Learning SQL

O'REILLY®

Alan Beaulieu

O'REILLY®

Learning SQL

Updated for the latest database management systems—including MySQL 6.0, Oracle 11g, and Microsoft's SQL Server 2008—this introductory guide will get you up and running with SQL quickly. Whether you need to write database applications, perform administrative tasks, or generate reports, *Learning SQL*, Second Edition, will help you easily master all the SQL fundamentals.

Each chapter presents a self-contained lesson on a key SQL concept or technique, with numerous illustrations and annotated examples. Exercises at the end of each chapter let you practice the skills you learn.

With this book, you will:

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

Knowledge of SQL is a must for interacting with data. With *Learning SQL*, you'll quickly learn how to put the power and flexibility of this language to work.

"If you have decided to learn SQL, then roll up your sleeves and let Learning SQL *be your partner. Reading the well-organized chapters and completing along the way each set of practical exercises will prepare you for creating your own 'databased' solutions. Databases are everywhere—with this book you can take advantage of Mr. Beaulieu's proven approach to working with SQL."*

—Roy Owens,
Database Developer,
CBORD Group, Inc.

Alan Beaulieu has been designing, building, and implementing custom database applications for over 15 years. He runs his own consulting company, which specializes in designing Oracle databases and supporting services in the financial services and telecommunications fields. Alan is a graduate of the Cornell University School of Engineering.

www.oreilly.com

US $39.99 **CAN $49.99**

ISBN: 978-0-596-52083-0

SECOND EDITION

Learning SQL

Alan Beaulieu

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Learning SQL, Second Edition
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://safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mary E. Treseler
Production Editor: Loranah Dimant
Copyeditor: Audrey Doyle
Proofreader: Nancy Reinhardt

Indexer: Ellen Troutman Zaig
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
August 2005: First Edition.
April 2009: Second Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-52083-0

[M]

1239115419

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, and C, which is still quite popular for operating system 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 is the language for generating, manipulating, and retrieving data from a relational database. One of the reasons for the popularity of relational databases is that properly designed relational databases can handle huge amounts of data. When working with large data sets, SQL is akin to one of those snazzy digital cameras with the high-power zoom lens in that you can use SQL to look at large sets of data, or you can zoom in on individual rows (or anywhere in between). Other database management systems tend to break down under heavy loads because their focus is too narrow (the zoom lens is stuck on maximum), which is why attempts to dethrone relational databases and SQL have largely failed. Therefore, even though SQL is an old language, it is going to be around for a lot longer and has a bright future in store.

Why Learn SQL?

If you are going to work with a relational database, whether you are writing applications, performing administrative tasks, or generating reports, you will need to know how to interact with the data in your database. Even if you are using a tool that generates SQL for you, such as a reporting tool, there may be times when you need to bypass the automatic generation feature and write your own SQL statements.

Learning SQL has the added benefit of forcing you to confront and understand the data structures used to store information about your organization. As you become comfortable with the tables in your database, you may find yourself proposing modifications or additions to your database schema.

Why Use This Book to Do It?

The SQL language is broken into several categories. Statements used to create database objects (tables, indexes, constraints, etc.) are collectively known as SQL *schema statements*. The statements used to create, manipulate, and retrieve the data stored in a database are known as the SQL *data statements*. If you are an administrator, you will be using both SQL schema and SQL data statements. If you are a programmer or report writer, you may only need to use (or be *allowed* to use) SQL data statements. While this book demonstrates many of the SQL schema statements, the main focus of this book is on programming features.

With only a handful of commands, the SQL data statements look deceptively simple. In my opinion, many of the available SQL books help to foster this notion by only skimming the surface of what is possible with the language. However, if you are going to work with SQL, it behooves you to understand fully the capabilities of the language and how different features can be combined to produce powerful results. I feel that this is the only book that provides detailed coverage of the SQL language without the added benefit of doubling as a "door stop" (you know, those 1,250-page "complete references" that tend to gather dust on people's cubicle shelves).

