O'REILLY®

Third Edition

Learning SQL

Generate, Manipulate, and Retrieve Data

Alan Beaulieu

O'REILLY®

Learning SQL

As data floods into your company, you need to put it to work right away—and SQL is the best tool for the job. With the latest edition of this introductory guide, author Alan Beaulieu helps developers get up to speed with SQL fundamentals for writing database applications, performing administrative tasks, and generating reports. You'll find new chapters on analytic functions, strategies for working with large databases, and SQL and big data.

Each chapter presents a self-contained lesson on a key SQL concept or technique using numerous illustrations and annotated examples. Exercises let you practice the skills you learn. Knowledge of SQL is a must for interacting with data. With *Learning SQL*, you'll quickly discover how to put the power and flexibility of this language to work.

- Move quickly through SQL basics and several advanced features
- Use SQL data statements to generate, manipulate, and retrieve data
- Create database objects such as tables, indexes, and constraints with SQL schema statements
- Learn how data sets interact with queries; understand the importance of subqueries
- Convert and manipulate data with SQL's built-in functions and use conditional logic in data statements

"From the basics of SQL to advanced topics such as analytical functions and working with large databases, this third edition of *Learning SQL* provides everything you need to know about SQL in today's modern database world."

—**Mark Richards**
Author of *Fundamentals of Software Architecture* (O'Reilly)

Alan Beaulieu has been designing, building, and implementing custom database applications for over 25 years. He's the coauthor of *Mastering Oracle SQL* (O'Reilly) and has written an online course on SQL for the University of California. Alan runs his own consulting company that specializes in database design and development for financial services and telecommunications.

DATABASES

US $59.99 CAN $79.99
ISBN: 978-1-492-05761-1

55999

9 781492 057611

Twitter: @oreillymedia
facebook.com/oreilly

THIRD EDITION

Learning SQL

Generate, Manipulate, and Retrieve Data

Alan Beaulieu

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Learning SQL

by Alan Beaulieu

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman
Development Editor: Jeff Bleiel
Production Editor: Deborah Baker
Copyeditor: Charles Roumeliotis
Proofreader: Chris Morris
Indexer: Angela Howard
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2005:	First Edition
April 2009:	Second Edition
April 2020:	Third Edition

Revision History for the Third Edition

2020-03-04: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492057611* for release details.

978-1-492-05761-1

[MBP]

Table of Contents

Preface

Programming languages come and go constantly, and very few languages in use today have roots going back more than a decade or so. Some examples are COBOL, which is still used quite heavily in mainframe environments; Java, which was born in the mid-1990s and has become one of the most popular programming languages; and C, which is still quite popular for operating systems and server development and for embedded systems. In the database arena, we have SQL, whose roots go all the way back to the 1970s.

SQL was initially created to be the language for generating, manipulating, and retrieving data from relational databases, which have been around for more than 40 years. Over the past decade or so, however, other data platforms such as Hadoop, Spark, and NoSQL have gained a great deal of traction, eating away at the relational database market. As will be discussed in the last few chapters of this book, however, the SQL language has been evolving to facilitate the retrieval of data from various platforms, regardless of whether the data is stored in tables, documents, or flat files.

Why Learn SQL?

Whether you will be using a relational database or not, if you are working in data science, business intelligence, or some other facet of data analysis, you will likely need to know SQL, along with other languages/platforms such as Python and R. Data is everywhere, in huge quantities, and arriving at a rapid pace, and people who can extract meaningful information from all this data are in big demand.

Why Use This Book to Do It?

There are plenty of books out there that treat you like a dummy, idiot, or some other flavor of simpleton, but these books tend to just skim the surface. At the other end of the spectrum are reference guides that detail every permutation of every statement in a language, which can be useful if you already have a good idea of what you want to do but just need the syntax. This book strives to find the middle ground, starting with some background of the SQL language, moving through the basics, and then progressing into some of the more advanced features that will allow you to really shine. Additionally, this book ends with a chapter showing how to query data in nonrelational databases, which is a topic rarely covered in introductory books.

Structure of This Book

This book is divided into 18 chapters and 2 appendixes:

Chapter 1, A Little Background
: Explores the history of computerized databases, including the rise of the relational model and the SQL language.

Chapter 2, Creating and Populating a Database
: Demonstrates how to create a MySQL database, create the tables used for the examples in this book, and populate the tables with data.

Chapter 3, Query Primer
: Introduces the `select` statement and further demonstrates the most common clauses (`select`, `from`, `where`).

Chapter 4, Filtering
: Demonstrates the different types of conditions that can be used in the `where` clause of a `select`, `update`, or `delete` statement.

Chapter 5, Querying Multiple Tables
: Shows how queries can utilize multiple tables via table joins.

