlsx
```

NOTE *You might need to launch the previous Python script using Python 3.7 instead of Python 3.8 or Python 3.9.*

The next section contains Pandas-related functionality that does not involve any database connectivity. Since the previous portion of this chapter contains Pandas-related functionality, it's a convenient location for this material. However, if you prefer, you can skip this section with no loss of continuity, and proceed to the next section that discusses SQLite.

MYSQL AND CONNECTOR/PYTHON

MySQL provides a connector/Python API as another mechanism for connecting to a MySQL database. This section contains some simple Python code samples that rely on `connector/Python` to connect to a database and retrieve rows from a database table.

Before delving into the code samples, keep in mind that MySQL 8 uses `mysql_native_password` instead of `caching_sha2_password`. As a result, you need to specify a value for `auth_plugin` (which is not specified in various online code samples). Here is the error message:

```
mysql.connector.errors.NotSupportedError: Authentication
plugin 'caching_sha2_password' is not supported
```

The solution is highlighted in the Python code sample in the next section.

Establishing a Database Connection

Listing 5.6 shows the content of `mysql_conn1.py` that illustrates how to establish a `connector/Python` database connection.

LISTING 5.6: mysql_conn1.py

```
# optional for OS X:
import sys
sys.path.append('/usr/local/lib/python3.9/site-packages')

import mysql.connector

cnx = mysql.connector.connect(user='root',
                              password='yourpassword',
                              host='localhost',
                              database='employees',
                              auth_plugin='mysql_native_password')
cnx.close()
```

Listing 5.6 contains an `import` statement in order to set the appropriate path for Python 3.9. If the code executes correctly on your system without these two lines of code, then you can safely delete them.

The next code snippet is an `import` statement, followed by initializing the variable `cnx` as a database connection. Note the snippet shown in bold, which is required for MySQL 8 to connect to a MySQL database, as described in the introductory portion of this section. Launch the code in Listing 5.6, and if you don't see any error messages, then the code worked correctly.

Reading Data from a Database Table

Listing 5.7 shows the content of `mysql_pandas.py` that illustrates how to establish a database connection and retrieve the rows in a database table.

LISTING 5.7: mysql_pandas.py

```
# optional:
import sys
sys.path.append('/usr/local/lib/python3.9/site-packages')

import mysql.connector

mydb = mysql.connector.connect(user='root',
                              password='yourpassword',
                              host='localhost',
                              database='employees',
                              auth_plugin='mysql_native_password')

mycursor = mydb.cursor()

# select all rows from the employees table:
mycursor.execute('SELECT * FROM employees')

import pandas as pd
```

```
# populate a Pandas data frame with the data:
table_rows = mycursor.fetchall()
df = pd.DataFrame(table_rows)

print("data frame:")
print(df)

mydb.close()
```

Listing 5.7 starts with the same `import` statement as Listing 5.6 and for the same purpose. The next code snippet is an import statement, followed by initializing the variable `cnx` as a database connection. Note the snippet shown in bold, which is required for MySQL 8 in order to connect to a MySQL database. Launch the code in Listing 5.7, and if everything worked correctly, you will see the following output:

```
=> Contents of data frame:
      0     1                   2
0  1000  2000           Developer
1  2000  3000        Project Lead
2  3000  4000         Dev Manager
3  4000  4000  Senior Dev Manager
```

Creating a Database Table

Listing 5.8 shows the content of `create_fun_table.py` that illustrates how to establish a database connection and create a database table.

LISTING 5.8: create_fun_table.py

```
# optional for OS X:
import sys
sys.path.append('/usr/local/lib/python3.9/site-packages')

my_table = (
    "CREATE TABLE 'for_fun' ("
    "  'dept_no' char(4) NOT NULL,"
    "  'dept_name' varchar(40) NOT NULL,"
    "  PRIMARY KEY ('dept_no'), UNIQUE KEY 'dept_name'
('dept_name')"
    ") ENGINE=InnoDB")

DB_NAME = 'for_fun_db'

import mysql.connector
cnx = mysql.connector.connect(user='root',
                              password='yourpassword',
                              host='localhost',
                              database='mytools')
cursor = cnx.cursor()

try:
  print("Creating table {}: ".format(my_table), end='')
  cursor.execute(my_table)
```

```
except mysql.connector.Error as err:
  if err.errno == errorcode.ER_TABLE_EXISTS_ERROR:
    print("already exists.")
  else:
    print(err.msg)
else:
  print("Table created:",my_table)

cursor.close()

cnx.close()
```

Listing 5.8 starts by initializing the variable `my_table` as a string that contains a SQL statement for creating a MySQL table. The next portion of Listing 5.8 initializes the variable `cnx` as a connection to the `mytools` database, and then initializes the variable `cursor` as a database cursor.

The next portion of Listing 5.8 contains a `try/catch` block to create the table `for_fun` that is specified in the string variable `my_table`. The `except` block catches the connection-related error, and displays an appropriate message if the error occurred because the specified table already exist, or for some other reason.

Now launch the code in Listing 5.8, and if everything worked correctly, you will see the following output:

```
Creating table CREATE TABLE 'for_fun' (  'dept_no' char(4)
NOT NULL,  'dept_name' varchar(40) NOT NULL,  PRIMARY
KEY ('dept_no'), UNIQUE KEY 'dept_name' ('dept_name'))
ENGINE=InnoDB: Table created: CREATE TABLE 'for_fun' (
'dept_no' char(4) NOT NULL,  'dept_name' varchar(40) NOT
NULL,  PRIMARY KEY ('dept_no'), UNIQUE KEY 'dept_name'
('dept_name')) ENGINE=InnoDB
```

Open a command shell, and from the MySQL prompt, enter the following command:

```
MySQL [mytools]> desc for_fun;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| dept_no   | char(4)     | NO   | PRI | NULL    |       |
| dept_name | varchar(40) | NO   | UNI | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
2 rows in set (0.060 sec)
```

WHAT IS SQLITE?

SQLite is a light weight, portable, and open source RDBMS that is available on Windows, Linux, and MacOS, as well as Android and iOS. More information is available online:

https://www.sqlite.org

https://www.sqlitetutorial.net/sqlite-commands/

SQLite is ACID-compliant and also implements most SQL standards. Let's look at some features of SQLite and the installation process, both of which are discussed in two subsections.

SQLite Features

SQLite provides several useful features, some of which are listed as follows:

- doesn't require a separate server process or system to operate
- no system administration
- no external dependencies
- can operate in a serverless environment.
- Available in multiple platforms (Unix, Linux, Mac, and Windows)
- ACID transactions
- Full support for all features in SQL92

SQLite Installation

Download the distribution for your operating system from the following site:

https://www.sqlite.org/download.html

The second step is to unzip the downloaded file in a convenient location, which we'll assume is the directory `$HOME/sqlite3_home`.

Note that if you have a MacBook, then the directory that contains the `sqlite3` executable is automatically in the `PATH` variable. Type the following command to see if `sqlite3` is accessible:

```
which sqlite3
```

If the preceding command returns a blank line, then you need to include the path to the `bin` directory where `sqlite3` is located. For example, if the preceding directory is `$HOME/sqlite3_home/bin`, then update the `PATH` environment variable as follows:

```
export PATH=/$HOME/sqlite_home/bin:$PATH
```

The following sequence of commands shows you how to launch sqlite, open a database, and display the contents of the `employees` table (which is created in the next section). Type all the text that is displayed in bold below:

```
sqlite3
sqlite> use sqlite3_mytools
sqlite> .open /Users/oswaldcampesato/sqlite3_mytools
sqlite> .tables
employees
sqlite> select * from employees;
1200|10000|BizDev
1100|10000|Sales
1000|10000|Developer
sqlite> .quit
```

The `.open` command opens existing databases and creates a new database, as shown above. The `employees` table was already created in an IDE, and you will see how to create that table (and any other table that you want) in the next section.

Although you can perform SQL operations from the command line, just like you can with MySQL, it's probably easier to work with SQLite in an IDE. In fact, a very robust IDE is SQLiteStudio, which is discussed in the next section.

SQLiteStudio Installation

SQLiteStudio is an open source IDE for SQLite that enables you to perform many database operations, such as creating, updating, and dropping tables and views. Download the distribution for your operating system, and perform the specified installation steps:

https://sqlitestudio.pl/

https://mac.softpedia.com/get/Developer-Tools/SQLiteStudio.shtml

Figure 5.1 shows the structure of the `employees` table whose definition is the same as the `employees` table in the `mytools` database in MySQL.

FIGURE 5.1 The `employees` table.

Figure 5.2 displays a screenshot of three rows in the `employees` table, where you can insert a fourth row of data in the top row that is pre-populated with `NULL` values.

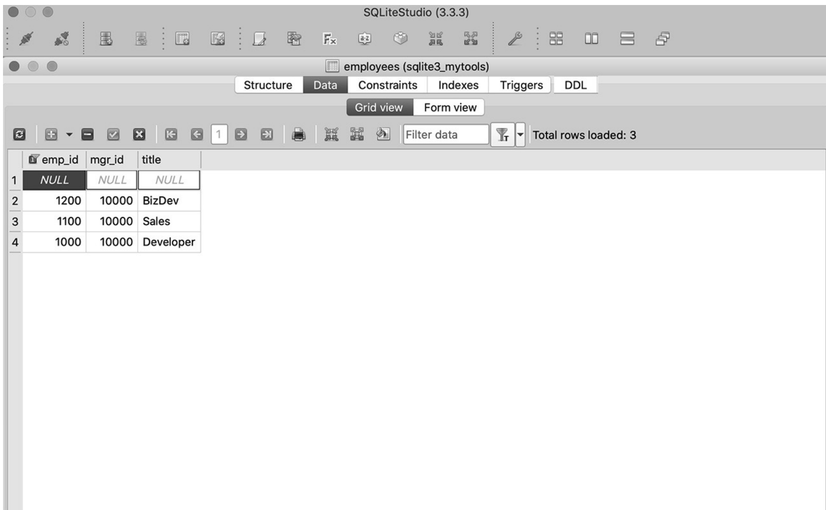

FIGURE 5.2 Three rows in the `employees` table.

DB Browser for SQLite Installation

DB Browser is an open source and visually-oriented tool for SQLite that that enables you to perform various database-related operations, such as creating and updating files. Moreover, this tool enables you to manage data through an interface that resembles a spreadsheet.

Download the distribution for your operating system, and perform the specified installation steps:

https://www.macupdate.com/app/mac/38584/db-browser-for-sqlite/download/secure

The following website contains a multitude of URLs that provide details regarding the features of DB Browser:

https://sqlitebrowser.org

SQLiteDict (optional)

SQLiteDict is an open source tool that is a wrapper around sqlite3, and it's available online:

https://pypi.org/project/sqlitedict/

SQLiteDict enables you to persist dictionaries to a file on the file system, as illustrated by the code in Listing 5.9.

LISTING 5.9: sqlitesavedict1.py

```
# pip3 install sqlitedict

from sqlitedict import SqliteDict

mydict = SqliteDict('./my_db.sqlite', autocommit=True)
mydict['pasta'] = 'pasta'
mydict['pizza'] = 'pizza'

for key, value in mydict.iteritems():
  print("key:",key," value:",value)

# dictionary functions work:
print("length:",len(mydict))
mydict.close()

# a client instance:
myclient = MongoClient("localhost",27017)
```

Listing 5.9 contains an `import` statement followed by the variable `mydict` that is initialized as a dictionary that includes the two strings `pasta` and `pizza`. The next code snippet contains a loop that displays the key/value pairs of `mydict`, followed by the length of the `mydict` dictionary. The next close snippet closes the dictionary and then launches a MongoDB client at the default port. Launch the code in Listing 5.9 to see the following output:

```
key: pasta  value: pasta
key: pizza  value: pizza
number of items: 2
```

As you can see, Listing 5.9 shows you how to save key/value pairs, and Listing 5.10 illustrates how to read the contents of the file saved in Listing 5.9.

LISTING 5.10: sqlitereaddict1.py

```
# pip3 install sqlitedict

# read the contents of my_db.sqlite
# and note no autocommit=True
with SqliteDict('./my_db.sqlite') as mydict:
  print("old:", mydict['pasta'])
  mydict['pasta'] = u"more pasta"
  print("new:", mydict['pasta'])
  mydict['pizza'] = range(10)
  mydict.commit()
  # this is not persisted to disk:
  mydict['dish'] = u"deep dish"

# open the same file again:
with SqliteDict('./my_db.sqlite') as mydict:
  print("pasta:",mydict['pasta'])
  # this line will cause an error:
  #print("dish  value:",mydict['dish'])
```

Listing 5.10 contains a block of code that reads the existing value of past from `mydict`, updates its value, and then saves its new value. The final code block in Listing 5.10 reads the stored contents and displays the key/value pairs. Now launch the code in Listing 5.10 to see the following output:

```
old: pasta
new: more pasta
pasta: more pasta
```

Check the online documentation for information regarding other functionality that is available through sqlitedict.

SUMMARY

This chapter introduced you to non-relational databases and some of their advantages. You learned about NoSQL and a NoSQL database called MongoDB. You saw how to create a database in MongoDB, how to create a collection, and how to populate the collection with documents. You also saw how to query data from a MongoDB collection and how to delete a document from a collection.

Next, you learned about Compass (a GUI tool for MongoDB) and PyMongo, which is a Python distribution for working with MongoDB. You also learned about DynamoDB, which is a NoSQL database from Amazon. Then you saw how to read MySQL data into a Pandas data frame and then save the data frame as an Excel spreadsheet.

In addition, you learned about SQLite, which is a command line tool for managing databases that is available on mobile devices. Then you learned about related tools, such as SQLiteStudio (an IDE for sqlite), DB Browser, and SQLiteDict.

CHAPTER 6

MISCELLANEOUS TOPICS

This chapter contains an overview of a highly eclectic mixture of SQL and RDBMS topics, such as normalization, schemas, performance tuning, and third-party tools such as MySQL Workbench for managing databases via a GUI interface. Although numerous topics in this chapter are relevant to a DBA, it's still worthwhile for you to be acquainted with these topics.

You can treat sections in this chapter as optional if you do not have an immediate need to acquire the information provided in those sections. Your time will obviously be better spent focusing on the portions of this chapter that are directly relevant to you.

The first section discusses how to manage database users: specifically, how to create users and how to drop users. Next, you will learn about the concept of roles in MySQL, followed by details about creating roles, granting privileges, revoking roles, and dropping roles. This section also contains information about stored procedures, stored functions, and SQL triggers.

The second section continues the explanation of normalization that was introduced briefly in Chapter 1. You will learn about the rules for the first three normal forms regarding tables in RDBMSs, which is most likely sufficient for your needs because the third normal form is sufficient for the majority of applications. You will also learn about denormalization, and why it can improve performance. This section also introduces schemas and transactions, which involve the keywords `COMMIT`, `ROLLBACK`, and `SAVEPOINT`. You will then learn about MySQL Workbench and some of its rich set of features, such as reverse engineering a database schema. In fact, this IDE can easily manage the details of exporting databases, such as the `mytools` database, as well as importing CSV files into database tables.

The third section introduces you to aspects of database optimization, performance tuning considerations, and SQL query optimization. You will also

learn about table fragmentation and table partitioning. In addition, you will learn about EXPLAIN plans and how they can be useful to you.

The fourth section introduces you to scaling an RDBMS, which can involve sharding and federation. This section also discusses MySQL caching and how it can be disabled. In addition, you will learn about the MySQL engines that are available.

The remaining portion of this chapter is an eclectic mix of topics: distributed databases, the CAP theorem, MySQL command line utilities, database backups and upgrades, character sets, regular expressions, and recursion in MySQL.

MANAGING USERS

MySQL enables you to define users with various roles (discussed later) that specify the privileges users have with respect to a database and its tables. There are many options for creating users, and this section describes a few of those options. If you need additional information, you can read the online documentation.

Listing Current Users

If you want to view the currently defined users in MySQL, the following SQL statement displays a list of users in a MySQL instance:

```
mysql>
SELECT USER
FROM mysql.user;
+------------------+
| user             |
+------------------+
| mysql.infoschema |
| mysql.session    |
| mysql.sys        |
| root             |
+------------------+
4 rows in set (0.000 sec)
```

Creating and Altering MySQL Users

The following SQL statements create user oswald as well as user mary in a MySQL instance:

```
mysql>CREATE USER 'oswald'@'localhost' COMMENT 'Account for Oswald';
Query OK, 0 rows affected (0.047 sec)

mysql>CREATE USER 'mary'@'localhost' COMMENT 'Account for Mary';
Query OK, 0 rows affected (0.047 sec)

mysql> ALTER USER 'mary'@'localhost'
ATTRIBUTE '{"fname":"Mary", "lname":"Smith"}';
Query OK, 0 rows affected (0.14 sec)
```

```
mysql> ALTER USER 'mary'@'localhost'
ATTRIBUTE '{"email":"msmith@example.com"}';
Query OK, 0 rows affected (0.12 sec)
```

Now let's confirm the details of the user `mary` by launching the following SQL statement:

```
SELECT USER,
ATTRIBUTE->>"$.fname" AS 'First Name',
ATTRIBUTE->>"$.lname" AS 'Last Name',
ATTRIBUTE->>"$.email" AS 'Email',
ATTRIBUTE->>"$.comment" AS 'Comment'
FROM INFORMATION_SCHEMA.USER_ATTRIBUTES
WHERE USER='mary'
AND HOST='localhost';
+------+------------+-----------+--------------------+------------------+
| USER | First Name | Last Name | Email              | Comment          |
+------+------------+-----------+--------------------+------------------+
| mary | Mary       | Smith     | msmith@example.com | Account for Mary |
+------+------------+-----------+--------------------+------------------+
1 row in set (0.002 sec)
```

The following SQL statement enables you to view more detailed information regarding the current MySQL users:

```
SELECT user, host, account_locked, password_expired
FROM mysql.user;
+------------------+-----------+----------------+------------------+
| user             | host      | account_locked | password_expired |
+------------------+-----------+----------------+------------------+
| mary             | localhost | N              | N                |
| mysql.infoschema | localhost | Y              | N                |
| mysql.session    | localhost | Y              | N                |
| mysql.sys        | localhost | Y              | N                |
| oswald           | localhost | N              | N                |
| root             | localhost | N              | N                |
+------------------+-----------+----------------+------------------+
6 rows in set (0.000 sec)
```

The following SQL statement displays a list of currently logged in users:

```
SELECT user, host, db, command
FROM information_schema.processlist;
+-----------------+-----------+---------+---------+
| user            | host      | db      | command |
+-----------------+-----------+---------+---------+
| root            | localhost | mytools | Query   |
| event_scheduler | localhost | NULL    | Daemon  |
+-----------------+-----------+---------+---------+
2 rows in set (0.006 sec)
```

Dropping MySQL Users

Dropping a MySQL user is illustrated in the following SQL statements that create the user `pasta` and then drop the user `pasta`:

```
MySQL [(none)]> create user 'pasta'@'localhost';
Query OK, 0 rows affected (0.002 sec)
```