While the examples in this book run on MySQL, Oracle Database, and SQL Server, I had to pick one of those products to host my sample database and to format the result sets returned by the example queries. Of the three, I chose MySQL because it is freely obtainable, easy to install, and simple to administer. For those readers using a different server, I ask that you download and install MySQL and load the sample database so that you can run the examples and experiment with the data.

Structure of This Book

This book is divided into 15 chapters and 3 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*, is all about data sets and how they can interact within queries.

Chapter 7, *Data Generation, Conversion, and Manipulation*, 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 the subquery (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.

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

Appendix B, *MySQL Extensions to the SQL Language*, demonstrates some of the interesting non-ANSI features of MySQL's SQL implementation.

Appendix C, *Solutions to Exercises*, shows solutions to the chapter exercises.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Used for filenames, directory names, and URLs. Also used for emphasis and to indicate the first use of a technical term.

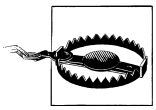
`Constant width`
: Used for code examples and to indicate SQL keywords within text.

`Constant width italic`
: Used to indicate user-defined terms.

UPPERCASE
: Used to indicate SQL keywords within example code.

`Constant width bold`
: Indicates user input in examples showing an interaction. Also indicates emphasized code elements to which you should pay particular attention.

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.

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)

O'Reilly maintains a web page for this book, which lists errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596520830

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about O'Reilly books, conferences, Resource Centers, and the O'Reilly Network, see the website at:

http://www.oreilly.com

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example, "*Learning SQL*, Second Edition, by Alan Beaulieu. Copyright 2009 O'Reilly Media, Inc., 978-0-596-52083-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

Acknowledgments

I would like to thank my editor, Mary Treseler, for helping to make this second edition a reality, and many thanks to Kevin Kline, Roy Owens, Richard Sonen, and Matthew Russell, who were kind enough to review the book for me over the Christmas/New Year holidays. I would also like to thank the many readers of my first edition who were kind enough to send questions, comments, and corrections. 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 might be beneficial to introduce some basic database concepts and look at the history of computerized data storage and retrieval.

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 terabytes of data spread across many fast-access disk drives, holding 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.