Chapter 6, Working with Sets
: This chapter is all about data sets and how they can interact within queries.

Chapter 7, Data Generation, Manipulation, and Conversion
: Demonstrates several built-in functions used for manipulating or converting data.

Chapter 8, Grouping and Aggregates
: Shows how data can be aggregated.

Chapter 9, Subqueries
: Introduces subqueries (a personal favorite) and shows how and where they can be utilized.

Chapter 10, Joins Revisited
: Further explores the various types of table joins.

Chapter 11, Conditional Logic
: Explores how conditional logic (i.e., if-then-else) can be utilized in `select`, `insert`, `update`, and `delete` statements.

Chapter 12, Transactions
: Introduces transactions and shows how to use them.

Chapter 13, Indexes and Constraints
: Explores indexes and constraints.

Chapter 14, Views
: Shows how to build an interface to shield users from data complexities.

Chapter 15, Metadata
: Demonstrates the utility of the data dictionary.

Chapter 16, Analytic Functions
: Covers functionality used to generate rankings, subtotals, and other values used heavily in reporting and analysis.

Chapter 17, Working with Large Databases
: Demonstrates techniques for making very large databases easier to manage and traverse.

Chapter 18, SQL and Big Data
: Explores the transformation of the SQL language to allow retrieval of data from nonrelational data platforms.

Appendix A, ER Diagram for Example Database
: Shows the database schema used for all examples in the book.

Appendix B, Solutions to Exercises
: Shows solutions to the chapter exercises.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

Indicates a tip, suggestion, or general note. For example, I use notes to point you to useful new features in Oracle9*i*.

Indicates a warning or caution. For example, I'll tell you if a certain SQL clause might have unintended consequences if not used carefully.

Using the Examples in This Book

To experiment with the data used for the examples in this book, you have two options:

- Download and install the MySQL server version 8.0 (or later) and load the Sakila example database from *https://dev.mysql.com/doc/index-other.html*.
- Go to *https://www.katacoda.com/mysql-db-sandbox/scenarios/mysql-sandbox* to access the MySQL Sandbox, which has the Sakila sample database loaded in a MySQL instance. You'll have to set up a (free) Katacoda account. Then, click the Start Scenario button.

If you choose the second option, once you start the scenario, a MySQL server is installed and started, and then the Sakila schema and data are loaded. When it's ready, a standard `mysql>` prompt appears, and you can then start querying the sample database. This is certainly the easiest option, and I anticipate that most readers will choose this option; if this sounds good to you, feel free to skip ahead to the next section.

If you prefer to have your own copy of the data and want any changes you have made to be permanent, or if you are just interested in installing the MySQL server on your own machine, you may prefer the first option. You may also opt to use a MySQL server hosted in an environment such as Amazon Web Services or Google Cloud. In either case, you will need to perform the installation/configuration yourself, as it is beyond the scope of this book. Once your database is available, you will need to follow a few steps to load the Sakila sample database.

First, you will need to launch the `mysql` command-line client and provide a password, and then perform the following steps:

1. Go to *https://dev.mysql.com/doc/index-other.html* and download the files for "sakila database" under the Example Databases section.
2. Put the files in a local directory such as *C:\temp\sakila-db* (used for the next two steps, but overwrite with your directory path).
3. Type `source c:\temp\sakila-db\sakila-schema.sql;` and press Enter.
4. Type `source c:\temp\sakila-db\sakila-data.sql;` and press Enter.

You should now have a working database populated with all the data needed for the examples in this book.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata and any additional information. You can access this page at *https://oreil.ly/Learning_SQL3*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to thank my editor, Jeff Bleiel, for helping to make this third edition a reality, along with Thomas Nield, Ann White-Watkins, and Charles Givre, who were kind enough to review the book for me. Thanks also go to Deb Baker, Jess Haberman, and all the other folks at O'Reilly Media who were involved. Lastly, I thank my wife, Nancy, and my daughters, Michelle and Nicole, for their encouragement and inspiration.

CHAPTER 1

A Little Background

Before we roll up our sleeves and get to work, it would be helpful to survey the history of database technology in order to better understand how relational databases and the SQL language evolved. Therefore, I'd like to start by introducing some basic database concepts and looking at the history of computerized data storage and retrieval.

For those readers anxious to start writing queries, feel free to skip ahead to Chapter 3, but I recommend returning later to the first two chapters in order to better understand the history and utility of the SQL language.

Introduction to Databases

A *database* is nothing more than a set of related information. A telephone book, for example, is a database of the names, phone numbers, and addresses of all people living in a particular region. While a telephone book is certainly a ubiquitous and frequently used database, it suffers from the following:

- Finding a person's telephone number can be time consuming, especially if the telephone book contains a large number of entries.
- A telephone book is indexed only by last/first names, so finding the names of the people living at a particular address, while possible in theory, is not a practical use for this database.
- From the moment the telephone book is printed, the information becomes less and less accurate as people move into or out of a region, change their telephone numbers, or move to another location within the same region.