```
MySQL [(none)]> drop user pasta;
ERROR 1396 (HY000): Operation DROP USER failed for 'pasta'@'%'
MySQL [(none)]> drop user pasta@localhost;
Query OK, 0 rows affected (0.004 sec)
```

At this point you know how to list users, create users, alter users, and drop users in MySQL. Consider this question: how do you assign different privileges to a large set of users in an efficient manner that's also easily managed? The answer involves the concept of roles, which is the topic of the next section.

WHAT ARE ROLES?

Roles are named collections of privileges that can be granted to user accounts. Each role can have a different set of privileges (specified by you) in order to control the access rights that are granted to different users. The following operations can be performed with roles and users:

- create and drop roles
- grant privileges to roles
- revoke privileges from roles
- grant roles to users
- revoke roles from users

For example, users of a Web application typically have fewer access privileges than application developers, who in turn generally have full access (i.e., read and write) to the tables in an underlying database. Assigning different sets of privileges to these two groups of users is simple: create a user role and a developer role with appropriate privileges and then grant the correct role to each type of user.

In fact, you can assign *multiple* roles to a given user, which enables a more fine-grained level of control. If need be, you can revoke one or more roles from users whenever it's necessary to do so.

Create Roles and Grant Privileges

This section contains simple examples of creating roles and granting privileges. Let's start by creating the role `developers` with the following SQL statement:

```
CREATE ROLE developers;
```

Grant the role `developers` to specific users as follows:

```
GRANT developers to Sara;
GRANT developers to Dave;
```

Grant `INSERT` privilege for table `customers` to the role `developers`:

```
GRANT INSERT ON CUSTOMER TO developers;
```

Grant SELECT privilege for table customers to the role developers:

```
GRANT SELECT ON CUSTOMER TO developers;
```

Grant SELECT privilege on a view to the role developers:

```
CREATE VIEW v_customers AS
SELECT last_name, first_name FROM customers;
GRANT SELECT ON v_customers TO developers;
```

You can also grant DELETE or UPDATE (or both) to a role. You can also create multiple roles with a single statement, as shown here:

```
CREATE ROLE 'all_privs', 'read_privs', 'write_privs';
```

Next, assign all privileges on all the tables and views in the mytools database to the all_privs role with this GRANT statement:

```
GRANT ALL ON mytools.* TO 'all_privs';
```

Assign SELECT privilege on all the tables and views in the mytools database to the app_read role:

```
GRANT SELECT ON mytools.* TO 'app_read';
```

Next, assign INSERT, UPDATE, and DELETE privileges and exclude SELECT privileges on all the tables and views in the mytools database to the appl_write role:

```
GRANT INSERT, UPDATE, DELETE ON mytools.* TO 'appl_write';

MySQL [mytools]> select current_role();
+----------------+
| current_role() |
+----------------+
| NONE           |
+----------------+
1 row in set (0.002 sec)

MySQL [mytools]> select user();
+----------------+
| user()         |
+----------------+
| root@localhost |
+----------------+
1 row in set (0.001 sec)

MySQL [mytools]> select user(), current_date();
+----------------+----------------+
| user()         | current_date() |
+----------------+----------------+
| root@localhost | 2021-06-17     |
+----------------+----------------+
1 row in set (0.001 sec)
```

It's also possible to grant a role the ability to grant privileges to other roles, as shown here:

```
GRANT DELETE ON customers TO role-name WITH GRANT OPTION;
```

Revoke Roles and Drop Roles

Specify `DROP ROLE` to drop roles, and those roles will no longer be available to any users that were assigned those roles:

```
DROP ROLE 'app_read', 'app_write';

MySQL [mytools]> select current_role();
+----------------+
| current_role() |
+----------------+
| NONE           |
+----------------+
1 row in set (0.002 sec)

mysql> SHOW GRANTS FOR 'app_write';
+---------------------------------------+
| Grants for app_write@%                |
+---------------------------------------+
| GRANT USAGE ON *.* TO 'app_write'@'%' |
+---------------------------------------+
```

WHAT IS A USER-DEFINED FUNCTION?

User-defined functions in SQL are similar to functions in any other programming language that accept parameters, perform complex calculations, and return a value. They are written to use the logic repetitively whenever required. There are several types of SQL user-defined functions:

- Scalar Function: a function that returns a single scalar value
- Table Valued Functions: a table-valued function that returns a table as output
- Inline: returns a table data type based on one `SELECT` statement
- Multi-statement: returns a tabular result-set but (unlike inline) can include multiple `SELECT` statements

WHAT IS A STORED PROCEDURE?

Stored procedures are subroutines for managing data in RDBMSs, and they are stored in the database data dictionary. Some of the features of stored procedures are as follows:

- they can only be invoked in the database
- they prevent users from accessing data directly
- they provide additional security

- they support imperative programming
- users are granted access to stored procedures
- access to stored procedures can be revoked

There are several important advantages to using MySQL stored procedures:

- Faster execution
- Greater Security
- Improved performance
- Portability
- Reusability/transparency

However, there are also some disadvantages to using MySQL stored procedures:

- Difficult to debug
- Increased maintenance complexity
- Increased memory consumption
- Unsuitable for complex business logic

Experiment with stored procedures using best practices, and you will be in a better position to assess how well they meet your needs and also the level of effort required to maintain or enhance them.

IN and OUT Parameters in Stored Procedures

An `IN` parameter passes a value into a procedure, and any changes that the procedure makes to `IN` parameters are not visible to the calling program. By contrast, an `OUT` parameter passes a value from the procedure back to the calling program. Finally, an `INOUT` parameter has these properties:

- It's initialized by the calling program.
- It can be modified by the procedure.
- Any change in the procedure is visible to the calling program.

A simplified and more concrete syntax for stored procedures is shown here:

```
Delimiter //
Create Procedure myprocedure()
BEGIN
  Select column_name from my_table;
END//

DELIMITER ;
Call myprocedure();
```

Now let's proceed to the next section to learn how to create a stored procedure in MySQL.

A Simple Stored Procedure

Listing 6.1 shows the contents of `stored1.sql` that illustrates how to define a stored procedure for selecting the rows in the table `user`.

LISTING 6.1: stored1.sql

```
use mytools;
\! echo '=> Rows from user via SQL Statement:';

SELECT * FROM user;

DROP PROCEDURE IF EXISTS allrows;

-- stored procedure to select rows
Delimiter //
Create Procedure allrows()
BEGIN
  SELECT * FROM user;
END//

DELIMITER ;

\! echo '=> Rows from user via Stored Procedure:';
Call allrows();
```

Listing 6.1 starts by specifying the `mytools` database, prints a comment on the screen, and then executes a SQL statement displays the contents of the `user` table. The next code snippet drops the `allrows` procedure (if it exists), and then defines the same procedure whose code block simply displays the contents of the table `user`.

This admittedly simple procedure is sufficient for confirming that the code *does* return the correct set of rows. The last portion of Listing 6.1 invokes the stored procedure. Navigate to the SQL prompt and launch the SQL script in Listing 6.1 with the following command:

```
source stored1.sql;
```

The generated output is shown here:

```
Database changed
=> Rows from user via SQL Statement:
+---------+--------------------+
| user_id | user_title         |
+---------+--------------------+
|    1000 | Developer          |
|    2000 | Project Lead       |
|    3000 | Dev Manager        |
|    4000 | Senior Dev Manager |
+---------+--------------------+
4 rows in set (0.000 sec)

Query OK, 0 rows affected (0.002 sec)
Query OK, 0 rows affected (0.001 sec)
```

```
=> Rows from user via Stored Procedure:
+---------+--------------------+
| user_id | user_title         |
+---------+--------------------+
|    1000 | Developer          |
|    2000 | Project Lead       |
|    3000 | Dev Manager        |
|    4000 | Senior Dev Manager |
+---------+--------------------+
4 rows in set (0.000 sec)

Query OK, 0 rows affected (0.000 sec)
```

As you can see, the output produced by the SQL statement and the stored procedure is the same.

Listing 6.2 shows the content of `double_number.sql` that illustrates how to define a stored procedure that doubles the integer-valued input argument.

LISTING 6.2: double_number.sql

```
use mytools;
DROP PROCEDURE IF EXISTS double_number;

DELIMITER //
CREATE PROCEDURE double_number(IN N INT, INOUT result INT)
BEGIN
  SET result := N * 2;
END //

DELIMITER ;
SET @result=0;
Call double_number(10,@result);
SELECT @result;
Call double_number(17,@result);
SELECT @result;
```

Listing 6.2 starts by specifying the `mytools` database, prints a comment on the screen, and then drops the `double_number` procedure (if it exists). The next portion of Listing 6.2 defines the same procedure whose code block doubles the value the input parameter `N`. The last portion of Listing 6.2 invokes the procedure with the value 10 for `N`. Navigate to the SQL prompt and launch the SQL script in Listing 6.2 with the following command:

```
MySQL [mytools]> source double_number.sql;
```

The generated output is shown here:

```
Database changed
Query OK, 0 rows affected (0.004 sec)
Query OK, 0 rows affected (0.003 sec)
Query OK, 0 rows affected (0.000 sec)
Query OK, 0 rows affected (0.000 sec)
```

```
+----------+
| @result |
+----------+
|       20 |
+----------+
1 row in set (0.000 sec)

Query OK, 0 rows affected (0.000 sec)

+----------+
| @result |
+----------+
|       34 |
+----------+
1 row in set (0.000 sec)
```

WHAT IS A STORED FUNCTION?

Stored functions are similar to stored procedures: the former is invoked with a function call, whereas the latter is invoked via a `CALL` statement. In addition, you can replace an argument of a SQL statement with a stored function. The term *stored routines* refers to stored procedures and stored functions. Some of the features of stored functions are

- They can only be invoked in the database.
- They prevent users from accessing data directly.

A Simple Stored Function

Listing 6.3 shows the content of `stored_function1.sql` that illustrates how to define and invoke a stored function in MySQL.

LISTING 6.3: stored_function1.sql

```
use mytools;

DROP FUNCTION IF EXISTS olympic_tier;
DELIMITER //

CREATE FUNCTION olympic_tier(medals INT)
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE medal_level VARCHAR(20);

    IF medals >= 30 THEN
                SET medal_level = 'TIER 1';
    ELSEIF medals >= 20 THEN
                SET medal_level = 'TIER 2';
    ELSE
                SET medal_level = 'TIER 3';
    END IF;
    -- return the customer level
```

```
    RETURN (medal_level);
END //
DELIMITER ;

SELECT country, count, olympic_tier(count)
FROM olympics
ORDER BY country;
```

Listing 6.3 starts by specifying the `mytools` database, then drops the function (if it already exists), and then defines the contents of the function. This function returns `TIER 1`, `TIER 2`, or `TIER 3`, depending on whether the number of medals (an input parameter) is at least 30, at least 30, or at most 19, respectively.

The final portion of Listing 6.3 contains a SQL statement that displays the country, count, and the tier of the country via the stored function `olympic_tier`. Launch the code in Listing 6.3 from the MySQL prompt as follows:

```
MySQL [mytools]> source stored_function1.sql;
```

The preceding command will display the following output:

```
Database changed
Query OK, 0 rows affected (0.052 sec)
Query OK, 0 rows affected (0.002 sec)

+---------+-------+---------------------+
| country | count | olympic_tier(count) |
+---------+-------+---------------------+
| CHINA   |    38 | TIER 1              |
| CHINA   |    32 | TIER 1              |
| CHINA   |    18 | TIER 3              |
| JAPAN   |    27 | TIER 2              |
| JAPAN   |    14 | TIER 3              |
| JAPAN   |    17 | TIER 3              |
| ROC     |    20 | TIER 2              |
| ROC     |    28 | TIER 2              |
| ROC     |    23 | TIER 2              |
| UK      |    22 | TIER 2              |
| UK      |    21 | TIER 2              |
| UK      |    22 | TIER 2              |
| USA     |    39 | TIER 1              |
| USA     |    41 | TIER 1              |
| USA     |    33 | TIER 1              |
+---------+-------+---------------------+
15 rows in set (0.001 sec)
```

WHAT ARE SQL TRIGGERS?

A *trigger* is a database object that executes when a particular event occurs for a permanent table. If need be, you can define multiple triggers, even with the same event, on the same table. Such triggers are executed in the order in which the triggers were defined.

However, you can change the order of execution via the FOLLOWS and PRECEDES keywords. For example, if trigger A follows trigger B, then A is executed after B; if A precedes B, then A is executed before B. You can define a maximum of six triggers on a MySQL table, which are listed below as pairs of before/after triggers:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE

A Simple MySQL Trigger

Listing 6.4 shows the content of trigger1.sql that illustrates how to define a trigger that updates the value of an attribute in the table average_val after one or more rows are inserted into the account table.

LISTING 6.4: trigger1.sql

```
use mytools;

-- 1) drop, recreate, and populate table account:
DROP TABLE IF EXISTS account;
CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
INSERT INTO account VALUES(1000,1.00);
INSERT INTO account VALUES(1000,2.00);
INSERT INTO account VALUES(1000,3.00);
SELECT * FROM account;

-- 2) drop, recreate, and populate table average_val:
DROP TABLE IF EXISTS average_val;
CREATE TABLE average_val (average double);
INSERT INTO average_val VALUES(1.00);
SELECT * FROM average_val;

-- 3) drop and redefine trigger inserted_sum:
DROP TRIGGER IF EXISTS inserted_sum;
CREATE TRIGGER update_table_avg AFTER INSERT ON account
FOR EACH ROW SET @sum = @sum + NEW.amount;
UPDATE average_val SET average = (SELECT AVG(amount) FROM account);
SELECT * FROM average_val;
```

Listing 6.4 contains three sections, each of which starts with a comment statement that describes its purpose. For example, the first section drops, recreates, and populates the table account, and the second section does so for the table average_val.

The third section executes the trigger update_table_avg that updates the value of the average attribute in the table average_val. Launch the SQL script in Listing 6.4 with the following command:

```
source trigger1.sql;
```

The generated output from the SQL statements is shown here:

```
+----------+--------+
| acct_num | amount |
+----------+--------+
|     1000 |   1.00 |
|     1000 |   2.00 |
|     1000 |   3.00 |
+----------+--------+
3 rows in set (0.000 sec)

+---------+
| average |
+---------+
|       1 |
+---------+
1 row in set (0.000 sec)

+---------+
| average |
+---------+
|       2 |
+---------+
1 row in set (0.000 sec)
```

MYSQL ENGINES

The SQL scripts in this book that create MySQL tables do not specify a database engine. However, MySQL supports several database engines, and the two most popular engines are InnoDB and MyISAM. If you want to see the list of engines in your instance of MySQL, enter the following SQL statement from the SQL prompt:

```
MySQL [mytools]> SHOW ENGINES;
```

The following SQL statement displays the tables in the `mytools` database and the MySQL engine for each table:

```
MySQL [mytools]>
SELECT TABLE_NAME, ENGINE
FROM   information_schema.TABLES
WHERE  TABLE_SCHEMA = 'mytools';
+--------------------+--------+
| TABLE_NAME         | ENGINE |
+--------------------+--------+
| account            | InnoDB |
| courses            | InnoDB |
| curr_exchange_rate | InnoDB |
| currencies         | InnoDB |
| cust_history       | InnoDB |
| customers          | InnoDB |
| employees          | InnoDB |
| FRIENDS            | InnoDB |
```

```
| FRIENDS2            | InnoDB |
| item_desc           | InnoDB |
| japn1               | InnoDB |
| japn2               | InnoDB |
| japn3               | MyISAM |
| japn_emps           | InnoDB |
| json1               | InnoDB |
| line_items          | InnoDB |
| new_items           | InnoDB |
| people              | InnoDB |
| people2             | InnoDB |
| purchase_orders     | InnoDB |
| sample              | InnoDB |
| schedule            | InnoDB |
| students            | InnoDB |
| temp_cust2          | InnoDB |
| user                | InnoDB |
| user2               | InnoDB |
| user3               | InnoDB |
| weather             | InnoDB |
| weather2            | InnoDB |
+---------------------+--------+
29 rows in set (0.002 sec)
```

Notice that the table `japn3` uses the `MyISAM` engine, whereas the other tables in the `mytools` database use the `InnoDB` engine.

More information regarding MySQL database engines is available online:

https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html

https://dev.mysql.com/doc/refman/8.0/en/storage-engines.html

WHAT IS NORMALIZATION?

Normalization in an RDBMS refers to a methodology for defining the structure of tables in a way that reduces data redundancy and helps to maintain data integrity. The way to achieve normalization involves subdividing a given table into smaller tables when the given table contains multiple copies of the same data.

For example, the `customers` table contains the information pertaining to each customer, and each customer is assigned a unique `cust_id` value. Whenever a customer makes a new purchase, a new row is inserted into the `purchase_orders` table that contains the `cust_id` value of the customer that made the purchase.

The personal details of each customer appear only **once** *in the* `customers` *table instead of repeating the same information in every purchase that is made by each customer. As a result, any updates to a customer's personal details are made in only one location, which helps to maintain data integrity.*

Edgar Codd invented the relational model in RDBMSs, which consists of the following normal forms that are increasingly restrictive from first to sixth normal form:

- 1NF (First Normal Form)
- 2NF (Second Normal Form)
- 3NF (Third Normal Form)
- BCNF (Boyce-Codd Normal Form)
- 4NF (Fourth Normal Form)
- 5NF (Fifth Normal Form)
- 6NF (Sixth Normal Form)

Second normal form (2NF) is more restrictive than first normal form (1NF), and 3NF is more restrictive than 2NF, and so forth. In general, 3NF is suitable for applications that store data in an RDBMS. The first normal form is the minimum requirement: the attributes consist of atomic elements instead of sets of elements. For example, the first name and last name values are stored in different attributes. More precisely, first normal form enforces these criteria:

- no repeating groups in any database table
- a separate table for any set of related data
- a primary key for each set of related data

In general, the goal for applications that have an RDBMS data store is to achieve third normal form. In addition, the remaining normal forms (fourth, fifth, and sixth) are more advanced and they can be useful if you are an application DBA (but not for beginners in SQL).

What is Denormalization?

Denormalization refers to converting a normalized table (or tables) to denormalized form. Despite the importance of database normalization, sometimes you can improve the performance of an application by denormalizing a database tables. However, *determining* which table (or tables) to denormalize is an advanced topic, typically performed by a senior application DBA.

WHAT ARE SCHEMAS?

The meaning of the word *schema* depends on the context in which it's used. For example, XML includes XML schemas, which are XML documents that describe the structure of *other* XML documents that "conform" to the given XML schema.

However, in this section (and elsewhere in this book), a schema refers to an RDBMS schema for *databases*. There are three types of schemas in RDBMSs, from abstract to concrete, as shown in the following list:

- conceptual schema
- logical schema
- physical schema

A *conceptual schema* is the most abstract of the three types of schema, and it consists of high-level data constructs that involve the semantics of an organization.

A *logical schema* includes entities such as tables, along with their attributes and relationships between entities. A logical schema is also called a *logical data model*, which is a data model of a specific problem. Note that a logical schema does not contain any hardware-specific restrictions.

A *physical schema* includes all the objects that have been defined for a database: tables, columns, keys, data types, validation rules, database triggers, stored procedures, and constraints. A physical schema is a SQL script that contains the complete definition of every entity (and relationships) in a database.

A physical schema is useful when you want to export a database from one environment and recreate that database in a different environment. For example, the SQL file `mytools.sql` that is available for this chapter is a physical design of the `mytools` database.

MYSQL WORKBENCH

The Community Edition Workbench is a free GUI-based tool that enables you to create new databases and manage existing databases in a GUI environment. The Workbench supports many other features, such as performance monitoring, reverse generating schemas for databases, database backups, and database exports. The Workbench can be found online:

https://www.mysql.com/products/workbench/

Note that the version of Workbench that you download must be compatible with the version of the operating system on your machine. An earlier version of Workbench can be found online:

https://downloads.mysql.com/archives/workbench/

The preceding website displays the version of the operating system that is compatible for a given version of Workbench.

Exporting a Schema in Workbench

This section shows you how to export the `mytools` database. Before we export a database using MySQL Workbench, let's see how to do so from the command line with the `mysqldump` utility:

```
mysqldump -u root -p -R mytools > mytools.sql
```

However, you might encounter the following error message:

```
mysqldump: unknown variable 'local_infile=1'
```

You can search online and find many suggestions for resolving this error. However, if none of those solutions solves this issue for your system, use MySQL Workbench to export the `mytools` database.

The first step is to launch MySQL Workbench and then navigate to the "Data Export" tab. For your convenience, Figure 6.2 shows a screenshot of the screen where you can export the `mytools` database from MySQL Workbench.

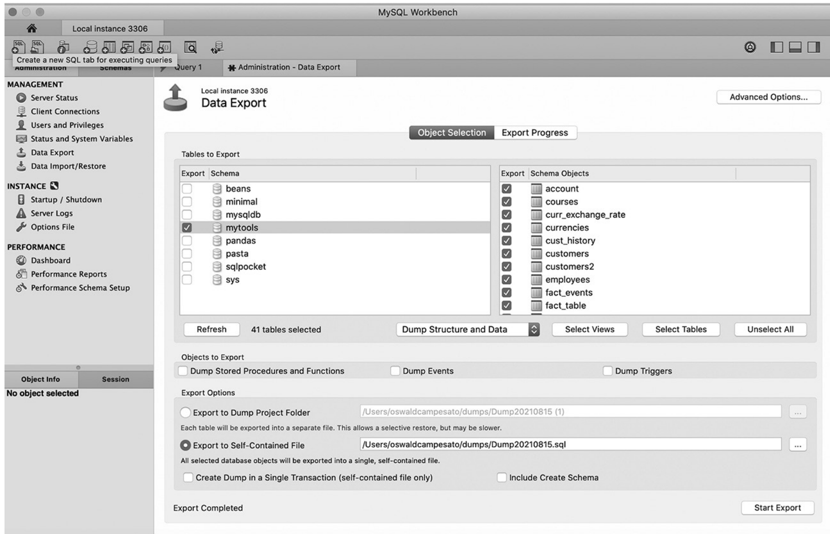

FIGURE 6.1 Exporting the `mytools` database.

Next, notice two labeled radio buttons near the bottom of Figure 6.1, along with editable text fields where you can specify the `export` directory:

```
Export to Dump Project Folder
Export to Self-Contained File
```

If you select the *first* option that is listed above, then MySQL Workbench will generate a *separate* SQL file for *every* table in the database. If you select the *second* option that is listed above, then MySQL Workbench will generate a *single* `SQL` file that contains SQL statements for every table in the database.

If you wish, you can choose the first option and then the second option (the order is irrelevant) to generate a single SQL file with all the table definitions as well as a set of SQL files that contain a single table definition.

Creating a Schema in Workbench

MySQL Workbench can be used to create a schema for the `mytools` database in MySQL Workbench. In fact, you can also reverse engineer a database schema from an existing database.

Figure 6.2 shows a screenshot of some of the tables in the `mytools` database that are visible in MySQL Workbench.

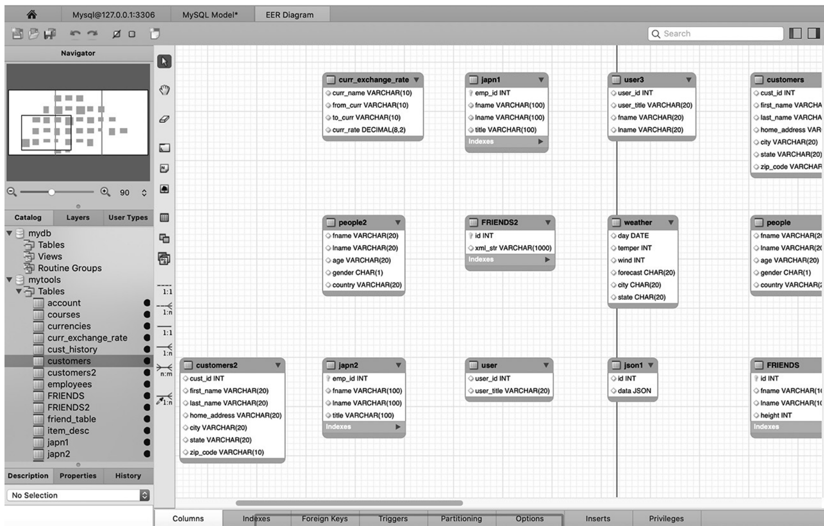

FIGURE 6.2 A visual display of tables in the `mytools` database.

ERM and Tools

ERM is an acronym for Entity Relationship Modeling, which you can think of as a diagram that contains entities (such as tables) and relationships between tables (one-to-many, many-to-many, and so forth).

Entities and relationships are somewhat analogous to nouns and verbs, respectively. For example, the tables `customers`, `purchase_orders`, `line_items`, and `item_desc` are entities. As you learned in Chapter 1, there is a one-to-many relationship between the following pairs of tables:

```
customers and purchase_orders
purchase_orders and line_items
```

An Entity Relationship Diagram (ERD) is a standard way to display the logical structure of RDBMS tables in a visual manner. Various tools are available for creating ERDs, including the following tool for Macbooks:

https://www.conceptdraw.com/How-To-Guide/erd-entity-relationship-diagram-software-for-mac

A list of additional ERD tools, along with their description and pricing options (many are free) is available online:

https://chartio.com/learn/databases/7-free-database-diagramming-tools-for-busy-data-folks/

WHAT IS A TRANSACTION?

In the database world, a *transaction* is an atomic unit of work, which means that a transaction only succeeds when its "components" succeed. Otherwise, the transaction fails. Recall the example in Chapter 1 of a transaction that transfers money from a savings account to a checking account: the

transaction is completed when both table-related updates are successful. The SET TRANSACTION statement enables you to specify a particular lock on tables or rows in a table, which is called the *isolation level*.

You can also set a READ lock or a WRITE lock on tables or sets of rows in a table, each of which imposes restrictions on what other users can do when a lock has been set on an object. Different RDBMSs have their own mechanism for locking database objects.

The COMMIT and ROLLBACK Statements

Invoke the COMMIT keyword when a transaction has successfully completed and you want to persist the result of that transaction. By contrast, the ROLLBACK keyword restores the database to the state before you performed the most recent transaction. If an error occurs during a COMMIT statement, it might be necessary to roll back the transaction, re-execute the SQL statement and then issue the COMMIT statement. If an error occurs during a ROLLBACK statement, you can re-issue the ROLLBACK statement after the system has been restored.

The SAVEPOINT Statement

The ROLLBACK statement cancels an entire transaction. However, more recent versions of SQL support the SAVEPOINT statement that enables you to roll back a transaction to a specified save point in the given transaction. Hence, you can perform partial rollbacks as well as full roll backs in SQL. In addition, you can specify multiple SAVEPOINTs in a SQL transaction. Here is the syntax for a SAVEPOINT statement:

```
SAVEPOINT savepoint_name;
```

Another variant involves specifying the ROLLBACK statement, as shown here:

```
ROLLBACK TO SAVEPOINT savepoint_name;
```

The SAVEPOINT statement can be useful in multi-step transactions where the execution of a sub-task produces unfavorable results. You can roll back to a specified SAVEPOINT and resume the execution of another portion of the transaction.

In addition, a transaction completes with a COMMIT if it's successful; otherwise, it completes with a ROLLBACK statement. Some databases (such as ORACLE) also support nested transactions, which means that an on-going transaction can execute a second transaction before the initial transaction is completed.

Furthermore, you can release a particular SAVEPOINT via the RELEASE SAVEPOINT statement, which removes the specified SAVEPOINT from the set of SAVEPOINTs of the current transaction. Moreover, no commit or rollback occurs, an error occurs if the specified SAVEPOINT does not exist. In summary, MySQL supports the following transaction-related keywords:

- START TRANSACTION statements (BEGIN or BEGIN WORK are aliases)
- COMMIT (commit the current transaction)
- ROLLBACK (roll back the current transaction)
- SET autocommit (disable or enable the auto-commit for the current transaction)

The default action is for MySQL to automatically commit changes to a database.

DATABASE OPTIMIZATION AND PERFORMANCE

Database optimization is an important task that involves many factors, such as manually modifying SQL statements, redefining database tables, creating new indexes, and tuning built-in database parameters.

Optimization strategies changed from older rule-based optimization to cost-based optimization, where the latter involves collecting statistics regarding the frequency of accessing specific tables.

If you are motivated to learn about performance tuning (whether by choice or as part of your job), some useful tips for database tuning are available online:

https://www.tecmint.com/mysql-mariadb-performance-tuning-and-optimization/

Perform an online search and you will find many blog posts and links for open source (as well as commercial) tools for performance tuning.

Performance Tuning Considerations

Performance tuning can involve deciding whether to keep tables in RAM (often called "pinning" a table). Candidate tables are tables that are static (i.e., they change rarely or never) and are frequently accessed. One candidate is the item_desc table because this table is unaffected by any customer transactions. Over a period of time, the contents of an item_desc table will undergo fewer updates and will have a decreasing number of new insertions. Hence, it's worth investigating if a significant performance improvement in an application will occur if this table is pinned in RAM.

Next, collect two sets of execution times for SQL queries that involve the item_desc table: one set is for the item_desc table "pinned" in memory, and the other set is for the item_desc table that is located on disk. Analyze those results to see whether it's worthwhile to pin the item_desc table. Repeat the preceding process for any other tables that are frequently accessed and are rarely updated.

Given the emphasis on normalization in this book, it might seem ironic or counter-intuitive that sometimes denormalizing a table *can* improve performance. Determining whether it's worthwhile to do so typically involves an experienced DBA who can make an assessment and suggest feasible options.

Another scenario pertains to smaller tables: if they are frequently accessed, consider "pinning" their contents in memory, which is obviously faster than

searching through a table that is in secondary storage. Indexes on tables are easy to create, but knowing which indexes will be most effective is not necessarily obvious in every case.

Perform an online search and you will find an assortment of blog posts and links for open source (as well as commercial) tools for performance tuning.

SQL QUERY OPTIMIZATION

This section provides a high-level view of query optimization. Database optimization often refers to making changes to SQL statements so that they will execute faster than the original SQL statements.

The following list contains various techniques for improving SQL query performance, some of which are briefly discussed in this section:

- Define a suitable index (or indexes) on tables;
- Specify index hints (ex: `USE INDEX`)
- Analyze the `JOIN` order
- Simplify multi-level queries with multiple subqueries
- Execute and analyze an `EXPLAIN PLAN`
  - execute `ANALYZE TABLE`
  - analyze `SHOW TABLE STATUS`
  - Denormalize a table (requires significant expertise)

Analyzing SQL Queries for Their Performance

Instead of using a trial-and-error approach, take advantage of IDEs that provide a list of the most time-consuming SQL queries in your application. IDEs display SQL statements in descending order of execution time, starting from the most computationally expensive query to the least expensive query.

Useful tools for monitoring database performance are as follows:

https://www.dnsstuff.com/mysql-optimize-database

https://www.solarwinds.com/database-performance-analyzer/use-cases/mysql-optimization

https://www.solarwinds.com/database-performance-monitor/integrations/mysql-monitoring

Performance Tuning Tools

This section contains an assortment of links for performance tuning tools, from command line tools to GUI tools, some of which are free and others which have a free trial version.

This website provides performance tuning tips for databases:

https://haydenjames.io/mysql-performance-tuning-tips-scripts-tools/

MySQLTuner is a Perl script that you can download from Github:

https://github.com/major/MySQLTuner-perl

The Persona toolkit (command line instead of GUI) is available online:

https://www.percona.com/software/database-tools/percona-toolkit

Some useful tips for database tuning are available online:
https://www.tecmint.com/mysql-mariadb-performance-tuning-and-optimization/

The Persona toolkit is available online (not available for Mac):
https://www.percona.com/downloads/percona-toolkit/LATEST/

Cost-Based Optimizers (optional)

MySQL and other RDBMSs provide an *optimizer*, which determines the most efficient way to execute a SQL query. As a side note, optimizers used to be rule-based optimizers, but during the 1990s, there was a switch from rule-based optimizers to cost-based optimizers.

Cost-based optimization, where the latter involves collecting "statistics" regarding the frequency of accessing specific tables. A cost-based optimizer can involve a table of queries that have been executed over a period of time, which are used to determine the pattern of execution of SQL queries, thereby providing information to the optimizer for the purpose of anticipating which SQL queries are more likely to be executed in the future.

Table Fragmentation

Table fragmentation means that the data in a database table is stored in non-contiguous memory. When such tables become large and are frequently accessed, the result can be performance degradation.

There are two additional factors to consider: the column size and the columns in the `WHERE` clause of SQL statements. You can view table size by executing the following command from the command line:

```
mysqlshow –status <dbname>
```

Another useful SQL statement for finding indexes associated with a table is the following:

```
MySQL [mytools]> show index from <table_name>;
```

The preceding SQL statement enables you to check the indexes and their relative cardinality.

Table Partitioning

MySQL supports database partitioning via hashing functions, which avoids bottlenecks and can simplify maintenance. Depending on your application, you might discover that a portion of a particular table is accessed much more frequently than the other attributes in that table. *Table partitioning* refers to placing the highly accessed portions of that table in a separate table, which can help to keep the highly accessed table in memory. A DBA can assist in the task of determining the most frequently accessed tasks in the tables of your application.

Remember that splitting the table into two tables involves defining a suitable foreign key, and most likely rewriting one or more of the SQL statements in the application. In addition, table partitioning can be performed in conjunction with (or separate from) table sharding. If possible, use a test environment to perform the preceding changes so that you can obtain benchmarks to compare the before-and-after performance numbers.

WHAT IS AN EXPLAIN PLAN?

The `EXPLAIN` statement provides information about how MySQL executes SQL statements. Specifically, MySQL provides the details of how it would process a given SQL statement, such as how tables in the SQL statement are joined (if any) and the order in which they are joined.

An `EXPLAIN` statement can be generated with various SQL statements, such as `SELECT`, `DELETE`, `INSERT`, `REPLACE`, and `UPDATE` statements.

An `EXPLAIN` plan displays the actual order of execution of a SQL statement. In addition, it's worthwhile to execute `ANALYZE TABLE <table-name>` in MySQL, an example of which is shown here:

```
MySQL [mytools]> ANALYZE TABLE customers;
+-------------------+---------+----------+----------+
| Table             | Op      | Msg_type | Msg_text |
+-------------------+---------+----------+----------+
| mytools.customers | analyze | status   | OK       |
+-------------------+---------+----------+----------+
1 row in set (0.005 sec)
```

Another useful SQL statement is `SHOW TABLE STATUS`, an example of which is shown below:

```
+----------------+--------+---------+------------+------+-
--------------+-------------+----------------+-----------
---+-----------+----------------+---------------------+----
-----------------+-----------+--------------------+-------
---+---------------+---------+
| Name           | Engine | Version | Row_format | Rows
| Avg_row_length | Data_length | Max_data_length | Index_
length | Data_free | Auto_increment | Create_time         |
Update_time         | Check_time | Collation          |
Checksum | Create_options | Comment |
+----------------+--------+---------+------------+------+-
--------------+-------------+----------------+-----------
---+-----------+----------------+---------------------+----
-----------------+-----------+--------------------+-------
---+---------------+---------+
| courses        | InnoDB |      10 | Dynamic    |  112 |
146 |       16384 |               0 |            0
|         0 |           NULL | 2021-07-15 17:02:50 | 2021-
07-15 17:02:50 | NULL       | utf8mb4_0900_ai_ci |     NULL
|                |         |
```

```
| cust_history    | InnoDB |      10 | Dynamic    |    0 |
0 |       16384 |               0 |            0
|         0 |           NULL | 2021-07-15 17:02:50 | 2021-
07-15 17:02:50 | NULL       | utf8mb4_0900_ai_ci |     NULL
|                |          |
| customers       | InnoDB |      10 | Dynamic    |    1 |
16384 |       16384 |               0 |            0
|         0 |           NULL | 2021-07-15 17:02:50 | 2021-
07-15 17:02:50 | NULL       | utf8mb4_0900_ai_ci |     NULL
|                |          |

// details omitted for brevity
| students        | InnoDB |      10 | Dynamic    |    6 |
2730 |       16384 |               0 |            0
|         0 |           NULL | 2021-07-15 17:02:50 | 2021-
07-15 17:02:50 | NULL       | utf8mb4_0900_ai_ci |     NULL
|                |          |
| user            | InnoDB |      10 | Dynamic    |    4 |
4096 |       16384 |               0 |            0
|         0 |           NULL | 2021-07-15 17:02:50 | 2021-
07-15 17:02:50 | NULL       | utf8mb4_0900_ai_ci |     NULL
|                |          |
| weather         | InnoDB |      10 | Dynamic    |   11 |
1489 |       16384 |               0 |            0
|         0 |           NULL | 2021-07-15 17:02:50 | 2021-
07-15 17:02:50 | NULL       | utf8mb4_0900_ai_ci |     NULL
|                |          |
+------------------+--------+---------+------------+------+-
--------------+-------------+------------------+-----------
---+-----------+-----------------+----------------------+----
-----------------+------------+---------------------+-------
---+---------------+---------+
23 rows in set (0.002 sec)
```

EXPLAIN ANALYZE

MySQL 8.0.18 provides `EXPLAIN` that executes a SQL statement in order to generate `EXPLAIN` output. The output contains various details, some of which are listed here:

- Estimated execution cost
- Estimated number of returned rows
- Time to return first row
- Time (milliseconds) to return all rows (actual cost)
- Number of loops

Here is an example of a SQL statement that generates an execution plan:

```
MySQL [mytools]>
EXPLAIN SELECT 1;
SELECT *
FROM customers;
```

Launch the preceding SQL statement to see the following type of output:

```
+----+-------------+-------+------------+------+----------
-----+------+---------+------+------+----------+----------
------+
| id | select_type | table | partitions | type | possible_
keys | key  | key_len | ref  | rows | filtered | Extra        |
+----+-------------+-------+------------+------+----------
-----+------+---------+------+------+----------+----------
------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL
| NULL | NULL    | NULL | NULL |     NULL | No tables used |
+----+-------------+-------+------------+------+----------
-----+------+---------+------+------+----------+----------
------+
1 row in set, 1 warning (0.003 sec)