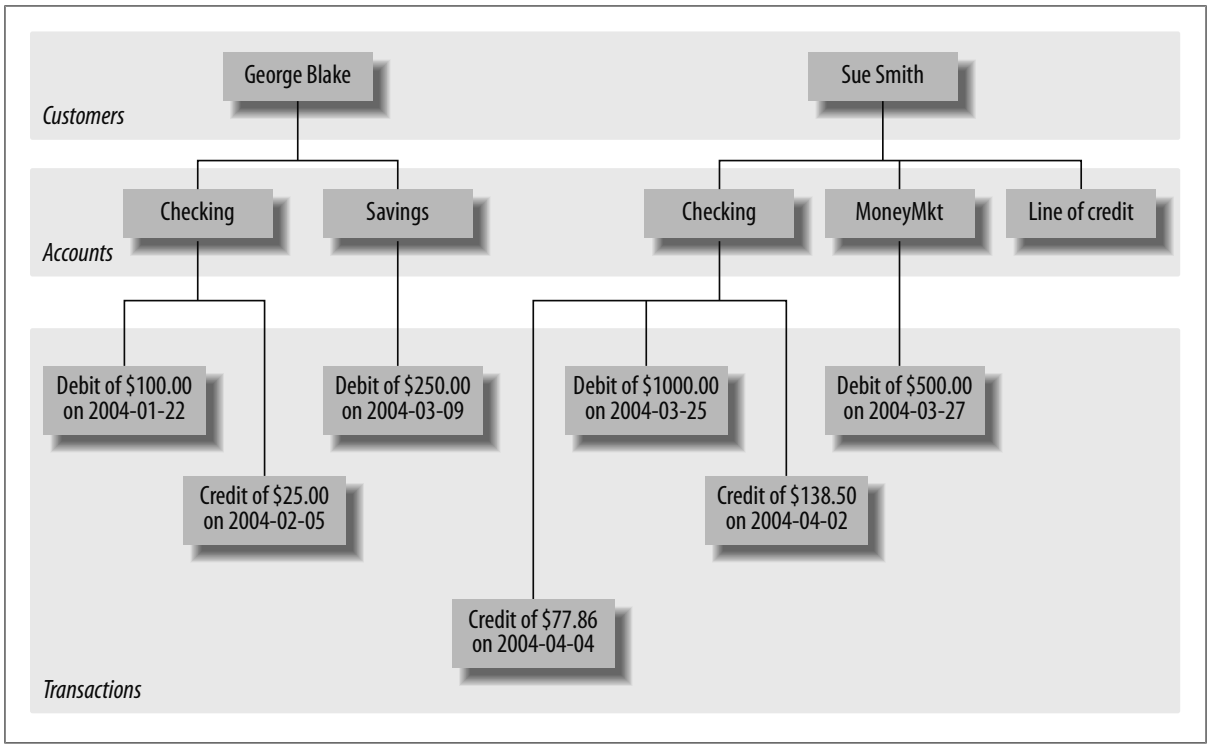


Figure 1-1. Hierarchical view of account data

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.