The same drawbacks attributed to telephone books can also apply to any manual data storage system, such as patient records stored in a filing cabinet. Because of the cumbersome nature of paper databases, some of the first computer applications developed were *database systems*, which are computerized data storage and retrieval mechanisms. Because a database system stores data electronically rather than on paper, a database system is able to retrieve data more quickly, index data in multiple ways, and deliver up-to-the-minute information to its user community.

Early database systems managed data stored on magnetic tapes. Because there were generally far more tapes than tape readers, technicians were tasked with loading and unloading tapes as specific data was requested. Because the computers of that era had very little memory, multiple requests for the same data generally required the data to be read from the tape multiple times. While these database systems were a significant improvement over paper databases, they are a far cry from what is possible with today's technology. (Modern database systems can manage petabytes of data, accessed by clusters of servers each caching tens of gigabytes of that data in high-speed memory, but I'm getting a bit ahead of myself.)

Nonrelational Database Systems

> This section contains some background information about prerelational database systems. For those readers eager to dive into SQL, feel free to skip ahead a couple of pages to the next section.

Over the first several decades of computerized database systems, data was stored and represented to users in various ways. In a *hierarchical database system*, for example, data is represented as one or more tree structures. Figure 1-1 shows how data relating to George Blake's and Sue Smith's bank accounts might be represented via tree structures.

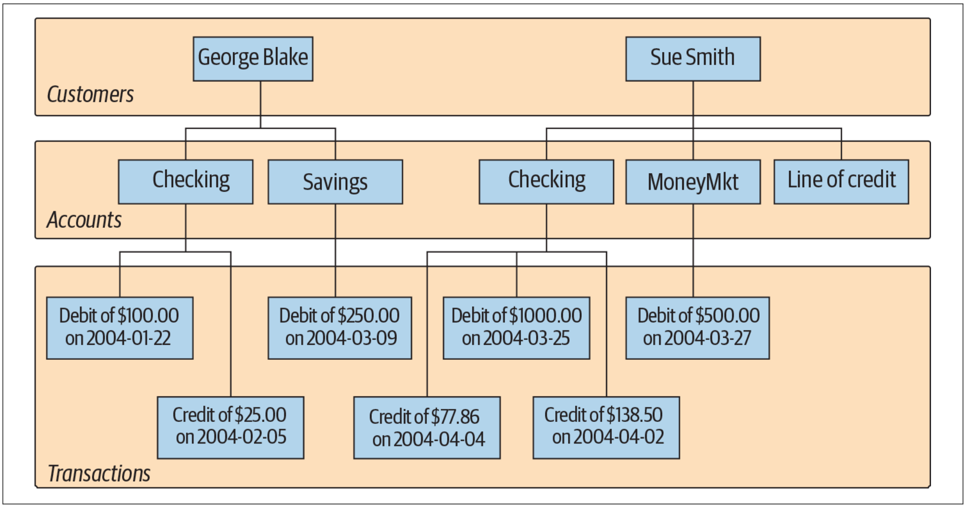

Figure 1-1. Hierarchical view of account data

George and Sue each have their own tree containing their accounts and the transactions on those accounts. The hierarchical database system provides tools for locating a particular customer's tree and then traversing the tree to find the desired accounts and/or transactions. Each node in the tree may have either zero or one parent and zero, one, or many children. This configuration is known as a *single-parent hierarchy*.

Another common approach, called the *network database system*, exposes sets of records and sets of links that define relationships between different records. Figure 1-2 shows how George's and Sue's same accounts might look in such a system.