+---------+------------+-----------+--------------+--------
-+-------+----------+
| cust_id | first_name | last_name | home_address | city
| state | zip_code |
+---------+------------+-----------+--------------+--------
-+-------+----------+
|    1000 | John       | Smith     | 123 Main St  | Fremont
| CA    | 94123    |
+---------+------------+-----------+--------------+--------
-+-------+----------+
1 row in set (0.000 sec)
```

SCALING AN RDBMS

An RDBMS can be scaled in various ways, including the techniques in the following list, some of which are discussed in more detail later in this section:

- SQL tuning
- denormalization
- sharding
- federation
- master-slave replication
- master-master replication

You have already learned about denormalization in a previous section in this chapter, and the following subsections discuss SQL tuning, sharding, and federation.

What is SQL Tuning?

SQL tuning is a vast topic that is typically conducted by an experienced DBA, and it's primarily for improving application performance, which is often

related to SQL statements, as well as uncovering performance bottlenecks in applications. One technique involves simulating high loads on an application so that you can analyze the performance of the application. Another technique involves profiling SQL statements to track performance problems, using a tool such as "the slow query log."

In general, there are powerful tools available that determine which SQL queries require the most execution time, and those SQL queries can be displayed in decreasing order of execution time. Potential solutions involve creating new indexes, restructuring database tables, or sometimes denormalizing database tables.

Although SQL tuning is performed at the database level, sometimes you might need to perform additional tuning at the application level. Specifically, one scenario that arises with large databases (especially involving social media) requires splitting or "sharding" a table, as discussed in the next section.

What is Sharding?

Sharding is a horizontal scaling technique that logically partitions the rows in a table so that each partition can be stored and accessed independently of the other partitions. For example, suppose you have a user table that contains millions of rows and you need to support SQL operations such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. One way to improve the performance of such queries is to shard the user table by the letters in the alphabet: the first shard contains the people whose last name starts with "A," the second shard for the letter "B," and so forth.

The preceding technique can refined: since there are very few people whose last names start with "Q," "X," or "Z," we can combine those three shards into a single shard. Moreover, shards for the letters "M" and "S" are probably very large, so that they can be further sharded. For example, the shard for "M" can be split into the shards MA through MG, MH through MP, and MQ through MZ, and similarly for the shard for the letter "S." The actual splits are specific to the actual data in your user table.

An additional advantage to sharding is that each shard works independently of the other shards. Hence, if the shards are on different servers, then multiple shards can be operational even if some individual shards are unavailable (perhaps due to a power outage). By contrast, an unsharded table is all-or-nothing: if there is a power outage, then no table data is accessible.

RDBMS Support for Sharding

MySQL as well as Oracle and PostgresSQL (and others) do not support automatic sharding, which means that sharding must be implemented manually at the application layer. Consequently, sharding entails additional database design decisions, and you can perform an online search to for articles and blogposts that provide additional information.

If you are interested in delving further into the topic of database sharding, the following list contains several types of sharding techniques:

- Algorithmic Sharding
- Consistent Hash Sharding
- Linear Hash Sharding
- Range Sharding

Perform an online search to obtain more information about the sharding techniques in the preceding list.

What is Federation?

Federation (or functional partitioning) involves splitting a database in terms of its functionality instead of defining a monolithic database. For example, the four-table schema that you learned about in Chapter 1 involves separate tables for customers, purchase orders, line items, and item descriptions. In the event that any of these tables become extremely large, they can be placed in different locations. This technique can improve both read and write performance. By contrast, a smaller database can be placed in memory, which can improve so-called cache "hits." However, there are some disadvantages to federation:

- ineffective for schema involving very large tables
- additional application logic is involved
- joining data from two databases is complex
- hardware and additional complexity

DATABASE REPLICATION

Database replication refers to the process of copying data from one source to another, which effectively provides an online backup in the event of a data loss from the primary copy of the data. Replication provides a "fail over" capability. Otherwise, a primary system can become a single point of failure. Database replication provides the following advantages:

- improved read performance
- replicas are a complete copy of the primary database
- modifications to the primary copy are immediately propagated to replicas
- a replica can process incoming requests if the primary database is unavailable

A synchronous replication is typically slower yet has consistent data, whereas an asynchronous replication is performed in "detached" mode, which is faster but not always immediately consistent.

Incidentally, a common technique in high volume systems (such as social media applications) involves one server to handle "read" requests and another server to handle write, update, and delete requests. This technique works well because the ratio of "read" requests to all other request can be 100:1 or 200:1. Moreover, multiple servers can be allocated for "read" requests as well as multiple servers for the other types of requests.

There are also some disadvantages to database replication, such as higher cost and higher bandwidth requirements.

DISTRIBUTED DATABASES, SCALABILITY, AND THE CAP THEOREM

Now that you have a grasp of MySQL and some of its features, this section briefly discusses terminology such as distributed databases, scalable databases, and the CAP theorem. Although it's unlikely that you will be directly involved in these tasks (unless you are a DBA), it's worthwhile to have some understanding of these topics. However, if there is no pressing need, feel free to treat this section as optional.

In general, a service is *scalable* if its performance increases in proportion to the additional resources that are added to that service. A service can be database-related as well as software that is not directly coupled to a database.

A *distributed database* (DDB) are systems that focus on providing greater flexibility, reducing cost, and increasing performance. A DDB comprises a group of databases that are located in different sites, whereas a distributed database management system (DDBMS) manages a DDB. Users are unaware of the details (such as the location of the hardware and software) of the components of a DDB because it's irrelevant from the standpoint of performing their tasks.

Master-Slave Replication

The purpose of the master is to serve read operations and write operations. In addition, the master replicates (duplicates) write operations to one or more slaves because the slaves only perform read operations. In the event that the master is unavailable (for whatever reason), a system can still function, but only in read-only mode. For the system to resume write operations, the system must either promote a slave to the status of master or provision a new master in the system.

The CAP Theorem

CAP is an acronym for Consistency, Availability, and Partition Tolerance. The CAP theorem states that a distributed computer system support only two of the following three:

- *Consistency* means that every read receives the most recent write or an error.
- *Availability* means that every request receives a response, without guarantee that it contains the most recent version of the information.
- *Partition Tolerance* means that the system continues to operate despite arbitrary partitioning due to network failures.

Partition tolerance must be supported simply because networks are not reliable, and you need to decide between consistency and availability. Consistency is a good choice if your business needs require atomic reads and writes, whereas

availability is a good choice if a system must continue to function even though there are external errors.

Given the preceding points about the CAP theorem, the following statement will make sense: *MongoDB favors consistency over availability*.

What are Consistency Patterns?

Weak consistency means that read requests might not see the most recent write operation. A "best effort approach" can be adopted, which is true of systems such as memcached, which is in-memory key-value store. Weak consistency works well for various types of real time systems, such as VoIP, video chat, and multiplayer games.

Eventual consistency means that read requests will see the most recent write operation after a short delay, after data is replicated asynchronously. This type of consistency is applicable to email systems.

A third type of consistency is called *strong consistency*, in which read requests see data after a write operation because data is replicated synchronously. For example, an RDBMS provides strong consistency, which is also true of systems that support transactions.

MYSQL COMMAND LINE UTILITIES

To invoke a MySQL program from the command line (that is, from your shell or command prompt), enter the program name followed by any options or other arguments needed to instruct the program what you want it to do. The following commands show some sample program invocations.

The text string **`shell>`** represents the prompt for the command interpreter; it is not part of the text that you type at the command prompt. The particular prompt you see depends on your command interpreter. Typical prompts are `$` for `sh`, `ksh`, or `bash`; `%` for `csh` or `tsch`, and `C:\>` for the Windows `command.com` or `cmd.exe` command interpreters. Here are examples of several command line utilities:

```
shell> mysql --user=root test
shell> mysqladmin extended-status variables
shell> mysqlshow --help
shell> mysqldump -u root personnel
```

DATABASE BACKUPS, RESTORING DATA, AND UPGRADES

A DBA (database administrator) performs many important tasks, one of which involves automatically performing database backups (e.g., via `cron` jobs). Moreover, you (or someone else) need to know how to manually restore data from a backup in cases of lost or corrupted data.

A related topic is *disaster recovery*, which specifies the procedure for recovering a system in the event of a catastrophic failure and involves storing a complete set of backups in an off-site location.

A system administrator can help you recover deleted files and directories, whereas a DBA can help you manually restore database data from a backup in situations involving lost or corrupted data. However, any data or transactions that are performed after the most recent backup will not be available.

Database upgrades can be simple for minor releases of a database, but upgrading to a major release might involve changes to table definitions in a database schema. In general, a test environment is set up to fully test the upgrade, and if all goes well, the production system can be switched over to the new release.

Depending on the amount of data in an RDBMS, a database upgrade can be a lengthy process. In fact, some large enterprises perform an intensive testing process that can require an entire year before switching the production system to the latest database upgrade.

MYSQL AND JSON DATA

In previous chapters, you learned how to manage the contents of table containing simple data types, such as `CHAR`, `DATE`, `INT`, and `TEXT`. However, MySQL also supports JSON-based data. In fact, you can define a MySQL table with one or more attributes of type JSON, insert JSON-based data into such a table, and then query the table for its contents, as well as the values that are contained in the JSON data. Although MySQL supports JSON files, there is no index support for JSON-based data.

Listing 6.5 shows the content of `customers_json.sql` that is the counterpart to the MySQL `customers` table for our fictitious website, which performs the tasks described in the preceding paragraph, as noted in the comments in the code blocks.

LISTING 6.5: customers_json.sql

```
use mytools;

DROP TABLE IF EXISTS customer_json;

-- a table with a JSON attribute:
CREATE TABLE customer_json (
  id int auto_increment primary key,
  customer json
);

-- insert JSON-based data into the table:
INSERT INTO customer_json(customer)
VALUES (
   '{ "cust_id": "1000", "first_name": "John", "last_name":
"Smith", "address": "123 Main Street", "city": "Fremont",
"state": "CA", "zip_code": "94123"}'
),
(
```

```
   '{ "cust_id": "2000", "first_name": "Jane", "last_name":
"Jones", "address": "456 Front Street", "city": "Fremont",
"state": "CA", "zip_code": "95015"}'
);

-- display the values of the first_name and last_name attributes:
SELECT id, customer->'$.first_name', customer->'$.last_name'
FROM customer_json;

-- the JSON_ARRAY() function creates arrays:
SELECT JSON_ARRAY(1000, "Deep", "Dish", "Pizza");

-- the JSON_OBJECT() function creates objects:
SELECT JSON_OBJECT(1000, "Deep", "Dish", "Pizza");

-- the JSON_QUOTE() function quotes a string as a JSON value:
SELECT JSON_QUOTE('[1000, "Deep", "Dish", "Pizza"]');
```

Listing 6.5 starts by defining the table `customer_json` with a `customer` attribute of type `JSON`, followed by inserting two JSON-based strings into this table. Next, a SQL statement retrieves the values the `first_name` and `last_name` attributes, followed by three SQL statements that illustrate how to use the `JSON_ARRAY()`, `JSON_OBJECT()`, and `JSON_QUOTE()` functions. Launch the code in Listing 6.5 to see the following output:

```
-- the JSON_QUOTE() function quotes a string as a JSON
value:
Database changed
Query OK, 0 rows affected (0.039 sec)
Query OK, 0 rows affected (0.021 sec)
Query OK, 2 rows affected (0.005 sec)
Records: 2  Duplicates: 0  Warnings: 0

+----+--------------------------+-------------------------+
| id | customer->'$.first_name'  | customer->'$.last_name' |
+----+--------------------------+-------------------------+
|  1 | "John"                   | "Smith"                 |
|  2 | "Jane"                   | "Jones"                 |
+----+--------------------------+-------------------------+
2 rows in set (0.093 sec)

+-------------------------------------------+
| JSON_ARRAY(1000, "Deep", "Dish", "Pizza") |
+-------------------------------------------+
| [1000, "Deep", "Dish", "Pizza"]           |
+-------------------------------------------+
1 row in set (0.000 sec)

+--------------------------------------------+
| JSON_OBJECT(1000, "Deep", "Dish", "Pizza") |
+--------------------------------------------+
| {"1000": "Deep", "Dish": "Pizza"}          |
+--------------------------------------------+
1 row in set (0.001 sec)
```

```
+-------------------------------------------------+
| JSON_QUOTE('[1000, "Deep", "Dish", "Pizza"]') |
+-------------------------------------------------+
| "[1000, \"Deep\", \"Dish\", \"Pizza\"]"         |
+-------------------------------------------------+
1 row in set (0.000 sec)
```

In Listing 6.5, several code blocks are preceded by self-explanatory comment statements that explain the purpose of the code. The only significant difference from previous code samples is the different syntax for working with JSON-specific data. If you want to learn more, navigate to the online documentation for more information regarding JSON-based data in MySQL.

DATA CLEANING IN SQL

This section contains several subsections that perform data cleaning tasks in SQL. Although this section could have been placed in an earlier chapter instead of a "miscellaneous" chapter, there is also a subsequent section that involves cleaning data from the command line in order to perform tasks that are not possible in Pandas or another similar type of tool. Hence, it was deemed better to keep these two sections together and to place them in this chapter.

This section illustrates how to perform the following data cleaning tasks that affect an attribute of a database table:

- replace NULL with 0
- replace NULL with the average value
- replace multiple values into a single value
- handle data type mismatch
- convert a string date to a date format

Replace NULL with 0

You can perform this task with either of the following SQL statements:

```
SELECT ISNULL(column_name, 0 ) FROM table_name
OR
SELECT COALESCE(column_name, 0 ) FROM table_name
```

Replace NULL Values with Average Value

This task involves two steps: first find the average of the non-`NULL` values of a column in a database table, and then update the `NULL` values in that column with the value that you found in the first step.

Listing 6.6 shows the content of `replace_null_values.sql` that performs this pair of steps.

LISTING 6.6: replace_null_values.sql

```
USE mytools;
DROP TABLE IF EXISTS temperatures;
CREATE TABLE temperatures (temper INT, city CHAR(20));
```

```
INSERT INTO temperatures VALUES(78,'sf');
INSERT INTO temperatures VALUES(NULL,'sf');
INSERT INTO temperatures VALUES(42,NULL);
INSERT INTO temperatures VALUES(NULL,'ny');
SELECT * FROM temperatures;

SELECT @avg1 := AVG(temper) FROM temperatures;
update temperatures
set temper = @avg1
where ISNULL(temper);
SELECT * FROM temperatures;

-- initialize city1 with the most frequent city value:
SELECT @city1 := (SELECT city FROM temperatures GROUP BY
city ORDER BY COUNT(*) DESC LIMIT 1);

-- update NULL city values with the value of city1:
update temperatures
set city = @city1
where ISNULL(city);
SELECT * FROM temperatures;
```

Listing 6.6 creates and populates the table `temperatures` with several rows, and then initializes the variable `avg1` with the average temperature in the `temper` attribute of the `temperatures` table. Launch the code in Listing 6.6 to see the following output:

```
+--------+------+
| temper | city |
+--------+------+
|     78 | sf   |
|   NULL | sf   |
|     42 | NULL |
|   NULL | ny   |
+--------+------+
4 rows in set (0.000 sec)

+----------------------+
| @avg1 := AVG(temper) |
+----------------------+
|         60.000000000 |
+----------------------+
1 row in set, 1 warning (0.000 sec)

Query OK, 2 rows affected (0.001 sec)
Rows matched: 2  Changed: 2  Warnings: 0

+--------+------+
| temper | city |
+--------+------+
|     78 | sf   |
|     60 | sf   |
|     42 | NULL |
|     60 | ny   |
+--------+------+
4 rows in set (0.000 sec)
```

```
+----------------------------------------------------------
----------+
| @city1 := (SELECT city FROM temperatures GROUP BY city
ORDER BY COUNT(*) DESC LIMIT 1) |
+----------------------------------------------------------
----------+
| sf
|
+----------------------------------------------------------
----------+
1 row in set, 1 warning (0.000 sec)

Query OK, 1 row affected (0.000 sec)
Rows matched: 1  Changed: 1  Warnings: 0

+--------+------+
| temper | city |
+--------+------+
|     78 | sf   |
|     60 | sf   |
|     42 | sf   |
|     60 | ny   |
+--------+------+
4 rows in set (0.000 sec)
```

Replace Multiple Values with a Single Value

An example of coalescing multiple values in an attribute involves replacing multiple strings for the state of New York (such as `new_york`, `NewYork`, and `New_York`) with `NY`. Listing 6.7 shows the content of `reduce_values.sql` that performs this pair of steps.

LISTING 6.7: reduce_values.sql

```
use mytools;
DROP TABLE IF EXISTS mytable;
CREATE TABLE mytable (str_date CHAR(15), state CHAR(20),
reply CHAR(10));

INSERT INTO mytable VALUES('20210915','New York','Yes');
INSERT INTO mytable VALUES('20211016','New York','no');
INSERT INTO mytable VALUES('20220117','Illinois','yes');
INSERT INTO mytable VALUES('20220218','New York','No');
SELECT * FROM mytable;

-- replace yes, Yes, y, Ys with Y:
update mytable
set reply = 'Y'
where upper(substr(reply,1,1)) = 'Y';

-- replace all other values with
update mytable
set reply = 'N' where substr(reply,1,1) != 'Y';
SELECT * FROM mytable;
```

Listing 6.7 creates and populates the table `mytable`, and then replaces the variants of the word "yes" with the letter `Y` in the reply attribute. The final portion of Listing 6.7 replaces any string that does *not* start with the letter `Y` with the letter `N`. Launch the code in Listing 6.7 to see the following output:

```
+----------+----------+-------+
| str_date | state    | reply |
+----------+----------+-------+
| 20210915 | New York | Yes   |
| 20211016 | New York | no    |
| 20220117 | Illinois | yes   |
| 20220218 | New York | No    |
+----------+----------+-------+
4 rows in set (0.000 sec)

Query OK, 2 rows affected (0.001 sec)
Rows matched: 2  Changed: 2  Warnings: 0

+----------+----------+-------+
| str_date | state    | reply |
+----------+----------+-------+
| 20210915 | New York | Y     |
| 20211016 | New York | N     |
| 20220117 | Illinois | Y     |
| 20220218 | New York | N     |
+----------+----------+-------+
4 rows in set (0.001 sec)
```

Handle Mismatched Attribute Values

This task involves two steps: first find the average of the non-`NULL` values of a column in a database table, and then update the `NULL` values in that column with the value that you found in the first step.

Listing 6.8 shows the content of `type_mismatch.sql` that performs this pair of steps.

LISTING 6.8: type_mismatch.sql

```
USE mytools;
DROP TABLE IF EXISTS emp_details;
CREATE TABLE emp_details (emp_id CHAR(15), city CHAR(20),
state CHAR(20));

INSERT INTO emp_details VALUES('1000','Chicago','Illinois');
INSERT INTO emp_details VALUES('2000','Seattle','Washington');
INSERT INTO emp_details VALUES('3000','Santa Cruz','California');
INSERT INTO emp_details VALUES('4000','Boston','Massachusetts');
SELECT * FROM emp_details;

select emp.emp_id, emp.title, det.city, det.state
from employees emp join emp_details det
WHERE emp.emp_id = det.emp_id;
```

```
--required for earlier versions of MySQL:
--WHERE emp.emp_id = cast(det.emp_id as INT);
```

Listing 6.8 creates and populates the table `emp_details`, followed by a SQL `JOIN` statement involving the tables `emp` and `emp_details`. Although the `emp_id` table is defined as an `INT` type and a `CHAR` type, respectively, in the tables `emp` and `emp_details`, the code works as desired. However, in earlier versions of MySQL, you need to use the built-in `CAST()` function to convert a `CHAR` value to an `INT` value (or vice versa), as shown in the commented out code snippet:

```
--WHERE emp.emp_id = cast(det.emp_id as INT);
```

Now launch the code in Listing 6.8 and you will see the following output:

```
+--------+------------+---------------+
| emp_id | city       | state         |
+--------+------------+---------------+
| 1000   | Chicago    | Illinois      |
| 2000   | Seattle    | Washington    |
| 3000   | Santa Cruz | California    |
| 4000   | Boston     | Massachusetts |
+--------+------------+---------------+
4 rows in set (0.000 sec)
+--------+--------------------+------------+---------------+
| emp_id | title              | city       | state         |
+--------+--------------------+------------+---------------+
|   1000 | Developer          | Chicago    | Illinois      |
|   2000 | Project Lead       | Seattle    | Washington    |
|   3000 | Dev Manager        | Santa Cruz | California    |
|   4000 | Senior Dev Manager | Boston     | Massachusetts |
+--------+--------------------+------------+---------------+
4 rows in set (0.002 sec)
```

Convert Strings to Date Values

Listing 6.9 shows the content of `str_to_date.sql` that illustrates how to populate a date attribute with date values that are determined from another string-based attribute that contains strings for dates.

LISTING 6.9: str_to_date.sql

```
use mytools;
DROP TABLE IF EXISTS mytable;
CREATE TABLE mytable (str_date CHAR(15), state CHAR(20),
reply CHAR(10));

INSERT INTO mytable VALUES('20210915','New York','Yes');
INSERT INTO mytable VALUES('20211016','New York','no'););
INSERT INTO mytable VALUES('20220117','Illinois','yes'););
INSERT INTO mytable VALUES('20220218','New York','No'););

SELECT * FROM mytable;
```

```
-- 1) insert date-based feature:
ALTER TABLE mytable
ADD COLUMN (real_date DATE);
SELECT * FROM mytable;

-- 2) populate real_date from str_date:
UPDATE mytable t1
        INNER JOIN mytable t2
             ON t1.str_date = t2.str_date
SET t1.real_date = DATE(t2.str_date);
SELECT * FROM mytable;

-- 3) Remove unwanted features:
ALTER TABLE mytable
DROP COLUMN str_date;
SELECT * FROM mytable;
```

Listing 6.9 creates and populates the table `mytable` and displays the contents of this table. The remainder of Listing 6.9 consists of three SQL statements, each of which starts with a comment statement that explains its purpose.

The first SQL statement inserts a new column `real_date` of type `DATE`. The second SQL statement populates the `real_date` column with the values in the `str_date` column that have been converted to a date value via the `DATE()` function. The third SQL statement is optional: it drops the `str_date` column if you wish to do so. Launch the code in Listing 6.9 to see the following output:

```
+----------+----------+-------+
| str_date | state    | reply |
+----------+----------+-------+
| 20210915 | New York | Yes   |
| 20211016 | New York | no    |
| 20220117 | Illinois | yes   |
| 20220218 | New York | No    |
+----------+----------+-------+
4 rows in set (0.000 sec)

Query OK, 0 rows affected (0.007 sec)
Records: 0  Duplicates: 0  Warnings: 0

+----------+----------+-------+-----------+
| str_date | state    | reply | real_date |
+----------+----------+-------+-----------+
| 20210915 | New York | Yes   | NULL      |
| 20211016 | New York | no    | NULL      |
| 20220117 | Illinois | yes   | NULL      |
| 20220218 | New York | No    | NULL      |
+----------+----------+-------+-----------+
4 rows in set (0.002 sec)

Query OK, 4 rows affected (0.002 sec)
Rows matched: 4  Changed: 4  Warnings: 0
```

```
+----------+----------+-------+------------+
| str_date | state    | reply | real_date  |
+----------+----------+-------+------------+
| 20210915 | New York | Yes   | 2021-09-15 |
| 20211016 | New York | no    | 2021-10-16 |
| 20220117 | Illinois | yes   | 2022-01-17 |
| 20220218 | New York | No    | 2022-02-18 |
+----------+----------+-------+------------+
4 rows in set (0.000 sec)

Query OK, 0 rows affected (0.018 sec)
Records: 0  Duplicates: 0  Warnings: 0

+----------+-------+------------+
| state    | reply | real_date  |
+----------+-------+------------+
| New York | Yes   | 2021-09-15 |
| New York | no    | 2021-10-16 |
| Illinois | yes   | 2022-01-17 |
| New York | No    | 2022-02-18 |
+----------+-------+------------+
4 rows in set (0.000 sec)
```

DATA CLEANING FROM THE COMMAND LINE (OPTIONAL)

This section is marked “optional” because the solutions to tasks involve an understanding of some Unix-based utilities. Although this book does not contain details about those utilities, you can find online tutorials with examples regarding these utilities.

This section contains several subsections that perform data cleaning tasks that involve the command line utilities `sed` and `awk`:

- replace multiple delimiters with a single delimiter (`sed`)
- restructure a dataset so all rows have the same column count (`awk`)

Keep in mind the following point about these examples: they must be performed from the command line before they can be processed in a Pandas data frame.

Working with the sed Utility

This section contains an example of how to use the `sed` command line utility to replace different delimiters with a single delimiter for the fields in a text file. You can use the same code for other file formats, such as CSV files and TSV files.

This section does not provide any details about `sed` beyond the code sample in this section. However, after you read the code, you will understand how to adapt that code snippet to your own requirements (i.e., how to specify different delimiters).

Listing 6.10 shows the content of `delimiter1.txt` and Listing 6.11 shows the content of `delimiter1.sh` that replaces all delimiters with a comma (",").

LISTING 6.10: delimiter1.txt

```
1000|Jane:Edwards^Sales
2000|Tom:Smith^Development
3000|Dave:Del Ray^Marketing
```

LISTING 6.11: delimiter1.sh

```
cat delimiter1.txt | sed -e 's/:/,/' -e 's/|/,/' -e 's/\^/,/'
```

Listing 6.11 starts with the `cat` command line utility, which sends the contents of the file `delimiter1.txt` "standard output," which is the screen (by default). However, in this example, the output of this command becomes the input to the sed command because of the pipe ("|") symbol.

The `sed` command consists of three parts, all of which are connected by the "-e" switch. You can think of "-e" as indicating "there is more processing to be done" by the `sed` command. In this example, there are three occurrences of "-e," which means that the `sed` command will be invoked three times.

The first code snippet is `'s/:/,/'`, which translates into "replace each semi-colon with a comma." The result of this operation is passed to the next code snippet, which is `'s/|/,/'`. This code snippet translates into "replace each pipe symbol with a comma." The result of this operation is passed to the next code snippet, which is `'s/\^/,/'`. This code snippet translates into "replace each caret symbol ("^") with a comma." The result of this operation is sent to standard output, which can be redirected to another text file. Launch the code in Listing 5.27 to see the following output:

```
1000,Jane,Edwards,Sales
2000,Tom,Smith,Development
3000,Dave,Del Ray,Marketing
```

Here are three comments to keep in mind. First, the snippet contains a backslash because the caret symbol ("^") is a meta character, so we need to "escape" this character. The same is true for other meta characters (such as "$" and ".").

Second, you can easily extend the `sed` command for each new delimiter that you encounter as a field separator in a text file: simply follow the pattern that you see in Listing 5.27 for each new delimiter.

Third, redirect the output of `delimiter1.sh` to the text file `delimiter2.txt` by launching the following command:

```
./delimiter1.sh > delimiter2.txt
```

If an error occurs in the preceding code snippet, make sure that `delimiter1.sh` is executable by invoking the following command:

```
chmod 755 delimiter1.sh
```

This concludes the example involving the `sed` command line utility, which is a very powerful utility for processing text files. Check online for articles and blog posts if you want to learn more about the `sed` utility.

Working with the awk Utility

The `awk` command line utility is a self-contained programming language, with a truly impressive capability for processing text files. However, this section does not provide details about `awk` beyond the code sample. If you're interested, there are plenty of online articles that provide in-depth explanations regarding the `awk` utility.

Listing 6.12 shows the content `FixedFieldCount1.sh` that illustrates how to use the `awk` utility to split a string into rows that contain three strings.

LISTING 6.12: FixedFieldCount1.sh

```
echo "=> pairs of letters:"
echo "aa bb cc dd ee ff gg hh"
echo

echo "=> split on multiple lines:"
echo "aa bb cc dd ee ff gg hh"| awk '
BEGIN { colCount = 3 }
{
  for(i=1; i<=NF; i++) {
     printf("%s ", $i)
     if(i % colCount == 0) { print "" }
  }
  print ""
}
'
```

Listing 6.12 shows the contents of a string, and then provides this string as input to the `awk` command. The main body of Listing 6.12 is a loop that iterates from 1 to `NF`, where `NF` is the number of fields in the input line, which in this example equals 8. The value of each field is represented by `$i`: `$1` is the first field, `$2` is the second field, and so forth. Note that `$0` is the contents of the *entire* input line (which is used in a subsequent code sample).

Next, if the value of `i` (which is the field *position*, not the contents of the field) is a multiple of 3, then the code prints a linefeed. Launch the code in Listing 6.12 to see the following output:

```
=> pairs of letters:
aa bb cc dd ee ff gg hh

=> split on multiple lines:
aa bb cc
```

```
dd ee ff
gg hh
```

Listing 6.13 shows the content of `employees.txt` and Listing 6.14 shows the content of `FixedFieldCount2.sh` that illustrates how to use the `awk` utility in order to ensure that all the rows have the same number of columns.

LISTING 6.13: employees.txt

```
jane:jones:SF:
john:smith:LA:
dave:smith:NY:
sara:white:CHI:
>>>none:none:none<<<:
jane:jones:SF:john:
smith:LA:
dave:smith:NY:sara:white:
CHI:
```

LISTING 6.14: FixedFieldCount2.sh

```
cat employees.txt | awk -F":" '{printf("%s", $0)}' | awk -F':' '
BEGIN { colCount = 3 }
{
  for(i=1; i<=NF; i++) {
     printf("%s#", $i)
     if(i % colCount == 0) { print "" }
  }
}
'
```

Notice that the code in Listing 6.14 is almost identical to the code in Listing 6.13: the code snippet that is shown in bold removes the `\n` character from its input that consists of the contents of `employees.txt`. The reason this happens is because of this code snippet:

```
printf("%s", $0)
```

If you want to retain the `\n` character after each input line, then replace the preceding code snippet with this snippet:

```
printf("%s\n", $0)
```

We have now reduced the task in Listing 6.14 to the same task as Listing 6.13, which is why the solution contains the same `awk-base` *code block.*

Launch the code in Listing 6.14 to see the following output:

```
1000,Jane,Edwards,Sales
jane#jones#SF#
john#smith#LA#
dave#smith#NY#
sara#white#CHI#
>>>none#none#none<<<#
```

```
jane#jones#SF#
john#smith#LA#
dave#smith#NY#
sara#white#CHI#
```

NEXT STEPS

Although the direction that you pursue after completing this book depends entirely on your list of objectives, you might be interested in some of the following topics:

- Pivot tables
- B trees
- B+ trees
- Hash indexes

If you have worked extensively with Excel spreadsheets, you are probably well acquainted with pivot tables. Although MySQL 8 does not provide a `PIVOT` function for pivot tables, you can implement this functionality with the `CASE` statement.

Alternatively, it might be simpler to use Excel to perform pivot-related functionality and then import the results into a MySQL table. Another possibility is to use a tool such as dbForge Studio for MySQL (free trial version available) or search for open source tools that provide support for pivot tables.

If you're interested in the implementation of indexes, then perform an online search for articles that discuss B-trees, B+ trees, and hash indexes.

Despite the rich functionality available in MySQL, you might also need to consider a different RDBMS if MySQL does not provide a critical feature for your needs.

In closing, it's worthwhile to perform online searches for tools that can simplify your SQL-related tasks, as well as documentation or blog posts that explain the implementation of more complex and lower-level tasks.

SUMMARY

This chapter started with an overview of managing database users: how to create users and how to drop users. Next, you learned about roles in MySQL, along with creating roles, granting privileges, revoking roles, and dropping roles.

Then you got a more detailed description of normalization and an introduction to entity-relationship modeling, which involves diagrams that display entities (tables) and the relationships between tables.

Next, you learned about schemas and how to generate schemas in the MySQL, as well as the concept of a transaction. In addition, you learned about aspects of database optimization, performance tuning considerations, and SQL

query optimization. You were introduced to database optimization and performance tuning.

You also became familiar with ways of scaling an RDBMS, such as sharding and federation. Then you learned an assortment of topics such as stored procedures, stored functions, and triggers.

Finally, you were exposed to an assortment of miscellaneous topics, including distributed databases, the CAP theorem, MySQL command line utilities, database backups and upgrades, and character sets in MySQL.

APPENDIX

INTRODUCTION TO PROBABILITY AND STATISTICS

This appendix introduces you to concepts in probability as well as an assortment of statistical terms and algorithms.

The first section of this appendix starts with a discussion of probability, how to calculate the expected value of a set of numbers (with associated probabilities), the concept of a random variable (discrete and continuous), and a short list of some well-known probability distributions.

The second section of this appendix introduces basic statistical concepts, such as mean, median, mode, variance, and standard deviation, along with simple examples that illustrate how to calculate these terms. You will also learn about the terms RSS, TSS, R^2, and F1 score.

The third section of this appendix introduces Gini Impurity, Entropy, Perplexity, Cross-Entropy, and KL Divergence. You will also learn about skewness and kurtosis.

The fourth section explains covariance and correlation matrices and how to calculate eigenvalues and eigenvectors.

The fifth section explains PCA (Principal Component Analysis), which is a well-known dimensionality reduction technique. The final section introduces you to Bayes' Theorem.

WHAT IS A PROBABILITY?

If you have ever performed a science experiment in one of your classes, you might remember that measurements have some uncertainty. In general, we assume that there is a correct value, and we endeavor to find the best estimate of that value.

When we work with an event that can have multiple outcomes, we try to define the probability of an outcome as the chance that it will occur, which is calculated as follows:

p(outcome) = (# of times outcome occurs)/(total number of outcomes)

For example, in the case of a single balanced coin, the probability of tossing a head H equals the probability of tossing a tail T:

p(H) = 1/2 = p(T)

Hence, the *set* of probabilities associated with the outcomes {H, T} is shown in the set P:

P = {1/2, 1/2}

Some experiments involve replacement while others involve non-replacement. For example, suppose that an urn contains 10 red balls and 10 green balls. What is the probability that a randomly selected ball is red? The answer is 10/(10+10) = 1/2. What is the probability that the second ball is also red?

The answer to the preceding question involves two scenarios with two different answers. If each ball is selected *with replacement*, that means each selected ball is returned to the urn, which in turn means that the urn *always* contains 10 red balls and 10 green balls. In this case, the probability of selecting a red ball is always the same, regardless of the number of times that a ball is selected from the urn. Hence, the answer to the preceding question is 1/2 * 1/2 = 1/4. In fact, the probability of any event is *independent* of all previous events.

On the other hand, if balls are selected *without replacement*, then the probability is 10/20 * 9/19. Card games are also examples of selecting cards without replacement.

One other concept is called *conditional probability*, which refers to the likelihood of the occurrence of event E1 given that event E2 has occurred. A simple example is the following statement:

"If it rains (E2), then I will carry an umbrella (E1)."

Calculating the Expected Value

Consider the following scenario involving a well-balanced coin: whenever a head appears, you earn $1 and whenever a tail appears, you earn $1 dollar. If you toss the coin 100 times, how much money do you expect to earn? Since you will earn $1 regardless of the outcome, the expected value (in fact, the guaranteed value) is 100.

Now consider this scenario: whenever a head appears, you earn $1 and whenever a tail appears, you earn 0 dollars. If you toss the coin 100 times, how much money do you expect to earn? You probably determined the value 50 (which is the correct answer) by making a quick mental calculation. The more formal derivation of the value of E (the expected earning) is here:

E = 100 *[1 * 0.5 + 0 * 0.5] = 100 * 0.5 = 50

The quantity **1** * 0.5 + **0** * 0.5 is the amount of money you expected to earn during each coin toss: half the time you earn $1 and half the time you earn 0 dollars. Multiply this value by 100 to compute the expected earnings after 100 coin tosses. Note that you might never earn $50: the actual amount that you earn can be *any* integer between 1 and 100 inclusive.

As another example, suppose that you earn $3 whenever a head appears, and you *lose* $1.50 dollars whenever a tail appears. Then the expected earning E after 100 coin tosses is shown here:

E = 100 *[3 * 0.5 – 1.5 * 0.5] = 100 * 1.5 = 150

We can generalize the preceding calculations as follows. Let P = {p1,…,pn} be a probability distribution, which means that the values in P are non-negative and their sum equals 1. In addition, let R = {R1,…,Rn} be a set of rewards, where reward Ri is received with probability pi. Then the expected value E after N trials is shown here:

E = N * [SUM pi*Ri]

In the case of a single balanced die, we have the following probabilities:

p(1) = 1/6
p(2) = 1/6
p(3) = 1/6
p(4) = 1/6
p(5) = 1/6
p(6) = 1/6
P = { 1/6, 1/6, 1/6, 1/6, 1/6, 1/6}

Next, we need to know the values in the set R before we can calculate the expected value E. As a simple example, suppose that the earnings are {1, 1, 1, 1, 1, 1} when the values 1, 2, 3, 4, 5, and 6, respectively, appear when tossing the single die. Then after 100 trials, our expected earnings are calculated as follows (and rounded to three decimal places):

E = 100 * [1 + 1 + 1 + 1 + 1 + 1]/6 = 100 * 1/6 = 16.667

As another example, suppose that the earnings are {3, 0, –1, 2, 4, –1} when the values 1, 2, 3, 4, 5, and 6, respectively, appear when tossing the single die. Then after 100 trials, our expected earnings are calculated as follows:

E = 100 * [3 + 0 + –1 + 2 + 4 + –1]/6 = 100 * 3/6 = 50

In the case of two balanced dice, we have the following probabilities of rolling 2, 3, … , or 12:

p(2) = 1/36
p(3) = 2/36
...

p(12) = 1/36
P = {1/36, 2/36, 3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36}

Construct a set with values for rewards for each of the 11 possible outcomes and then calculate the expected value.

RANDOM VARIABLES

A *random variable* is a variable that can have multiple values, and where each value has an associated probability of occurrence. For example, if we let X be a random variable whose values are the outcomes of tossing a well-balanced die, then the values of X are the numbers in the set {1, 2, 3, 4, 5, 6}. Each of those values can occur with equal probability (which is 1/6).

In the case of two well-balanced dice, let X be a random variable whose values can be any of the numbers in the set {2, 3, 4, . . . , 12}. Then the associated probabilities for the different values for X are listed in the previous section.

Discrete versus Continuous Random Variables

The preceding section contains examples of *discrete* random variables because the list of possible values is either finite or countably infinite (such as the set of integers). As an aside, the set of rational numbers and the set of algebraic numbers are also countably infinite, but the set of non-algebraic irrational numbers and the set of real numbers are both uncountably infinite (proofs are available online). As pointed out earlier, the associated set of probabilities must form a probability distribution, which means that the probability values are non-negative and their sum equals 1.

A *continuous* random variable whose values can be *any* number in an interval, which can be an uncountably infinite number of values. For example, the amount of time required to perform a task is represented by a continuous random variable.

A continuous random variable also has a probability distribution that is represented as a continuous function. The constraint for such a variable is that the area under the curve (which is sometimes calculated via a mathematical integral) equals 1.

Well-Known Probability Distributions

There are many probability distributions, and some of the well-known probability distributions are listed here:

- Gaussian distribution
- Poisson distribution
- Chi-squared distribution
- Binomial distribution

The Gaussian distribution is named after Karl F. Gauss, and it is sometimes called the normal distribution or the Bell curve. The Gaussian distribution is

symmetric: the shape of the curve on the left of the mean is identical to the shape of the curve on the right side of the mean. As an example, the distribution of IQ scores follows a curve that is similar to a Gaussian distribution.

The frequency of traffic at a given point in a road follows a Poisson distribution (which is not symmetric). Interestingly, if you count the number of people who go to a public pool based on five-degree (Fahrenheit) increments of the temperature, followed by five-degree decrements in temperature, that set of numbers follows a Poisson distribution.

Perform an Internet search for each of the bullet items in the preceding list and you will find numerous articles that contain images and technical details about these (and other) probability distributions.

This concludes the brief introduction to probability, and the next section delves into the concepts of mean, median, mode, and standard deviation.

FUNDAMENTAL CONCEPTS IN STATISTICS

This section contains several subsections that discuss the mean, median, mode, variance, and standard deviation. Feel free to skim (or skip) this section if you are already familiar with these concepts. As a start point, let's suppose that we have a set of numbers X ={x1, ..., xn} that can be positive, negative, integer-valued or decimal values.

The Mean

The *mean* of the numbers in the set X is the average of the values. For example, if the set X consists of {–10, 35, 75, 100}, then the mean equals (–10 + 35 + 75 + 100)/4 = 50. If the set X consists of {2, 2, 2, 2}, then the mean equals (2+2+2+2)/4 = 2. As you can see, the mean value is not necessarily one of the values in the set.

The mean is sensitive to outliers. For example, the mean of the set of numbers {1, 2, 3, 4} is 2.5, whereas the mean of the set of number {1, 2, 3, 4, 1000} is 202. Since the formulas for the variance and standard deviation involve the mean of a set of numbers, both of these terms are also more sensitive to outliers.

The Median

The *median* of the numbers (sorted in increasing or decreasing order) in the set X is the middle value in the set of values, which means that half the numbers in the set are less than the median and half the numbers in the set are greater than the median. For example, if the set X consists of {–10, 35, 75, 100}, then the median equals 55 because 55 is the average of the two numbers 35 and 75. As you can see, half the numbers are less than 55 and half the numbers are greater than 55. If the set X consists of {2, 2, 2, 2}, then the median equals 2.

By contrast, the median is much less sensitive to outliers than the mean. For example, the median of the set of numbers {1, 2, 3, 4} is 2.5, and the median of the set of numbers {1, 2, 3, 4, 1000} is 3.

The Mode

The *mode* of the numbers (sorted in increasing or decreasing order) in the set X is the most frequently occurring value, which means that there can be more than one such value. If the set X consists of {2, 2, 2, 2}, then the mode equals 2.

If X is the set of numbers {2, 4, 5, 5, 6, 8}, then the number 5 occurs twice and the other numbers occur only once, so the mode equals 5.

If X is the set of numbers {2, 2, 4, 5, 5, 6, 8}, then the numbers 2 and 5 occur twice and the other numbers occur only once, so the mode equals 2 and 5. A set that has two modes is called *bimodal*, and a set that has more than two modes is called *multi-modal*.

One other scenario involves sets that have numbers with the same frequency and they are all different. In this case, the mode does not provide meaningful information, and one alternative is to partition the numbers into subsets and then select the largest subset. For example, if set X has the values {1, 2, 15, 16, 17, 25, 35, 50}, we can partition the set into subsets whose elements are in range that are multiples of ten, which results in the subsets {1, 2}, {15, 16, 17}, {25}, {35}, and {50}. The largest subset is {15, 16, 17}, so we could select the number 16 as the mode.

As another example, if set X has the values {–10, 35, 75, 100}, then partitioning this set does not provide any additional information, so it's probably better to work with either the mean or the median.

The Variance and Standard Deviation

The *variance* is the sum of the squares of the difference between the numbers in X and the mean *mu* of the set X, divided by the number of values in X, as shown here:

variance = [SUM (xi – mu)**2] / n

For example, if the set X consists of {–10, 35, 75, 100}, then the mean equals (–10 + 35 + 75 + 100)/4 = 50, and the variance is computed as follows:

```
variance = [(–10–50)**2 + (35–50)**2 + (75–50)**2 + (100–50)**2]/4
         = [60**2 + 15**2 + 25**2 + 50**2]/4
         = [3600 + 225 + 625 + 2500]/4
         = 6950/4 = 1,737
```

The standard deviation *std* is the square root of the variance:

std = sqrt(1737) = 41.677

If the set X consists of {2, 2, 2, 2}, then the mean equals (2+2+2+2)/4 = 2, and the variance is computed as follows:

```
variance = [(2–2)**2 + (2–2)**2 + (2–2)**2 + (2–2)**2]/4
         = [0**2 + 0**2 + 0**2 + 0**2]/4
         = 0
```

The preceding result is intuitive: since the numbers all equal 2, they do not “vary” at all, so the variance equals 0. In addition, the standard deviation *std* is the square root of the variance:

std = sqrt(0) = 0

Population, Sample, and Population Variance

The *population* specifically refers to the entire set of entities in a given group, such as the population of a country, the people over 65 in the USA, or the number of first year students in a university.

However, in many cases, statistical quantities are calculated on samples instead of an entire population. Thus, a sample is (a much smaller) subset of the given population. See the Central Limit Theorem regarding the distribution of the mean of a set of samples of a population (which need not be a population with a Gaussian distribution).

If you want to learn about techniques for sampling data, here is a list of three different techniques that you can investigate:

- Stratified sampling
- Cluster sampling
- Quota sampling

The population variance is calculated by multiplying the sample variance by n/(n−1), as shown here:

population variance = [n/(n−1)]*variance

Chebyshev’s Inequality

Chebyshev’s inequality provides a simple way to determine the minimum percentage of data that lies within k standard deviations. Specifically, this inequality states that for any positive integer k greater than 1, the amount of data in a sample that lies within k standard deviations is at least 1 – 1/k**2. For example, if k = 2, then at least 1 – 1/2**2 = 3/4 of the data must lie within 2 standard deviations.

The interesting part of this inequality is that it has been mathematically proven to be true; i.e., it’s not an empirical or heuristic-based result. An extensive description regarding Chebyshev’s inequality (including some advanced mathematical explanations) is available online:

https://en.wikipedia.org/wiki/Chebyshev%27s_inequality

What is a p-value?

The null hypothesis states that there is no correlation between a dependent variable (such as y) and an independent variable (such as x). The p-value is used to reject the null hypothesis if the p-value is small enough (< 0.005), which indicates a higher significance. The threshold value for p is typically 1% or 5%.

There is no simple formula for calculating p-values, which are values that are always between 0 and 1. In fact, p-values are statistical quantities to evaluate the null hypothesis, and they are calculated by means of p-value tables or via spreadsheet/statistical software.

THE MOMENTS OF A FUNCTION (OPTIONAL)

The previous sections describe several statistical terms that can be viewed from the perspective of different moments of a function.

The *moments of a function* are measures that provide information regarding the shape of the graph of a function. In the case of a probability distribution, the first four moments are defined as follows:

- The mean is the first central moment.
- The variance is the second central moment.
- The skewness (discussed later) is the third central moment.
- The kurtosis (discussed later) is the fourth central moment.

More detailed information (including the relevant integrals) regarding moments of a function is available here:

https://en.wikipedia.org/wiki/Moment_(mathematics)#Variance

What is Skewness?

Skewness is a measure of the asymmetry of a probability distribution. A Gaussian distribution is symmetric, which means that its skew value is zero (it's not exactly zero, but close enough for our purposes). In addition, the skewness of a distribution is the *third* moment of the distribution.

A distribution can be skewed on the left side or on the right side. A *left-sided* skew means that the long tail is on the left side of the curve, with the following relationships:

mean < median < mode

A *right-sided* skew means that the long tail is on the right side of the curve, with the following relationships (compare with the left-sided skew):

mode < median < mean

If need be, you can transform skewed data to a normally distributed dataset using one of the following techniques (which depends on the specific use-case):

- Exponential transform
- Log transform
- Power transform

Perform an online search for more information regarding the preceding transforms and when to use each of these transforms.

What is Kurtosis?

Kurtosis is related to the skewness of a probability distribution, in the sense that both of them assess the asymmetry of a probability distribution. The kurtosis of a distribution is a scaled version of the *fourth* moment of the distribution, whereas its skewness is the *third* moment of the distribution. Note that the kurtosis of a univariate distribution equals 3.

If you are interested in learning about additional kurtosis-related concepts, you can perform an online search for information regarding mesokurtic, leptokurtic, and platykurtic types of "excess kurtosis."

DATA AND STATISTICS

This section contains various subsections that briefly discuss some of the challenges and obstacles that you might encounter when working with datasets. This section and subsequent sections introduce you to the following concepts:

- Correlation versus Causation
- The bias-variance tradeoff
- Types of bias
- The Central Limit Theorem
- Statistical inferences

Statistics typically involves data *samples*, which are subsets of observations of a population. The goal is to find well-balanced samples that provide a good representation of the entire population.

Although this goal can be very difficult to achieve, it's also possible to achieve highly accurate results with a very small sample size. For example, the Harris poll in the USA has been used for decades to analysis political trends. This poll computes percentages that indicate the favorability rating of political candidates, and it's usually within 3.5% of the correct percentage values. What's remarkable about the Harris poll is that its sample size is a mere 4,000 people that are from the US population, which is greater than 325,000,000 people.

Another aspect to consider is that each sample has a mean and variance, which do not necessarily equal the mean and variance of the actual population. However, the expected value of the sample mean and variance equal the mean and variance, respectively, of the population.

The Central Limit Theorem

Samples of a population have an interesting property. Suppose that you take a set of samples {S1, S3, …, Sn} of a population and you calculate the mean of those samples, which is {m1, m2, …, mn}. The Central Limit Theorem gives a remarkable result: given a set of samples of a population and the mean value of those samples, the distribution of the mean values can be approximated by a Gaussian distribution. Moreover, as the number of samples increases, the approximation becomes more accurate.

Correlation versus Causation

In general, datasets have some features (columns) that are more significant in terms of their set of values, and some features only provide additional information that does not contribute to potential trends in the dataset. For example, the passenger names in the list of passengers on the Titanic are unlikely to affect the survival rate of those passengers, whereas the gender of the passengers is likely to be an important factor.

In addition, a pair of significant features may also be "closely coupled" in terms of their values. For example, a real estate dataset for a set of houses will contain the number of bedrooms and the number of bathrooms for each house in the dataset. As you know, these values tend to increase together and also decrease together. For instance, have you ever seen a house that has 10 bedrooms and 1 bathroom, or a house that has 10 bathrooms and 1 bedroom? If you did find such a house, would you purchase that house as your primary residence?

The extent to which the values of two features change is called their *correlation*, which is a number between –1 and 1. Two "perfectly" correlated features have a correlation of 1, and two features that are not correlated have a correlation of 0. In addition, if the values of one feature decrease when the values of another feature increase, and vice versa, then their correlation is closer to –1 (and might also equal –1).

The causation between two features means that the values of one feature can be used to calculate the values of the second feature (within some margin of error).

Keep in mind this fundamental point about machine learning models: they can provide correlation but they cannot provide causation.

Statistical Inferences

Statistical thinking relates processes and statistics, whereas statistical inference refers to the process you use to make inferences about a population. Those inferences are based on statistics that are derived from samples of the population. The validity and reliability of those inferences depend on random sampling to reduce bias. There are various metrics that you can calculate to help you assess the validity of a model that has been trained on a particular dataset.

STATISTICAL TERMS RSS, TSS, R^2, AND F1 SCORE

Statistics is extremely important in machine learning, so it's not surprising that many concepts are common to both fields. Machine learning relies on a number of statistical quantities in order to assess the validity of a model, some of which are listed here:

- RSS
- TSS
- R^2

The term RSS is the "residual sum of squares" and the term TSS is the "total sum of squares." Moreover, these terms are used in regression models.

As a starting point so we can simplify the explanation of the preceding terms, suppose that we have a set of points {(x1,y1), . . . , (xn,yn)} in the Euclidean plane. In addition, let's define the following quantities:

- (x,y) is any point in the dataset.
- y is the y-coordinate of a point in the dataset.
- y_ is the mean of the y-values of the points in the dataset.
- y_hat is the y-coordinate of a point on a best-fitting line.

Just to be clear, (x,y) is a point in the *dataset*, whereas (x,y_hat) is the corresponding point that lies on the *best fitting line*. With these definitions in mind, the definitions of RSS, TSS, and R^2 are listed here (n equals the number of points in the dataset):

RSS = (y – y_hat)**2/n
TSS = (y – y_bar)**2/n
R^2 = 1 – RSS/TSS

We also have the following inequalities involving RSS, TSS, and R^2:

0 <= RSS <= TSS
0 <= RSS/TSS <= 1
0 <= 1 – RSS/TSS <= 1
0 <= R^2 <= 1

When RSS is close to 0, then RSS/TSS is also close to zero, which means that R^2 is close to 1. Conversely, when RSS is close to TSS, then RSS/TSS is close to 1, and R^2 is close to 0. In general, a larger R^2 is preferred (i.e., the model is closer to the data points), but a lower value of R^2 is not necessarily a bad score.

What is an F1 score?

In machine learning, an F1 score is for models that are evaluated on a feature that contains categorical data, and the p-value is useful for machine learning in general. An F1 score is a measure of the accuracy of a test, and it's defined as the *harmonic mean* of precision and recall. Here are the relevant formulas, where p is the precision and r is the recall:

p = (# of correct positive results)/(# of all positive results)
r = (# of correct positive results)/(# of all relevant samples)

F1-score = 1/[((1/r) + (1/p))/2]
= 2*[p*r]/[p+r]

The best value of an F1 score is 0 and the worst value is 0. An F1 score is for categorical classification problems, whereas the R^2 value is typically for regression tasks (such as linear regression).

GINI IMPURITY, ENTROPY, AND PERPLEXITY

These concepts are useful for assessing the quality of a machine learning model and the latter pair are useful for dimensionality reduction algorithms.

Before we discuss the details of Gini impurity, suppose that P is a set of non-negative numbers {p1, p2, …, pn} such that the sum of all the numbers in the set P equals 1. Under these two assumptions, the values in the set P comprise a probability distribution, which we can represent with the letter p.

Now suppose that the set K contains a total of M elements, with k1 elements from class S1, k2 elements from class S2, . . ., and kn elements from class Sn. Compute the fractional representation for each class as follows:

$p1 = k1/M, p2 = k2/M, \ldots, pn = kn/M$

As you can surmise, the values in the set {p1, p2, …, pn} form a probability distribution. We're going to use the preceding values in the following subsections.

What is the Gini Impurity?

The *Gini impurity* (or score) is defined as follows, where `{p1,p2,…,pn}` is a probability distribution:

$$
\begin{aligned}
Gini &= 1 - [p1*p1 + p2*p2 + \ldots + pn*pn] \\
&= 1 - SUM\ pi*pi \text{ (for all i, where } 1<=i<=n)
\end{aligned}
$$

Since each pi is between 0 and 1, then $pi*pi <= pi$, which means that

$$
\begin{aligned}
1 &= p1 + p2 + \ldots + pn \\
&>= p1*p1 + p2*p2 + \ldots + pn*pn >= 0
\end{aligned}
$$

Hence Gini impurity >= 0

Since the Gini impurity is the sum of the squared values of a set of probabilities, the Gini impurity cannot be negative. Hence, we have derived the following result:

$$0 <= \text{Gini impurity} <= 1$$

What is Entropy?

Entropy is a measure of the expected ("average") number of bits required to encode the outcome of a random variable. The calculation for the entropy H (the letter E is reserved for Einstein's formula) as defined via the following formula:

$$
\begin{aligned}
H &= (-1)*[p1*\log p1 + p2 * \log p2 + \ldots + pn * \log pn] \\
&= (-1)* SUM\ [pi * \log(pi)] \text{ (for all i, where } 1<=i<=n)
\end{aligned}
$$

Calculating Gini Impurity and Entropy Values

For our first example, suppose that we have three classes: A and B and a cluster of 10 elements with 8 elements from class A and 2 elements from

class B. Therefore, p1 and p2 are 8/10 and 2/10, respectively. We can compute the Gini score as follows:

Gini = 1 – [p1*p1 + p2*p2]
= 1 – [64/100 + 04/100]
= 1 – 68/100
= 32/100
= 0.32

We can also calculate the entropy for this example as follows:

Entropy = (–1)*[p1 * log p1 + p2 * log p2]
= (–1)*[0.8 * log 0.8 + 0.2 * log 0.2]
= (–1)*[0.8 * (–0.322) + 0.2 * (–2.322)]
= 0.8 * 0.322 + 0.2 * 2.322
= 0.7220

For our second example, suppose that we have three classes A, B, C and a cluster of 10 elements with 5 elements from class A, 3 elements from class B, and 2 elements from class C. Therefore p1, p2, and p3 are 5/10, 3/10, and 2/10, respectively. We can compute the Gini score as follows:

Gini = 1 – [p1*p1 + p2*p2 + p3*p3]
= 1 – [25/100 + 9/100 + 04/100]
= 1 – 38/100
= 62/100
= 0.62

We can also calculate the entropy for this example as follows:

Entropy = (–1)*[p1 * log p1 + p2 * log p2]
= (–1)*[0.5*log0.5 + 0.3*log0.3 + 0.2*log0.2]
= (–1)*[–1 + 0.3*(–1.737) + 0.2*(–2.322)]
= 1 + 0.3*1.737 + 0.2*2.322
= 1.9855

In both examples, the Gini impurity is between 0 and 1. However, while the entropy is between 0 and 1 in the first example, it's greater than 1 in the second example (which was the rationale for showing you two examples).

A set whose elements belong to the same class has a Gini impurity equal to 0 and also its entropy equal to 0. For example, if a set has 10 elements that belong to class S1, then

Gini = 1 – SUM pi*pi
= 1 – p1*p1
= 1 – (10/10)*(10/10)
= 1 – 1 = 0

Entropy = (–1)*SUM pi*log pi
= (–1) * p1*log pi
= (–1) * (10/10) * log(10/10)
= (–1)*1*0 = 0

Multi-Dimensional Gini Index

The Gini index is a one-dimensional index that works well because the value is uniquely defined. However, when working with multiple factors, we need a multidimensional index. Unfortunately, the multi-dimensional Gini index (MGI) is not unique defined. While there have been various attempts to define an MGI that has unique values, they tend to be non-intuitive and mathematically much more complex. More information about MGI is available online:

https://link.springer.com/appendix/10.1007/978-981-13-1727-9_5

What is Perplexity?

Suppose that we have a probability distribution q, and that {x1, x2, ..., xN} is a set of sample values that is drawn from a model whose probability distribution is p. In addition, suppose that b is a positive integer (it's usually equal to 2). Now define the variable S as the following sum (logarithms are in base b not 10):

S = (–1/N) * [log q(x1) + log q(x2) + . . . + log q(xN)]
= (–1/N) * SUM log q(xi)

The formula for the perplexity PERP of the model q is b raised to the power S, as shown here:

PERP = b^S

If you compare the formula for entropy with the formula for S, you can see that the formulas are similar, so the perplexity of a model is somewhat related to the entropy of a model.

CROSS ENTROPY AND KL DIVERGENCE

Cross entropy is useful for understanding machine learning algorithms, and frameworks such as TensorFlow, which supports multiple APIs that involve cross entropy. KL divergence is relevant in machine learning, deep learning, and reinforcement learning.

As an interesting example, consider the credit assignment problem, which involves assigning credit to different elements or steps in a sequence. For example, suppose that users arrive at a webpage by clicking on a previous page, which was also reached by clicking on yet another webpage. Then, on the final webpage, users click on an ad. How much credit is given to the first and second webpages for the selected ad? One solution to this problem involves KL Divergence.

What is Cross Entropy?

The following formulas for logarithms are presented here because they are useful for the derivation of cross entropy in this section:

- log (a * b) = log a + log b
- log (a / b) = log a – log b
- log (1 / b) = (–1) * log b

In a previous section, you learned that for a probability distribution P with values {p1, p2, …, pn}, its entropy is H defined as follows:

H(P) = (–1)*SUM pi*log(pi)

Now let's introduce another probability distribution Q whose values are {q1, q2, …, qn}, which means that the entropy H of Q is defined as follows:

H(Q) = (–1)*SUM qi*log(qi)

We can define the cross entropy CE of Q and P as follows (notice the log qi and log pi terms and recall the formulas for logarithms in the previous section):

CE(Q,P) = SUM (pi*log qi) – SUM (pi*log pi)
= SUM (pi*log qi – pi*log pi)
= SUM pi*(log qi – log pi)
= SUM pi*(log qi/pi)

What is KL Divergence?

Now that entropy and cross entropy have been discussed, we can easily define the KL Divergence of the probability distributions Q and P as follows:

KL(P||Q) = CE(P,Q) – H(P)

The definitions of entropy H, cross entropy CE, and KL Divergence in this appendix involve discrete probability distributions P and Q. However, these concepts have counterparts in continuous probability density functions. The mathematics involves the concept of a Lebesgue measure on Borel sets (which is beyond the scope of this book) that are described online:

https://en.wikipedia.org/wiki/Lebesgue_measure

https://en.wikipedia.org/wiki/Borel_set

In addition to the KL Divergence, there is also the JS Divergence, also called the Jenson-Shannon Divergence, which was developed by Johan Jensen and Claude Shannon (who defined the formula for entropy). Although the JS Divergence is based on the KL Divergence, there is an important difference: the JS Divergence is symmetric and a true metric, whereas the KL Divergence is neither. More information regarding JS Divergence is available online:

https://en.wikipedia.org/wiki/Jensen–Shannon_divergence

What's Their Purpose?

The Gini impurity is often used to obtain a measure of the homogeneity of a set of elements in a decision tree. The entropy of a set is an alternative to its Gini impurity, and you will see both of these quantities used in machine learning models.

The *perplexity* value in NLP is one way to evaluate language models, which are probability distributions over sentences or texts. This value provides an estimate for the encoding size of a set of sentences.

Cross entropy is used in various methods in the TensorFlow framework, and the KL Divergence is used in various algorithms, such as the dimensionality reduction algorithm t-SNE. For more information about any of these terms, perform an online search to find online tutorials that provide detailed information.

COVARIANCE AND CORRELATION MATRICES

This section explains two important matrices: the covariance matrix and the correlation matrix. Although these are relevant for PCA (Principal Component Analysis) that is discussed later in this appendix, these matrices are not specific to PCA, which is the rationale for discussing them in a separate section. If you are familiar with these matrices, feel free to skim through this section.

The Covariance Matrix

As a reminder, the statistical quantity called the *variance* of a random variable X is defined as follows:

variance(x) = [SUM (x – xbar)*(x – xbar)]/n

A covariance matrix C is an nxn matrix whose values on the main diagonal are the variance of the variables X1, X2, . . ., Xn. The other values of C are the covariance values of each pair of variables Xi and Xj.

The formula for the covariance of the variables X and Y is a generalization of the variance of a variable, and the formula is shown here:

covariance(X, Y) = [SUM (x – xbar)*(y – ybar)]/n

Notice that you can reverse the order of the product of terms (multiplication is commutative), and therefore the covariance matrix C is a symmetric matrix:

covariance(X, Y) = covariance(Y, X)

Suppose that a CSV file contains four numeric features, all of which have been scaled appropriately, and let's call them x1, x2, x3, and x4. Then the covariance matrix C is a 4×4 square matrix that is defined with the following entries (pretend that there are outer brackets on the left side and the right side to indicate a matrix):