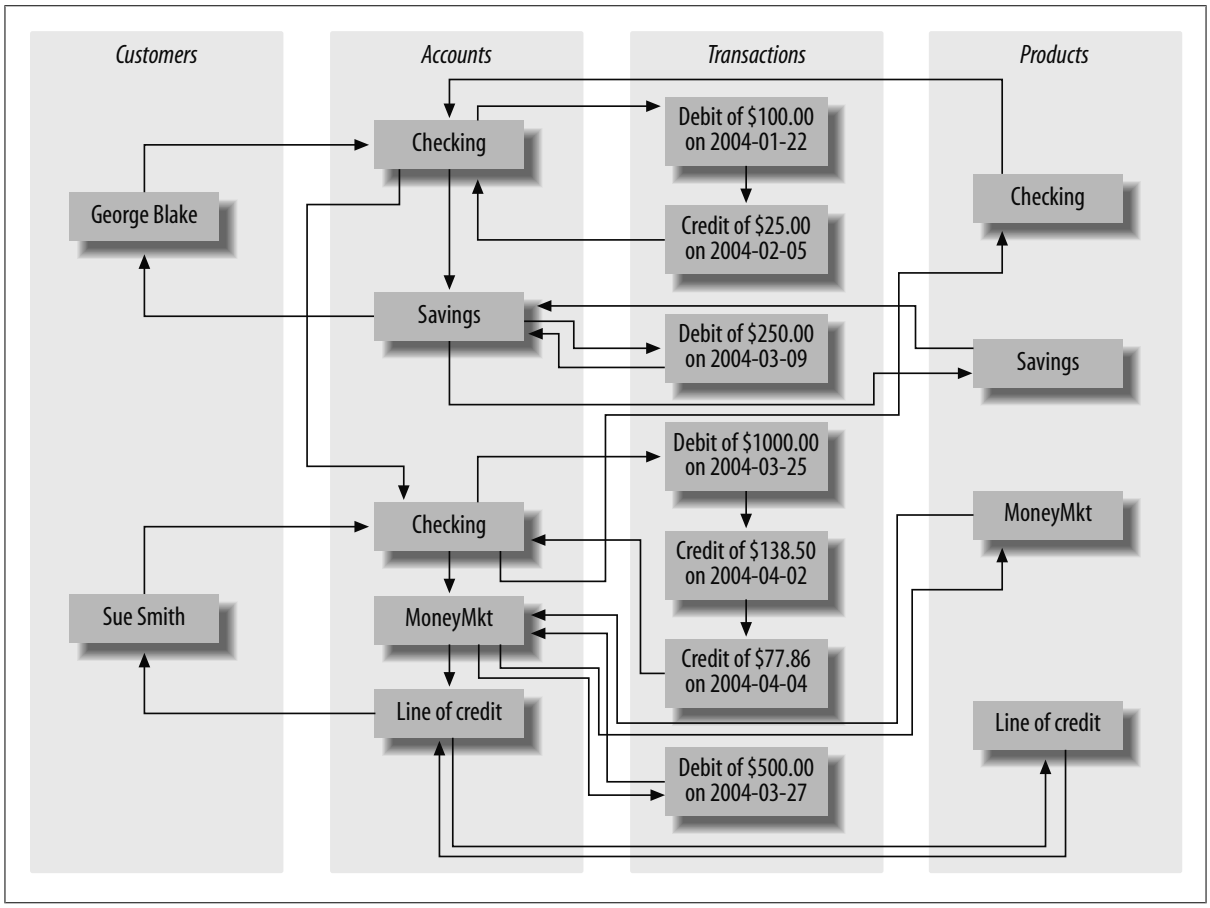


Figure 1-2. Network view of account data

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

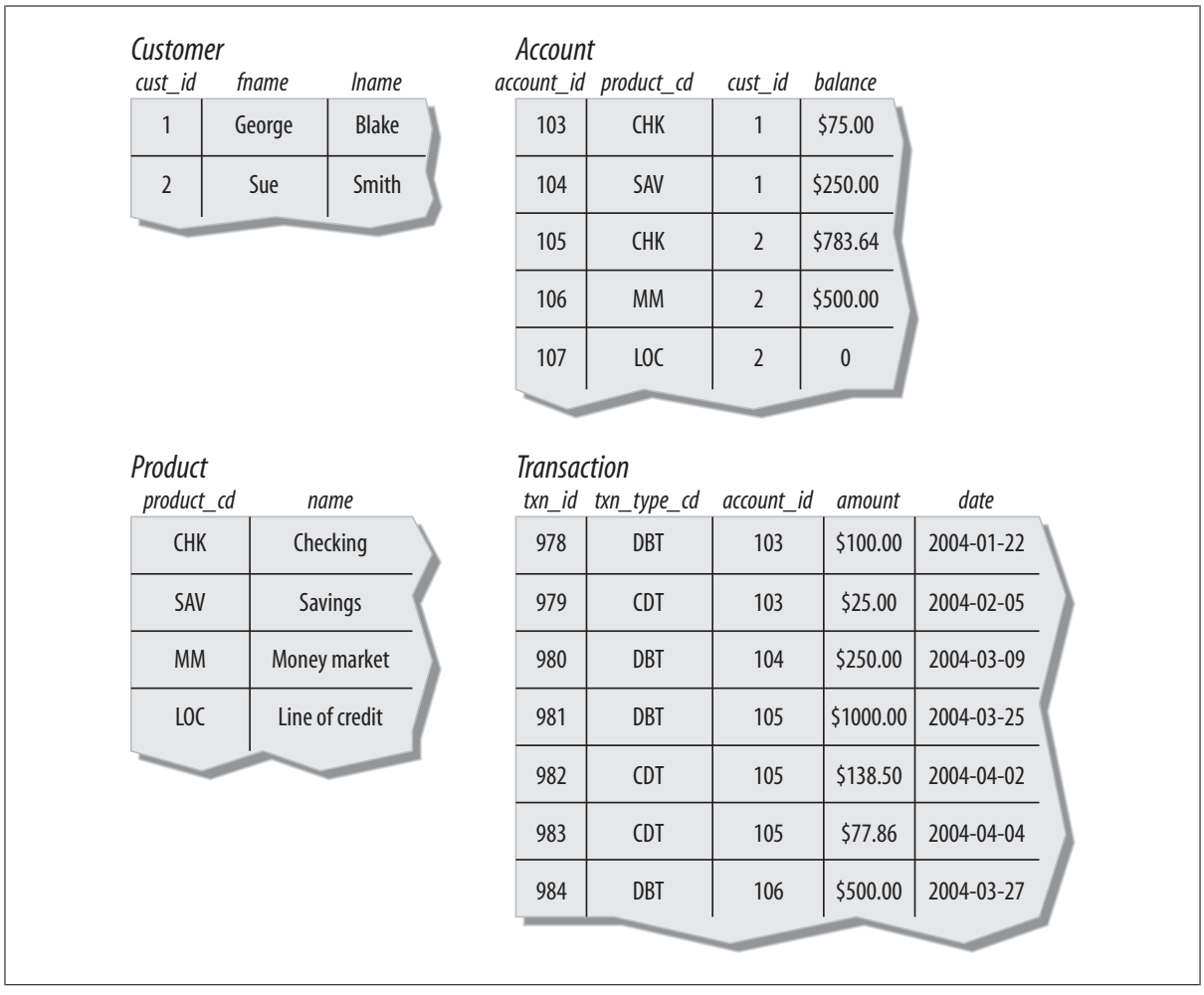
1. Find the customer record for Sue Smith.
2. Follow the link from Sue Smith's customer record to her list of accounts.
3. Traverse the chain of accounts until you find the money market account.
4. Follow the link from the money market record to its list of transactions.