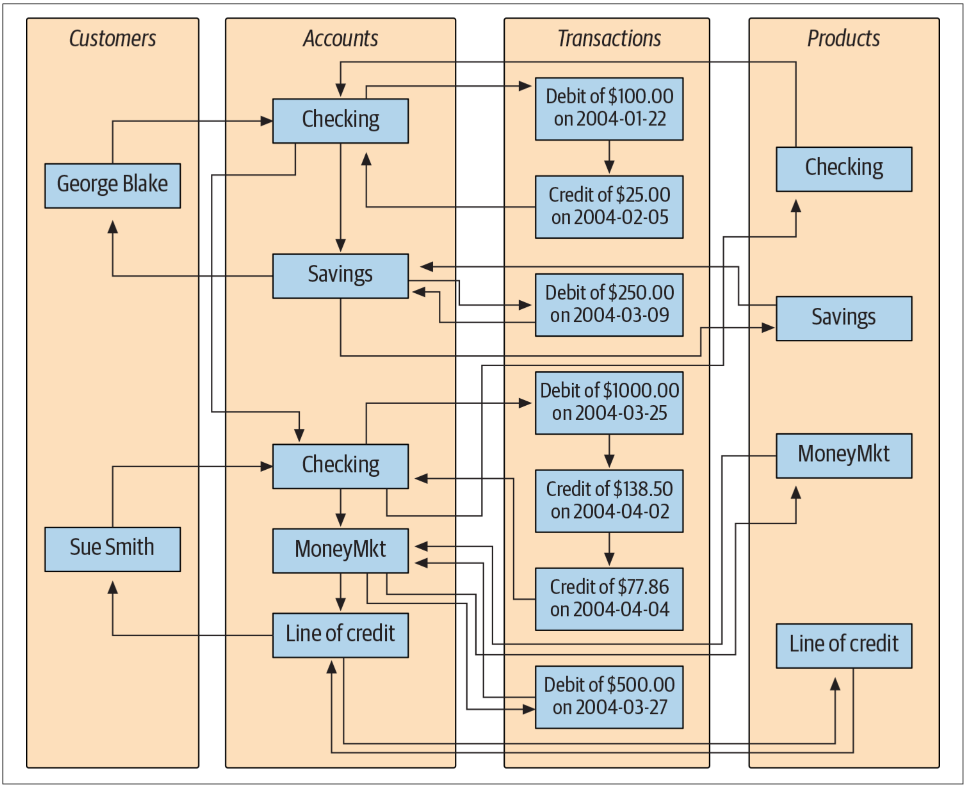

Figure 1-2. Network view of account data

In order to find the transactions posted to Sue's money market account, you would need to perform the following steps:

1. Find the customer record for Sue Smith.
2. Follow the link from Sue Smith's customer record to her list of accounts.
3. Traverse the chain of accounts until you find the money market account.
4. Follow the link from the money market record to its list of transactions.

One interesting feature of network database systems is demonstrated by the set of `product` records on the far right of Figure 1-2. Notice that each `product` record (Checking, Savings, etc.) points to a list of `account` records that are of that product type. `Account` records, therefore, can be accessed from multiple places (both `customer` records and `product` records), allowing a network database to act as a *multiparent hierarchy*.

Both hierarchical and network database systems are alive and well today, although generally in the mainframe world. Additionally, hierarchical database systems have enjoyed a rebirth in the directory services realm, such as Microsoft's Active Directory and the open source Apache Directory Server. Beginning in the 1970s, however, a new way of representing data began to take root, one that was more rigorous yet easy to understand and implement.

The Relational Model

In 1970, Dr. E. F. Codd of IBM's research laboratory published a paper titled "A Relational Model of Data for Large Shared Data Banks" that proposed that data be represented as sets of *tables*. Rather than using pointers to navigate between related entities, redundant data is used to link records in different tables. Figure 1-3 shows how George's and Sue's account information would appear in this context.

Customer

cust_id	*fname*	*lname*
1	George	Blake
2	Sue	Smith

Account

account_id	*product_cd*	*cust_id*	*balance*
103	CHK	1	$75.00
104	SAV	1	$250.00
105	CHK	2	$783.64
106	MM	2	$500.00
107	LOC	2	0

Product

product_cd	*name*
CHK	Checking
SAV	Savings
MM	Money market
LOC	Line of credit

Transaction

txn_id	*txn_type_cd*	*account_id*	*amount*	*date*
978	DBT	103	$100.00	2004-01-22
979	CDT	103	$25.00	2004-02-05
980	DBT	104	$250.00	2004-03-09
981	DBT	105	$1000.00	2004-03-25
982	CDT	105	$138.50	2004-04-02
983	CDT	105	$77.86	2004-04-04
984	DBT	106	$500.00	2004-03-27

Figure 1-3. Relational view of account data

The four tables in Figure 1-3 represent the four entities discussed so far: `customer`, `product`, `account`, and `transaction`. Looking across the top of the `customer` table in Figure 1-3, you can see three *columns*: `cust_id` (which contains the customer's ID number), `fname` (which contains the customer's first name), and `lname` (which contains the customer's last name). Looking down the side of the `customer` table, you can see two *rows*, one containing George Blake's data and the other containing Sue Smith's data. The number of columns that a table may contain differs from server to server, but it is generally large enough not to be an issue (Microsoft SQL Server, for example, allows up to 1,024 columns per table). The number of rows that a table may contain is more a matter of physical limits (i.e., how much disk drive space is available) and maintainability (i.e., how large a table can get before it becomes difficult to work with) than of database server limitations.

Each table in a relational database includes information that uniquely identifies a row in that table (known as the *primary key*), along with additional information needed to describe the entity completely. Looking again at the `customer` table, the `cust_id` column holds a different number for each customer; George Blake, for example, can be uniquely identified by customer ID 1. No other customer will ever be assigned that identifier, and no other information is needed to locate George Blake's data in the `customer` table.