```
cov(x1, x1) cov(x1, x2) cov(x1, x3) cov(x1, x4)
cov(x2, x1) cov(x2, x2) cov(x2, x3) cov(x2, x4)
cov(x3, x1) cov(x3, x2) cov(x3, x3) cov(x3, x4)
cov(x4, x1) cov(x4, x2) cov(x4, x3) cov(x4, x4)
```

Note that the following is true for the diagonal entries in the preceding covariance matrix C:

```
var(x1, x1) = cov(x1, x1)
var(x2, x2) = cov(x2, x2)
var(x3, x3) = cov(x3, x3)
var(x4, x4) = cov(x4, x4)
```

In addition, C is a symmetric matrix, which is to say that the transpose of matrix C (rows become columns and columns become rows) is identical to the matrix C. The latter is true because (as you saw in the previous section) cov(x, y) = cov(y, x) for any feature x and any feature y.

Covariance Matrix: An Example

Suppose we have the two-column matrix A defined as follows:

```
      x  y
A = | 1  1 | <= 6 × 2 matrix
    | 2  1 |
    | 3  2 |
    | 4  2 |
    | 5  3 |
    | 6  3 |
```

The mean x_bar of column x is (1+2+3+4+5+6)/6 = 3.5, and the mean y_bar of column y is (1+1+2+2+3+3)/6 = 2. Subtract x_bar from column x and subtract y_bar from column y and we get matrix B, as shown here:

```
B = | –2.5 –1 | <= 6 × 2 matrix
    | –1.5 –1 |
    | –0.5  0 |
    |  0.5  0 |
    |  1.5  1 |
    |  2.5  1 |
```

Let Bt indicate the transpose of the matrix B (i.e., switch columns with rows and rows with columns), which means that Bt is a 2 × 6 matrix, as shown here:

```
Bt = |–2.5 –1.5 –0.5 0.5, 1.5, 2.5|
     |–1   –1    0   0    1    1  |
```

The covariance matrix C is the product of Bt and B, as shown here:

```
C = Bt * B = | 15.25 4 |
             | 4     8 |
```

Note that if the units of measure of features `x` and `y` do not have a similar scale, then the covariance matrix is adversely affected. In this case, the solution is simple: use the correlation matrix, which defined in the next section.

The Correlation Matrix

As you learned in the preceding section, if the units of measure of features `x` and `y` do not have a similar scale, then the covariance matrix is adversely affected. The solution involves the correlation matrix, which equals the covariance values cov(x,y) divided by the standard deviation stdx and stdy of x and y, respectively, as shown here:

corr(x,y) = cov(x,y)/[stdx * stdy]

The correlation matrix no longer has units of measure, and we can use this matrix to find the eigenvalues and eigenvectors.

Now that you understand how to calculate the covariance matrix and the correlation matrix, you are ready for an example of calculating eigenvalues and eigenvectors, which are the topic of the next section.

Eigenvalues and Eigenvectors

According to a well-known theorem in mathematics (whose proof you can find online), the eigenvalues of a symmetric matrix are real numbers. Consequently, the eigenvectors of C are vectors in a Euclidean vector space (not a complex vector space).

Before we continue, a non-zero vector x' is an eigenvector of the matrix C if there is a non-zero scalar lambda such that C*x' = lambda * x'.

Suppose that the eigenvalues of C are b1, b2, b3, and b4, in decreasing numeric order from left-to-right, and that the corresponding eigenvectors of C are the vectors w1, w2, w3, and w4. Then the matrix M that consists of the column vectors w1, w2, w3, and w4 represents the principal components.

CALCULATING EIGENVECTORS: A SIMPLE EXAMPLE

As a simple illustration of calculating eigenvalues and eigenvectors, suppose that the square matrix C is defined as follows:

```
C = | 1 3 |
    | 3 1 |
```

Let I denote the 2 × 2 identity matrix, and let b' be an eigenvalue of C, which means that there is an eigenvector x' such that

C* x' = b' * x', or
(C−b*I)*x' = 0 (the right side is a 2 × 1 vector)

Since x' is non-zero, that means the following is true (where det refers to the determinant of a matrix):

```
det(C−b*I) = det |1−b 3  | = (1−b)*(1−b)−9 = 0
                 |3   1−b|
```

We can expand the quadratic equation in the preceding line to obtain

```
det(C−b*I) = (1−b)*(1−b) − 9
           = 1 − 2*b + b*b − 9
           = −8 − 2*b + b*b
           = b*b − 2*b − 8
```

Use the quadratic formula (or perform factorization by visual inspection) to determine that the solution for det(C-b*I) = 0 is b = −2 or b = 4. Next, substitute b = −2 into (C−b*I)x' = 0 to obtain the following result:

```
|1−(−2) 3     | |x1| = |0|
|3      1−(−2)| |x2|   |0|
```

The preceding reduces to the following identical equations:

3*x1 + 3*x2 = 0
3*x1 + 3*x2 = 0

The general solution is x1 = -x2, and we can choose any non-zero value for x2, so let's set x2 = 1, which yields x1 = −1. Therefore, the eigenvector [−1, 1] is associated with the eigenvalue −2. In a similar fashion, if x' is an eigenvector whose eigenvalue is 4, then [1, 1] is an eigenvector.

Notice that the eigenvectors [−1, 1] and [1, 1] are orthogonal because their inner product is zero, as shown here:

[−1, 1] * [1, 1] = (−1)*1 + (1)*1 = 0

In fact, the set of eigenvectors of a square matrix (whose eigenvalues are real) are always orthogonal, regardless of the dimensionality of the matrix.

Gauss Jordan Elimination (optional)

This simple technique enables you to find the solution to systems of linear equations "in place," which involves a sequence of arithmetic operations to transform a given matrix to an identity matrix.

The following example combines the Gauss-Jordan elimination technique (which finds the solution to a set of linear equations) with the "bookkeeper's method," which determines the inverse of an invertible matrix (its determinant is non-zero).

This technique involves two adjacent matrices: the left-side matrix is the initial matrix and the right-side matrix is an identity matrix. Next, perform various linear operations on the left-side matrix to reduce it to an identity matrix. The matrix on the right side equals its inverse. For example, consider the following pair of linear equations whose solution is x = 1 and y = 2:

2*x + 2*y = 6
4*x – 1*y = 2

Step 1: Create a 2 × 2 matrix with the coefficients of x in column 1 and the coefficients of y in column two, followed by the 2 × 2 identity matrix, and finally a column from the numbers on the right of the equals sign:

```
| 2   2 | 1 0 | 6|
| 4 –1 | 0 1 | 2|
```

Step 2: Add (–2) times the first row to the second row:

```
| 2    2 |   1 0 |6    |
| 0  –5 | –2 1 |–10|
```

Step 3: Divide the second row by 5:

```
| 2   2 |     1    0  |6       |
| 0 –1 | –2/5 1/5 |–10/5|
```

Step 4: Add 2 times the second row to the first row:

```
| 2    0 |   1/5 2/5 |  2|
| 0  –1 | –2/5 1/5 |–2|
```

Step 5: Divide the first row by 2:

```
| 1    0 | –2/10 2/10 |  1|
| 0  –1 | –2/5    1/5  |–2|
```

Step 6: Multiply the second row by (–1):

```
| 1  0  | –2/10 2/10 |1|
| 0  1  |    2/5 –1/5  |2|
```

As you can see, the left-side matrix is the 2 × 2 identity matrix, the right-side matrix is the inverse of the original matrix, and the right-most column is the solution to the original pair of linear equations (x=1 and y=2).

PCA (PRINCIPAL COMPONENT ANALYSIS)

PCA is a linear dimensionality reduction technique for determining the most important features in a dataset. This section discusses PCA because it's a very popular technique that you will encounter frequently. Other techniques are more efficient than PCA, so later on it's worthwhile to learn other dimensionality reduction techniques as well.

Keep in mind the following points regarding the PCA technique:

- PCA is a variance-based algorithm.
- PCA creates variables that are linear combinations of the original variables.
- The new variables are all pair-wise orthogonal.
- PCA can be a useful pre-processing step before clustering.
- PCA is generally preferred for data reduction.

PCA can be useful for variables that are strongly correlated. If most of the coefficients in the correlation matrix are smaller than 0.3, PCA is not helpful. PCA provides some advantages: less computation time for training a model (for example, using only five features instead of 100 features), a simpler model, and the ability to render the data visually when two or three features are selected. Here is a key point about PCA:

PCA calculates the eigenvalues and the eigenvectors of the covariance (or correlation) matrix C.

If you have four or five components, you won't be able to display them visually, but you could select subsets of three components for visualization, and perhaps gain some additional insight into the dataset.

The PCA algorithm involves the following sequence of steps:

1. Calculate the correlation matrix (from the covariance matrix) C of a dataset.
2. Find the eigenvalues of C.
3. Find the eigenvectors of C.
4. Construct a new matrix that comprises the eigenvectors.

The covariance matrix and correlation matrix were explained in a previous section. You also saw the definition of eigenvalues and eigenvectors, along with an example of calculating eigenvalues and eigenvectors.

The eigenvectors are treated as column vectors that are placed adjacent to each other in decreasing order (from left-to-right) with respect to their associated eigenvectors.

PCA uses the variance as a measure of information: the higher the variance, the more important the component. In fact, PCA determines the eigenvalues and eigenvectors of a covariance matrix (discussed in a previous section), and constructs a new matrix whose columns are eigenvectors, ordered from left-to-right in a sequence that matches the corresponding sequence of eigenvalues: the left-most eigenvector has the largest eigenvalue, the next eigenvector has the second-largest eigenvalue, and continuing in this fashion until the right-most eigenvector (which has the smallest eigenvalue).

Alternatively, there is an interesting theorem in linear algebra: if `C` is a symmetric matrix, then there is a diagonal matrix `D` and an orthogonal matrix `P` (the

columns are pair-wise orthogonal, which means their pair-wise inner product is zero), such that the following holds:

```
C = P * D * Pt (where Pt is the transpose of matrix P)
```

The diagonal values of `D` are eigenvalues, and the columns of `P` are the corresponding eigenvectors of the matrix `C`.

Fortunately, we can use NumPy and Pandas to calculate the mean, standard deviation, covariance matrix, correlation matrix, as well as the matrices `D` and `P` to determine the eigenvalues and eigenvectors.

As an interesting point: any positive definite square matrix has real-valued eigenvectors, which also applies to the covariance matrix `C` because it is a real-valued symmetric matrix.

The New Matrix of Eigenvectors

The previous section described how the matrices `D` and `P` are determined. The left-most eigenvector of `D` has the largest eigenvalue, the next eigenvector has the second-largest eigenvalue, and so forth. The eigenvector with the largest eigenvalue is the principal component of the dataset. The eigenvector with the second-largest eigenvalue is the second principal component, and so forth. You specify the number of principal components that you want via the `n_components` hyper parameter in the PCA class of Sklearn.

As a simple and minimalistic example, consider the following code block that uses PCA for a (somewhat contrived) dataset:

```
import numpy as np
from sklearn.decomposition import PCA
data = np.array([[-1,-1], [-2,-1], [-3,-2], [1,1], [2,1], [3,2]])
pca = PCA(n_components=2)
pca.fit(X)
```

Note that a trade-off here: we greatly reduce the number of components, which reduces the computation time and the complexity of the model, but we also lose some accuracy. However, if the unselected eigenvalues are small, we lose only a small amount of accuracy.

Now let's use the following notation:

- NM denotes the matrix with the new principal components.
- NMt is the transpose of NM.
- PC is the matrix of the subset of selected principal components.
- SD is the matrix of scaled data from the original dataset.
- SDt is the transpose of SD.

Then the matrix NM is calculated via the following formula:

```
NM = PCt * SDt
```

Although PCA is a useful technique for dimensionality reduction, keep in mind the following limitations of PCA:

- less suitable for data with non-linear relationships
- less suitable for special classification problems

A related algorithm is called Kernel PCA, which is an extension of PCA that introduces a non-linear transformation so you can still use the PCA approach.

WELL-KNOWN DISTANCE METRICS

There are several similarity metrics available, such as item similarity metrics, the Jaccard (user-based) similarity, and cosine similarity (which is used to compare vectors of numbers). The following subsections introduce you to these similarity metrics.

Another well-known distance metric is the so-called "taxicab" metric, which is also called the Manhattan distance metric. Given two points A and B in a rectangular grid, the taxicab metric calculates the distance between two points by counting the number of "blocks" that must be traversed in order to reach B from A (the other direction has the same taxicab metric value). For example, if you need to travel two blocks north and then three blocks east in a rectangular grid, then the Manhattan distance is 5.

There are various other metrics available, which you can learn about by searching Wikipedia. In the case of NLP, the most commonly used distance metric is calculated via the cosine similarity of two vectors, and it's derived from the formula for the inner ("dot") product of two vectors.

Pearson Correlation Coefficient

The *Pearson similarity* is the Pearson coefficient between two vectors. You are given random variables `X` and `Y`, and the following terms:

```
std(X)   = standard deviation of X
std(Y)   = standard deviation of Y
cov(X,Y) = covariance of X and Y
```

Then the Pearson correlation coefficient `rho(X,Y)` is defined as follows:

```
              cov(X,Y)
rho(X,Y) =  -------------
            std(X)*std(Y)
```

The Pearson coefficient is limited to items of the same type. More information about the Pearson correlation coefficient is available online:

https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

Jaccard Index (or Similarity)

The Jaccard similarity is based on the number of users that have rated item `A` *and* `B` (the cardinality of A intersect B) divided by the number of users who have rated either `A` *or* `B` (the cardinality of A union B). The Jaccard similarity is based on unique words in a sentence and is unaffected by duplicates, whereas the cosine similarity is based on the length of all word vectors (which changes when duplicates are added). The choice between cosine similarity and Jaccard similarity depends on whether word duplicates are important.

The following Python method illustrates how to compute the Jaccard similarity of two sentences:

```
def get_jaccard_sim(str1, str2):
  set1 = set(str1.split())
  set2 = set(str2.split())
  set3 = set1.intersection(set2)
  # (size of intersection) / (size of union):
  return float(len(set3)) / (len(set1) + len(set2) - len(set3))
```

The Jaccard similarity can be used in situations involving Boolean values, such as product purchases (true/false), instead of numeric values. More information is available online:

https://en.wikipedia.org/wiki/Jaccard_index

Local Sensitivity Hashing (optional)

If you are familiar with hash algorithms, you know that they are algorithms that create a hash table that associate items with a value. The advantage of hash tables is that the lookup time to determine whether an item exists in the hash table is constant.

Of course, it's possible for two items to "collide," which means that they both occupy the same bucket in the hash table. In this case, a bucket can consist of a list of items that can be searched in more or less constant time. If there are too many items in the same bucket, then a different hashing function can be selected to reduce the number of collisions. The goal of a hash table is to minimize the number of collisions.

The Local Sensitivity Hashing (LSH) algorithm hashes similar input items into the same "buckets." In fact, the goal of LSH is to *maximize* the number of collisions, whereas traditional hashing algorithms attempt to *minimize* the number of collisions.

Since similar items end up in the same buckets, LSH is useful for data clustering and nearest neighbor searches. Moreover, LSH is a dimensionality reduction technique that places data points of high dimensionality closer together in a lower-dimensional space, while simultaneously preserving the relative distances between those data points. More details about LSH are available online:

https://en.wikipedia.org/wiki/Locality-sensitive_hashing

TYPES OF DISTANCE METRICS

Non-linear dimensionality reduction techniques can also have different distance metrics. For example, linear reduction techniques can use the Euclidean distance metric (based on the Pythagorean theorem).

However, you need to use a different distance metric to measure the distance between two points on a sphere (or some other curved surface). In the case of NLP, the cosine similarity metric is frequently used to measure the distance between word embeddings (which are vectors of floating point numbers that represent words or tokens).

Distance metrics are used for measuring physical distances, and some well-known distance metrics are listed here:

- Euclidean distance
- Manhattan distance
- Chebyshev distance

The Euclidean algorithm also obeys the "triangle inequality," which states that for any triangle in the Euclidean plane, the sum of the lengths of any pair of sides must be greater than the length of the third side.

In spherical geometry, you can define the distance between two points as the arc of a great circle that passes through the two points (always selecting the smaller of the two arcs when they are different).

In addition to physical metrics, there are algorithms that implement the concept of "edit distance" (the distance between strings), as listed here:

- Hamming distance
- Jaro–Winkler distance
- Lee distance
- Levenshtein distance
- Mahalanobis distance metric
- Wasserstein metric

The Mahalanobis metric is based on an interesting idea: given a point P and a probability distribution D, this metric measures the number of standard deviations that separate point P from distribution D. More information about Mahalanobis is available online:

https://en.wikipedia.org/wiki/Mahalanobis_distance

In the branch of mathematics called *topology*, a metric space is a set for which distances between all members of the set are defined. Various metrics are available (such as the Hausdorff metric), depending on the type of topology.

The Wasserstein metric measures the distance between two probability distributions over a metric space X. This metric is also called the "earth mover's metric" for the following reason: given two unit piles of dirt, it's the measure of the minimum cost of moving one pile on top of the other pile.

KL Divergence bears some superficial resemblance to the Wasserstein metric. However, there are some important differences between them. Specifically, the Wasserstein metric has the following properties:

1. It is a metric.
2. It is symmetric.
3. It satisfies the triangle inequality.

The KL Divergence has the following properties:

1. It is not a metric (it's a divergence).
2. It is not symmetric: KL(P,Q) != KL(Q,P).
3. It does not satisfy the triangle inequality.

Note that the JS (Jenson-Shannon) Divergence (which is based on the KL Divergence) is a true metric, which would enable a more meaningful comparison with other metrics (such as the Wasserstein metric). More information is available online:

https://stats.stackexchange.com/questions/295617/what-is-the-advantages-of-wasserstein-metric-compared-to-kullback-leibler-diverg

https://en.wikipedia.org/wiki/Wasserstein_metric

WHAT IS BAYESIAN INFERENCE?

Bayesian inference is an important technique in statistics that involves statistical inference and Bayes' theorem to update the probability for a hypothesis as more information becomes available. Bayesian inference is often called "Bayesian probability," and it's important in dynamic analysis of sequential data.

Bayes Theorem

Given two sets A and B, let's define the following numeric values (all of them are between 0 and 1):

P(A) = probability of being in set A
P(B) = probability of being in set B
P(Both) = probability of being in A intersect B
P(A|B) = probability of being in A (given you're in B)
P(B|A) = probability of being in B (given you're in A)

Then the following formulas are also true:

P(A|B) = P(Both)/P(B) (#1)
P(B|A) = P(Both)/P(A) (#2)

Multiply the preceding pair of equations by the term that appears in the denominator to obtain these equations:

P(B)*P(A|B) = P(Both) (#3)
P(A)*P(B|A) = P(Both) (#4)

Now set the left-side of Equations #3 and #4 equal to each another, and that gives us this equation:

P(B)*P(A|B) = P(A)*P(B|A) (#5)
Divide both sides of #5 by P(B) to obtain this well-known equation:
P(A|B) = P(A)*P(A|B)/P(B) (#6)

Some Bayesian Terminology

In the previous section, we derived the following relationship:

P(h|d) = (P(d|h) * P(h)) / P(d)

There is a name for each of the four terms in the preceding equation, as discussed below.

First, the *posterior probability* is P(h|d), which is the probability of hypothesis h given the data d.

Second, P(d|h) is the probability of data d given that the hypothesis h was true.

Third, the *prior probability* of h is P(h), which is the probability of hypothesis h being true (regardless of the data).

Finally, P(d) is the probability of the data (regardless of the hypothesis)

We are interested in calculating the posterior probability of P(h|d) from the prior probability p(h) with P(d) and P(d|h).

What is MAP?

The maximum a posteriori (MAP) hypothesis is the hypothesis with the highest probability, which is the maximum probable hypothesis. This can be written as follows:

```
MAP(h) = max(P(h|d))
```

or

```
MAP(h) = max((P(d|h) * P(h)) / P(d))
```

or

```
MAP(h) = max(P(d|h) * P(h))
```

Why Use Bayes' Theorem?

Bayes' Theorem describes the probability of an event based on the prior knowledge of the conditions that might be related to the event. If we know the conditional probability, we can use Bayes' rule to find out the reverse probabilities. The previous statement is the general representation of the Bayes' rule.

SUMMARY

This appendix started with a discussion of probability, expected values, and the concept of a random variable. Then you learned about some basic statistical concepts, such as mean, median, mode, variance, and standard deviation. Next, you learned about the terms RSS, TSS, R^2, and F1 score. In addition, you had an introduction to the concepts of skewness, kurtosis, the Gini Impurity, entropy, perplexity, cross entropy, and KL Divergence.

Next, you learned about covariance and correlation matrices and how to calculate eigenvalues and eigenvectors. Then you were introduced to the dimensionality reduction technique known as PCA (Principal Component Analysis), after which you learned about Bayes' Theorem.

INDEX

A

AND, OR, and NOT operators, 153–154
Arithmetic aggregate operators
  finding average values, 157–158
  SELECT clause, 158–159
Arithmetic operator, 154–156
ASC keyword, 116
Atomicity, Consistency, Isolation, and Durability (ACID), 5–6

B

Bayesian inference, 282–283
Bayes' Theorem, 282–283
Binary large object (BLOB), 21
BIN() function, 180
Boolean operations
  BETWEEN, 150
  IN, 151
  IS NULL, 151
  LIKE, 151
Built-in number functions, 164–165

C

CASE keyword, 174–176
CAST() function, 181–183
CEIL() and FLOOR() function, 131
COALESCE() function, 181
COMBINED GROUP BY, HAVING, AND ORDER BY CLAUSE, 101–102
Command line utilities, 241
COMMIT and ROLLBACK statement, 231
Common table expression (CTE)
  definition, 166
  JOIN keyword, 167–168
  WITH keyword, 168
  mean, standard deviation, and z-scores, 169–171
  recursive SQL query, 168–169
  single and multiple attributes, 167
Compass, 198–199
Consistency, Availability, and Partition Tolerance (CAP) Theorem, 240–241
CONVERT() function, 181
CONV() function, 180–181
Correlated subqueries, 82
CREATE keyword, 19–20
Cross entropy, 270–272
CTE. *see* Common table expression (CTE)

D

Database backups, restoring data, and upgrades, 241–242
Database engines, 225–226
Database normalization, 7–8
Database optimization, 232
  performance tuning, 232–233
Database replication, 239–240
Database tables, 25–27
  attributes, 37–38
  create
    from command line, 36–37
    with Japanese text, 35–36
    manual, 32–34
    via SQL script, 34–35

creating tables from existing tables
creating copies of existing tables, 58
memory-stored tables, 56
temporary tables, 57–58
drop, 32
INFORMATION_SCHEMA table, 27–28
PROCESSLIST table, 28
Database user management
create and alter, 214–215
drop, 215–216
list users, 214
roles
create roles and grant privileges, 216–218
revoke roles and drop roles, 218
Data cleaning
from command line, 250–254
convert strings to date values, 248–250
handle mismatched attribute values, 247–248
replace multiple values into a single value, 246–247
replace NULL with 0, 244
replace NULL with the average value, 244–246
Data Control Language (DCL), 18
Data Definition Language (DDL), 18
Data Manipulation Language (DML), 18
Data Query Language (DQL), 18
Date-related operations
arithmetic operations, 111–112
components and formats, 112–114
CURRENT_DATE() function, 106
date_format() function, 108–109
day and month-related functions, 107–108
NOW() function, 106
ranges, 109–110
SYSDATE function, 106
WEEK() function, 114–116
Denormalization, 227
DESC keyword, 116
Distance metrics, 281. *See also* Well-known distance metrics
Distributed database (DDB), 240

E
Entity Relationship Diagram (ERD), 230
Entity Relationship Modeling (ERM), 230
Entity relationships, 64–65
EXPLAIN statement, 235–237

F
F1 score, 267
Fugue, 197–198

G
Gini impurity and entropy, 268–270
GREATEST() function, 180
GROUP BY clause, 90–93
and ROLLUP clause, 95–96
GROUP BY, HAVING, AND ORDER BY CLAUSE, 100–101

H
HAVING clause, 91–92
Histogram, 90
on a table copy, 93–95

I
Index(es)
clustered index, 59
column selection, 62–63
considerations, 61–62
creation, 59–60
description, 58–59
disable indexes, 62
enable and disable, 60
finding columns, 63
invisible index, 59
overhead of, 61
unique index, 59
view and drop, 60–61
InnoDB, 3

J
Jaccard similarity, 280
JOIN statement, 68
CROSS JOIN statement, 69, 73
delete duplicate attributes, 74–75
four-table RDBMS, 69–71
INNER JOIN statement, 69, 71
LEFT JOIN statement, 69, 72
NATURAL JOIN, 73
RIGHT JOIN statement, 69, 72–73
SELF JOIN statement, 69
on tables with international text, 75–76
JSON data, 242–244

K
Keys
composite key, 80
foreign key, 79

parent_child.sql, 80–82
vs. primary keys, 79–80
non-key columns, 79
primary key, 79
KL Divergence, 270–272
Kurtosis, 265

L
LEAST() function, 180
Linear regression, 171–172
Local Sensitivity Hashing (LSH) algorithm, 280
Log, exponential, and trigonometric functions, 132–134

M
MariaDB database, 3
MATCH() function and text search, 165–166
MAX() and MIN() functions, 138–139
with subqueries, 139–143
top-ranked numeric values, 143–144
Maximum a posteriori (MAP) hypothesis, 283
Money transfer between bank accounts, RDBMS, 6–7
MongoDB
APIs, 191–192
collections and documents
aggregate() function, 197
cellphones collection, 194–195
CREATE, 193
document format, 193
find() function, 195–196
insertOne() function, 196
mongoimport utility, 197
update() function, 196
Compass, 198–199
features, 190
Fugue, 197–198
installation, 190
launch, 190–191
meta characters, 192–193
PyMongo, 199–200
Multi-dimensional Gini index (MGI), 270
Multiple-row functions, 163
MyRocks, 3
MySQL
aliases, 38–39
connector/Python API
create_fun_table.py, 205–206
database connection, 204
mysql_pandas.py, 204–205
database operations
create, 22
display, 22–23
drop, 23
import/export, 23–24
rename, 24–25
data types
BLOB and TEXT, 21
CHAR and VARCHAR, 20
FLOAT and DOUBLE, 21
string-based, 20–21
download, 2
installation, 3
vs. MariaDB, 3
storage engines, 3
tables (*see* Database tables)
useful links for, 3–4

N
NewSQL, 188
Non-correlated subquery, 82
Non-relational database systems
advantages, 187
document store, 186
graph databases, 186
key/value store, 186
wide document store, 186
Normalization, 226–227
NoSQL, 187–188
databases, 189–190
data types, 188–189
vs. RDBMSs, 188
NULL values, 176–179
Numeric functions
calculated columns, 128–129
FORMAT() function, 126
LEN() function, 126–127
MOD() function, 127–128
POSITION() function, 128
ROUND() function, 128

O
OFFSET() keyword, 145
2021 Olympics medals in Japan
OlympicsJAPN2021.csv, 97
olympics.sql, 97–98
RANK() operator, 98–99
ROLLUP keyword, 98
ORDER BY clause, 91–93
with aggregate functions, 160–161

ascending or descending order, 159–160
largest distinct values and frequency of values, 161–163

P

`PARTITION BY` clause, 99
Perplexity, 270
Probability
conditional probability, 258
description, 257
expected value calculation, 258–260
random variables, 260
discrete and continuous, 260
well-known probability distributions, 260–261
PyMongo, 199–200

Q

Query execution order, 67–68
Query optimization
cost-based optimization, 234
performance tuning tools, 233–234
table fragmentation, 234
table partitioning, 234–235

R

`RAND()` function, 132
Relational DataBase Management System (RDBMS), 4
ACID, 5–6
characteristics, 5
logical schema, 5
MongoDB, 189
needs, 6–7
normalization, 7–8
vs. NoSQL, 188–189
vs. tables, 4–5, 8–9
`customers` table, 10–11
`item_desc` table, 13–14
`line_items` table, 12–13
`purchase_orders` table, 11–12
`ROUND()` function, 129–130
RSS, TSS, and R^2, 266–267

S

`SAVEPOINT` statement, 231–232
Scalable databases, 240
Scalar functions, 135
Scaling
federation, 239
sharding, 238–239
SQL tuning, 237–238
Schemas, 227–228
`SESSION()` function, 181
Set operators, 152–153
Single-row functions, 163–164
Skewness, 264
SQLAlchemy and Pandas, 200–203
SQLite
DB Browser, 209
features, 207
installation, 207–208
SQLiteStudio, 208–209
SQLiteDict, 209–211
SQLiteStudio, 208–209
Statistics
Bayesian inference, 282–283
Central Limit Theorem, 265
Chebyshev's inequality, 263
correlation *vs*. causation, 266
covariance and correlation matrices, 272–274
Cross entropy and KL Divergence, 270–272
eigenvalues and eigenvectors, 274–275
F1 score, 267
Gauss-Jordan elimination technique, 275–276
Gini impurity and entropy, 268–270
mean, median and mode, 261–262
moments of a function, 264
multi-dimensional Gini index (MGI), 270
PCA technique, 276–278
perplexity, 270
population, sample, and population variance, 263
p-value, 263–264
RSS, TSS, and R^2, 266–267
skewness and kurtosis, 264–265
statistical inference, 266
variance and standard deviation, 262–263
well-known distance metrics
Jaccard similarity, 280
Local Sensitivity Hashing (LSH) algorithm, 280
Pearson correlation coefficient, 279
types, 281–282
Stored functions, 222–223
Stored procedures
advantages and disadvantages, 219
`double_number.sql`, 221–222
features, 218–219

IN and OUT parameters, 219
stored1.sql, 220–221
String functions
CONCAT() function, 147–148
LCASE() function, 146
MID() function, 146–147
SUBSTR() function, 147–150
UCASE() function, 146
String operators, 165
Structured Query Language (SQL)
ad hoc reports, 119
aggregate functions, 136–138
arithmetic aggregate operators
finding average values, 157–158
SELECT clause, 158–159
arithmetic operator, 154–156
ASC keyword, 116
BIN() function, 180
boolean operations
BETWEEN, 150
IN, 151
IS NULL, 151
LIKE, 151
built-in number functions, 164–165
CASE keyword, 174–176
CAST() function, 181–183
CEIL() and FLOOR() function, 131
character functions
case manipulation functions, 164
character manipulation functions, 164
COALESCE() function, 181
column alias, 116–117
CONVERT() function, 181
CONV() function, 180–181
CREATE keyword, 19–20
CTE (*see* Common table expression (CTE))
date-related operations
arithmetic operations, 111–112
components and formats, 112–114
CURRENT_DATE() function, 106
date_format() function, 108–109
day and month-related functions, 107–108
NOW() function, 106
ranges, 109–110
SYSDATE function, 106
WEEK() function, 114–116
DCL and DDL, 18
DESC keyword, 116
DQL and DML, 18
formatting tools, 29
four-table join, 102–105
GREATEST() function, 180
LEAST() function, 180
linear regression, 171–172
log, exponential, and trigonometric functions, 132–134
MATCH() function and text search, 165–166
MAX() and MIN() functions, 138–139
with subqueries, 139–143
top-ranked numeric values, 143–144
modification times, 110–111
multiple-row functions, 163
NULL values, 176–179
numeric functions
calculated columns, 128–129
FORMAT() function, 126
LEN() function, 126–127
MOD() function, 127–128
POSITION() function, 128
ROUND() function, 128
object privileges, 19
OFFSET() keyword, 145
AND, OR, and NOT operators, 153–154
ORDER BY clause
with aggregate functions, 160–161
ascending or descending order, 159–160
largest distinct values and frequency of values, 161–163
query execution order, 67–68
RAND() function, 132
ROUND() function, 129–130
scalar functions, 135
SESSION() function, 181
set operators, 152–153
single-row functions, 163–164
statements, 19
string functions
CONCAT() function, 147–148
LCASE() function, 146
MID() function, 146–147
SUBSTR() function, 147–150
UCASE() function, 146
string operators, 165
summary reports, 118–119
cumulative totals, 123
sold items, 119–120
sold price, 120–121
subtotals, 122

system privileges, 18
TCL, 18
user-defined functions, 218
variables, 117–118
window functions
aggregate functions, 173
description, 172
functions for time series, 173
RANK and DENSE_RANK functions, 173–174
rank-related functions, 173
statistical functions, 173
Subquery
to find customers without purchase orders, 83–85
heights.sql, 88–90
MAX() and AVG() functions, 88
IN and NOT IN clause, 85–86
SOME, ALL, ANY clause, 86–88
types, 82–83
Summary reports, 118–119
cumulative totals, 123
sold items, 119–120
sold price, 120–121
subtotals, 122
System privileges, 18

T
Transaction, 230–231
Transaction Control Language (TCL), 18
Trigger, 223–225

U
User-defined functions, 218

V
Variables, 117–118
random, 260
View
advantages, 77–78
CREATE VIEW, 77
description, 76–77
DROP VIEW, 77
multiple table, 78
single table, 78
updatable view, 79

W
Well-known distance metrics
Jaccard similarity, 280
Local Sensitivity Hashing (LSH) algorithm, 280
Pearson correlation coefficient, 279
types, 281–282
Window functions
aggregate functions, 173
description, 172
functions for time series, 173
RANK and DENSE_RANK functions, 173–174
rank-related functions, 173
statistical functions, 173
Workbench, 228–230