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 Red Hat Directory Server, as well as with Extensible Markup Language (XML). 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

There are four tables in Figure 1-3 representing 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 that 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 a topic of widespread debate, 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).

Term	Definition
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, renamed SQL.

SQL is now entering middle age (as is this author, alas), 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, and 2006. 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 latest standard, SQL:2006, focuses on the integration of SQL and XML and defines a language called XQuery which is used to query data in XML documents.

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 (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 is demonstrated in Chapter 2, where the sample database used throughout this book is generated. 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, C#, C, Visual Basic, 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.

With SQL, therefore, 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, 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 are 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; JavaSoft)
C++	Rogue Wave SourcePro DB (third-party tool to connect to Oracle, SQL Server, MySQL, Informix, DB2, Sybase, and PostgreSQL databases)
C/C++	Pro*C (Oracle), MySQL C API (open source), and DB2 Call Level Interface (IBM)
C#	ADO.NET (Microsoft)
Perl	Perl DBI
Python	Python DB
Visual Basic	ADO.NET (Microsoft)

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. 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 over two 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
- Sybase Adaptive Server from Sybase

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 five years with the goal of creating a viable alternative to the commercial database servers. Two of the most commonly used open source database servers are PostgreSQL and MySQL. The MySQL website (*http://www.mysql.com*) currently claims over 10 million installations, its server is available for free,

and I have found its server 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 6.0) 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, Sybase Adaptive Server, and SQL Server with few or no modifications.

To keep the code in this book as vendor-independent as possible, I will refrain from demonstrating some of the interesting things that the MySQL SQL language implementers have decided to do that can't be done on other database implementations. Instead, Appendix B covers some of these features for readers who are planning to continue using MySQL.

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 bank 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 your 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 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:

- C.J. Date's *Database in Depth: Relational Theory for Practitioners (http://oreilly.com/catalog/9780596100124/)* (O'Reilly)
- C.J. Date's *An Introduction to Database Systems*, Eighth Edition (Addison-Wesley)

- C.J. Date's *The Database Relational Model: A Retrospective Review and Analysis: A Historical Account and Assessment of E. F. Codd's Contribution to the Field of Database Technology* (Addison-Wesley)
- *http://en.wikipedia.org/wiki/Database_management_system*
- *http://www.mcjones.org/System_R/*

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

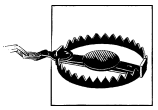
If you already have a MySQL database server available for your use, you can skip the installation instructions and start with the instructions in Table 2-1. Keep in mind, however, that this book assumes that you are using MySQL version 6.0 or later, so you may want to consider upgrading your server or installing another server if you are using an earlier release.

The following instructions show you the minimum steps required to install a MySQL 6.0 server on a Windows computer:

1. Go to the download page for the MySQL Database Server at *http://dev.mysql.com/downloads*. If you are loading version 6.0, the full URL is *http://dev.mysql.com/downloads/mysql/6.0.html*.
2. Download the Windows Essentials (x86) package, which includes only the commonly used tools.
3. When asked "Do you want to run or save this file?" click Run.
4. The MySQL Server 6.0—Setup Wizard window appears. Click Next.
5. Activate the Typical Install radio button, and click Next.
6. Click Install.
7. A MySQL Enterprise window appears. Click Next twice.