> Every database server provides a mechanism for generating unique sets of numbers to use as primary key values, so you won't need to worry about keeping track of what numbers have been assigned.

While I might have chosen to use the combination of the `fname` and `lname` columns as the primary key (a primary key consisting of two or more columns is known as a *compound key*), there could easily be two or more people with the same first and last names who have accounts at the bank. Therefore, I chose to include the `cust_id` column in the `customer` table specifically for use as a primary key column.

> In this example, choosing `fname`/`lname` as the primary key would be referred to as a *natural key*, whereas the choice of `cust_id` would be referred to as a *surrogate key*. The decision whether to employ natural or surrogate keys is up to the database designer, but in this particular case the choice is clear, since a person's last name may change (such as when a person adopts a spouse's last name), and primary key columns should never be allowed to change once a value has been assigned.

Some of the tables also include information used to navigate to another table; this is where the "redundant data" mentioned earlier comes in. For example, the account table includes a column called cust_id, which contains the unique identifier of the customer who opened the account, along with a column called product_cd, which contains the unique identifier of the product to which the account will conform. These columns are known as *foreign keys*, and they serve the same purpose as the lines that connect the entities in the hierarchical and network versions of the account information. If you are looking at a particular account record and want to know more information about the customer who opened the account, you would take the value of the cust_id column and use it to find the appropriate row in the customer table (this process is known, in relational database lingo, as a *join*; joins are introduced in Chapter 3 and probed deeply in Chapters 5 and 10).

It might seem wasteful to store the same data many times, but the relational model is quite clear on what redundant data may be stored. For example, it is proper for the account table to include a column for the unique identifier of the customer who opened the account, but it is not proper to include the customer's first and last names in the account table as well. If a customer were to change her name, for example, you want to make sure that there is only one place in the database that holds the customer's name; otherwise, the data might be changed in one place but not another, causing the data in the database to be unreliable. The proper place for this data is the customer table, and only the cust_id values should be included in other tables. It is also not proper for a single column to contain multiple pieces of information, such as a name column that contains both a person's first and last names, or an address column that contains street, city, state, and zip code information. The process of refining a database design to ensure that each independent piece of information is in only one place (except for foreign keys) is known as *normalization*.

Getting back to the four tables in Figure 1-3, you may wonder how you would use these tables to find George Blake's transactions against his checking account. First, you would find George Blake's unique identifier in the customer table. Then, you would find the row in the account table whose cust_id column contains George's unique identifier and whose product_cd column matches the row in the product table whose name column equals "Checking." Finally, you would locate the rows in the transaction table whose account_id column matches the unique identifier from the account table. This might sound complicated, but you can do it in a single command, using the SQL language, as you will see shortly.

Some Terminology

I introduced some new terminology in the previous sections, so maybe it's time for some formal definitions. Table 1-1 shows the terms we use for the remainder of the book along with their definitions.

Table 1-1. Terms and definitions

Term	Definition
Entity	Something of interest to the database user community. Examples include customers, parts, geographic locations, etc.
Column	An individual piece of data stored in a table.
Row	A set of columns that together completely describe an entity or some action on an entity. Also called a record.
Table	A set of rows, held either in memory (nonpersistent) or on permanent storage (persistent).
Result set	Another name for a nonpersistent table, generally the result of an SQL query.
Primary key	One or more columns that can be used as a unique identifier for each row in a table.
Foreign key	One or more columns that can be used together to identify a single row in another table.

What Is SQL?

Along with Codd's definition of the relational model, he proposed a language called DSL/Alpha for manipulating the data in relational tables. Shortly after Codd's paper was released, IBM commissioned a group to build a prototype based on Codd's ideas. This group created a simplified version of DSL/Alpha that they called SQUARE. Refinements to SQUARE led to a language called SEQUEL, which was, finally, shortened to SQL. While SQL began as a language used to manipulate data in relational databases, it has evolved (as you will see toward the end of this book) to be a language for manipulating data across various database technologies.

SQL is now more than 40 years old, and it has undergone a great deal of change along the way. In the mid-1980s, the American National Standards Institute (ANSI) began working on the first standard for the SQL language, which was published in 1986. Subsequent refinements led to new releases of the SQL standard in 1989, 1992, 1999, 2003, 2006, 2008, 2011, and 2016. Along with refinements to the core language, new features have been added to the SQL language to incorporate object-oriented functionality, among other things. The later standards focus on the integration of related technologies, such as extensible markup language (XML) and JavaScript object notation (JSON).

SQL goes hand in hand with the relational model because the result of an SQL query is a table (also called, in this context, a *result set*). Thus, a new permanent table can be created in a relational database simply by storing the result set of a query. Similarly, a query can use both permanent tables and the result sets from other queries as inputs (we explore this in detail in Chapter 9).

One final note: SQL is not an acronym for anything (although many people will insist it stands for "Structured Query Language"). When referring to the language, it is equally acceptable to say the letters individually (i.e., S. Q. L.) or to use the word *sequel*.

SQL Statement Classes

The SQL language is divided into several distinct parts: the parts that we explore in this book include *SQL schema statements*, which are used to define the data structures stored in the database; *SQL data statements*, which are used to manipulate the data structures previously defined using SQL schema statements; and *SQL transaction statements*, which are used to begin, end, and roll back transactions (concepts covered in Chapter 12). For example, to create a new table in your database, you would use the SQL schema statement `create table`, whereas the process of populating your new table with data would require the SQL data statement `insert`.

To give you a taste of what these statements look like, here's an SQL schema statement that creates a table called `corporation`:

```
CREATE TABLE corporation
 (corp_id SMALLINT,
  name VARCHAR(30),
  CONSTRAINT pk_corporation PRIMARY KEY (corp_id)
 );
```

This statement creates a table with two columns, `corp_id` and `name`, with the `corp_id` column identified as the primary key for the table. We probe the finer details of this statement, such as the different data types available with MySQL, in Chapter 2. Next, here's an SQL data statement that inserts a row into the `corporation` table for Acme Paper Corporation:

```
INSERT INTO corporation (corp_id, name)
VALUES (27, 'Acme Paper Corporation');
```

This statement adds a row to the `corporation` table with a value of `27` for the `corp_id` column and a value of `Acme Paper Corporation` for the `name` column.

Finally, here's a simple `select` statement to retrieve the data that was just created:

```
mysql< SELECT name
    -> FROM corporation
    -> WHERE corp_id = 27;
+------------------------+
| name                   |
+------------------------+
| Acme Paper Corporation |
+------------------------+
```

All database elements created via SQL schema statements are stored in a special set of tables called the *data dictionary*. This "data about the database" is known collectively as *metadata* and is explored in Chapter 15. Just like tables that you create yourself, data dictionary tables can be queried via a `select` statement, thereby allowing you to discover the current data structures deployed in the database at runtime. For example, if you are asked to write a report showing the new accounts created last month, you could either hardcode the names of the columns in the `account` table that were known to you when you wrote the report, or query the data dictionary to determine the current set of columns and dynamically generate the report each time it is executed.

Most of this book is concerned with the data portion of the SQL language, which consists of the `select`, `update`, `insert`, and `delete` commands. SQL schema statements are demonstrated in Chapter 2, which will lead you through the design and creation of some simple tables. In general, SQL schema statements do not require much discussion apart from their syntax, whereas SQL data statements, while few in number, offer numerous opportunities for detailed study. Therefore, while I try to introduce you to many of the SQL schema statements, most chapters in this book concentrate on the SQL data statements.

SQL: A Nonprocedural Language

If you have worked with programming languages in the past, you are used to defining variables and data structures, using conditional logic (i.e., if-then-else) and looping constructs (i.e., do while ... end), and breaking your code into small, reusable pieces (i.e., objects, functions, procedures). Your code is handed to a compiler, and the executable that results does exactly (well, not always *exactly*) what you programmed it to do. Whether you work with Java, Python, Scala, or some other *procedural* language, you are in complete control of what the program does.

> A procedural language defines both the desired results and the mechanism, or process, by which the results are generated. Nonprocedural languages also define the desired results, but the process by which the results are generated is left to an external agent.

With SQL, however, you will need to give up some of the control you are used to, because SQL statements define the necessary inputs and outputs, but the manner in which a statement is executed is left to a component of your database engine known as the *optimizer*. The optimizer's job is to look at your SQL statements and, taking into account how your tables are configured and what indexes are available, decide the most efficient execution path (well, not always the *most* efficient). Most database engines will allow you to influence the optimizer's decisions by specifying *optimizer hints*, such as suggesting that a particular index be used; most SQL users, however, will never get to this level of sophistication and will leave such tweaking to their database administrator or performance expert.

Therefore, with SQL, you will not be able to write complete applications. Unless you are writing a simple script to manipulate certain data, you will need to integrate SQL with your favorite programming language. Some database vendors have done this for you, such as Oracle's PL/SQL language, MySQL's stored procedure language, and Microsoft's Transact-SQL language. With these languages, the SQL data statements are part of the language's grammar, allowing you to seamlessly integrate database queries with procedural commands. If you are using a non-database-specific language such as Java or Python, however, you will need to use a toolkit/API to execute SQL statements from your code. Some of these toolkits are provided by your database vendor, whereas others have been created by third-party vendors or by open source providers. Table 1-2 shows some of the available options for integrating SQL into a specific language.

Table 1-2. SQL integration toolkits

Language	Toolkit
Java	JDBC (Java Database Connectivity)
C#	ADO.NET (Microsoft)
Ruby	Ruby DBI
Python	Python DB
Go	Package database/sql

If you only need to execute SQL commands interactively, every database vendor provides at least a simple command-line tool for submitting SQL commands to the database engine and inspecting the results. Most vendors provide a graphical tool as well that includes one window showing your SQL commands and another window showing the results from your SQL commands. Additionally, there are third-party tools such as SQuirrel, which will connect via a JDBC connection to many different database servers. Since the examples in this book are executed against a MySQL database, I use the `mysql` command-line tool that is included as part of the MySQL installation to run the examples and format the results.

SQL Examples

Earlier in this chapter, I promised to show you an SQL statement that would return all the transactions against George Blake's checking account. Without further ado, here it is:

```
SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM individual i
  INNER JOIN account a ON i.cust_id = a.cust_id
  INNER JOIN product p ON p.product_cd = a.product_cd
  INNER JOIN transaction t ON t.account_id = a.account_id
WHERE i.fname = 'George' AND i.lname = 'Blake'
  AND p.name = 'checking account';

+--------+-------------+---------------------+--------+
| txn_id | txn_type_cd | txn_date            | amount |
```

```
+--------+------------+---------------------+--------+
|     11 | DBT        | 2008-01-05 00:00:00 | 100.00 |
+--------+------------+---------------------+--------+
1 row in set (0.00 sec)
```

Without going into too much detail at this point, this query identifies the row in the `individual` table for George Blake and the row in the `product` table for the "checking" product, finds the row in the `account` table for this individual/product combination, and returns four columns from the `transaction` table for all transactions posted to this account. If you happen to know that George Blake's customer ID is 8 and that checking accounts are designated by the code `'CHK'`, then you can simply find George Blake's checking account in the `account` table based on the customer ID and use the account ID to find the appropriate transactions:

```
SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM account a
  INNER JOIN transaction t ON t.account_id = a.account_id
WHERE a.cust_id = 8 AND a.product_cd = 'CHK';
```

I cover all of the concepts in these queries (plus a lot more) in the following chapters, but I wanted to at least show what they would look like.

The previous queries contain three different *clauses*: `select`, `from`, and `where`. Almost every query that you encounter will include at least these three clauses, although there are several more that can be used for more specialized purposes. The role of each of these three clauses is demonstrated by the following:

```
SELECT /* one or more things */ ...
FROM /* one or more places */ ...
WHERE /* one or more conditions apply */ ...
```

> Most SQL implementations treat any text between the `/*` and `*/` tags as comments.

When constructing your query, your first task is generally to determine which table or tables will be needed and then add them to your `from` clause. Next, you will need to add conditions to your `where` clause to filter out the data from these tables that you aren't interested in. Finally, you will decide which columns from the different tables need to be retrieved and add them to your `select` clause. Here's a simple example that shows how you would find all customers with the last name "Smith":

```
SELECT cust_id, fname
FROM individual
WHERE lname = 'Smith';
```

This query searches the `individual` table for all rows whose `lname` column matches the string `'Smith'` and returns the `cust_id` and `fname` columns from those rows.

Along with querying your database, you will most likely be involved with populating and modifying the data in your database. Here's a simple example of how you would insert a new row into the `product` table:

```
INSERT INTO product (product_cd, name)
VALUES ('CD', 'Certificate of Depysit')
```

Whoops, looks like you misspelled "Deposit." No problem. You can clean that up with an `update` statement:

```
UPDATE product
SET name = 'Certificate of Deposit'
WHERE product_cd = 'CD';
```

Notice that the `update` statement also contains a `where` clause, just like the `select` statement. This is because an `update` statement must identify the rows to be modified; in this case, you are specifying that only those rows whose `product_cd` column matches the string `'CD'` should be modified. Since the `product_cd` column is the primary key for the `product` table, you should expect your `update` statement to modify exactly one row (or zero, if the value doesn't exist in the table). Whenever you execute an SQL data statement, you will receive feedback from the database engine as to how many rows were affected by your statement. If you are using an interactive tool such as the `mysql` command-line tool mentioned earlier, then you will receive feedback concerning how many rows were either:

- Returned by your `select` statement
- Created by your `insert` statement
- Modified by your `update` statement
- Removed by your `delete` statement

If you are using a procedural language with one of the toolkits mentioned earlier, the toolkit will include a call to ask for this information after your SQL data statement has executed. In general, it's a good idea to check this info to make sure your statement didn't do something unexpected (like when you forget to put a `where` clause on your `delete` statement and delete every row in the table!).

What Is MySQL?

Relational databases have been available commercially for more than three decades. Some of the most mature and popular commercial products include:

- Oracle Database from Oracle Corporation
- SQL Server from Microsoft
- DB2 Universal Database from IBM

All these database servers do approximately the same thing, although some are better equipped to run very large or very high throughput databases. Others are better at handling objects or very large files or XML documents, and so on. Additionally, all these servers do a pretty good job of complying with the latest ANSI SQL standard. This is a good thing, and I make it a point to show you how to write SQL statements that will run on any of these platforms with little or no modification.

Along with the commercial database servers, there has been quite a bit of activity in the open source community in the past two decades with the goal of creating a viable alternative. Two of the most commonly used open source database servers are PostgreSQL and MySQL. The MySQL server is available for free, and I have found it to be extremely simple to download and install. For these reasons, I have decided that all examples for this book be run against a MySQL (version 8.0) database, and that the `mysql` command-line tool be used to format query results. Even if you are already using another server and never plan to use MySQL, I urge you to install the latest MySQL server, load the sample schema and data, and experiment with the data and examples in this book.

However, keep in mind the following caveat:

> This is not a book about MySQL's SQL implementation.

Rather, this book is designed to teach you how to craft SQL statements that will run on MySQL with no modifications, and will run on recent releases of Oracle Database, DB2, and SQL Server with few or no modifications.

SQL Unplugged

A great deal has happened in the database world during the decade between the second and third editions of this book. While relational databases are still heavily used and will continue to be for some time, new database technologies have emerged to meet the needs of companies like Amazon and Google. These technologies include Hadoop, Spark, NoSQL, and NewSQL, which are distributed, scalable systems typically deployed on clusters of commodity servers. While it is beyond the scope of this book to explore these technologies in detail, they do all share something in common with relational databases: SQL.

Since organizations frequently store data using multiple technologies, there is a need to unplug SQL from a particular database server and provide a service that can span multiple databases. For example, a report may need to bring together data stored in

Oracle, Hadoop, JSON files, CSV files, and Unix log files. A new generation of tools have been built to meet this type of challenge, and one of the most promising is Apache Drill, which is an open source query engine that allows users to write queries that can access data stored in most any database or filesystem. We will explore Apache Drill in Chapter 18.

What's in Store

The overall goal of the next four chapters is to introduce the SQL data statements, with a special emphasis on the three main clauses of the `select` statement. Additionally, you will see many examples that use the Sakila schema (introduced in the next chapter), which will be used for all examples in the book. It is my hope that familiarity with a single database will allow you to get to the crux of an example without having to stop and examine the tables being used each time. If it becomes a bit tedious working with the same set of tables, feel free to augment the sample database with additional tables or to invent your own database with which to experiment.

After you have a solid grasp on the basics, the remaining chapters will drill deep into additional concepts, most of which are independent of each other. Thus, if you find yourself getting confused, you can always move ahead and come back later to revisit a chapter. When you have finished the book and worked through all of the examples, you will be well on your way to becoming a seasoned SQL practitioner.

For readers interested in learning more about relational databases, the history of computerized database systems, or the SQL language than was covered in this short introduction, here are a few resources worth checking out:

- *Database in Depth: Relational Theory for Practitioners* by C. J. Date (O'Reilly)
- *An Introduction to Database Systems*, Eighth Edition, by C. J. Date (Addison-Wesley)
- *The Database Relational Model: A Retrospective Review and Analysis*, by C. J. Date (Addison-Wesley)
- Wikipedia subarticle on definition of "Database Management System"

CHAPTER 2

Creating and Populating a Database

This chapter provides you with the information you need to create your first database and to create the tables and associated data used for the examples in this book. You will also learn about various data types and see how to create tables using them. Because the examples in this book are executed against a MySQL database, this chapter is somewhat skewed toward MySQL's features and syntax, but most concepts are applicable to any server.

Creating a MySQL Database

If you want the ability to experiment with the data used for the examples in this book, you have two options:

- Download and install the MySQL server version 8.0 (or later) and load the Sakila example database from *https://dev.mysql.com/doc/index-other.html*.
- Go to *https://www.katacoda.com/mysql-db-sandbox/scenarios/mysql-sandbox* to access the MySQL Sandbox, which has the Sakila sample database loaded in a MySQL instance. You'll have to set up a (free) Katacoda account. Then, click the Start Scenario button.

If you choose the second option, once you start the scenario, a MySQL server is installed and started, and then the Sakila schema and data are loaded. When it's ready, a standard `mysql>` prompt appears, and you can then start querying the sample database. This is certainly the easiest option, and I anticipate that most readers will choose this option; if this sounds good to you, feel free to skip ahead to the next section.

If you prefer to have your own copy of the data and want any changes you have made to be permanent, or if you are just interested in installing the MySQL server on your