8. When the installation is complete, make sure the box is checked next to "Configure the MySQL Server now," and then click Finish. This launches the Configuration Wizard.
9. When the Configuration Wizard launches, activate the Standard Configuration radio button, and then select both the "Install as Windows Service" and "Include Bin Directory in Windows Path" checkboxes. Click Next.
10. Select the Modify Security Settings checkbox and enter a password for the `root` user (make sure you write down the password, because you will need it shortly!), and click Next.
11. Click Execute.

At this point, if all went well, the MySQL server is installed and running. If not, I suggest you uninstall the server and read the "Troubleshooting a MySQL Installation Under Windows" guide (which you can find at *http://dev.mysql.com/doc/refman/6.0/en/windows-troubleshooting.html*).



If you uninstalled an older version of MySQL before loading version 6.0, you may have some further cleanup to do (I had to clean out some old Registry entries) before you can get the Configuration Wizard to run successfully.

Next, you will need to open a Windows command window, launch the `mysql` tool, and create your database and database user. Table 2-1 describes the necessary steps. In step 5, feel free to choose your own password for the `lrngsql` user rather than "xyz" (but don't forget to write it down!).

Table 2-1. Creating the sample database

Step	Description	Action
1	Open the Run dialog box from the Start menu	Choose Start and then Run
2	Launch a command window	Type **cmd** and click OK
3	Log in to MySQL as `root`	`mysql -u root -p`
4	Create a database for the sample data	`create database bank;`
5	Create the `lrngsql` database user with full privileges on the `bank` database	`grant all privileges on bank.* to 'lrngsql'@'localhost' identified by 'xyz';`
6	Exit the `mysql` tool	`quit;`
7	Log in to MySQL as `lrngsql`	`mysql -u lrngsql -p;`
8	Attach to the `bank` database	`use bank;`

You now have a MySQL server, a database, and a database user; the only thing left to do is create the database tables and populate them with sample data. To do so, download the script at *http://examples.oreilly.com/learningsql/* and run it from the `mysql` utility. If you saved the file as *c:\temp\LearningSQLExample.sql*, you would need to do the following:

1. If you have logged out of the `mysql` tool, repeat steps 7 and 8 from Table 2-1.
2. Type **`source c:\temp\LearningSQLExample.sql;`** and press Enter.

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

Using the mysql Command-Line Tool

Whenever you invoke the `mysql` command-line tool, you can specify the username and database to use, as in the following:

```
mysql -u lrngsql -p bank
```

This will save you from having to type `use bank;` every time you start up the tool. You will be asked for your password, and then the `mysql>` prompt will appear, via which you will be able to issue SQL statements and view the results. For example, if you want to know the current date and time, you could issue the following query:

```
mysql> SELECT now();
+---------------------+
| now()               |
+---------------------+
| 2008-02-19 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

The `now()` function is a built-in MySQL function that returns the current date and time. As you can see, the `mysql` command-line tool formats the results of your queries within a rectangle bounded by +, -, and | characters. After the results have been exhausted (in this case, there is only a single row of results), the `mysql` command-line tool shows how many rows were returned and how long the SQL statement took to execute.

About Missing from Clauses

With some database servers, you won't be able to issue a query without a `from` clause that names at least one table. Oracle Database is a commonly used server for which this is true. For cases when you only need to call a function, Oracle provides a table called `dual`, which consists of a single column called `dummy` that contains a single row of data. In order to be compatible with Oracle Database, MySQL also provides a `dual` table. The previous query to determine the current date and time could therefore be written as:

```
mysql> SELECT now()
       FROM dual;
+---------------------+
| now()               |
+---------------------+
| 2005-05-06 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

If you are not using Oracle and have no need to be compatible with Oracle, you can ignore the `dual` table altogether and use just a `select` clause without a `from` clause.

When you are done with the `mysql` command-line tool, simply type **`quit;`** or **`exit;`** to return to the Windows command shell.

MySQL Data Types

In general, all the popular database servers have the capacity to store the same types of data, such as strings, dates, and numbers. Where they typically differ is in the specialty data types, such as XML documents or very large text or binary documents. Since this is an introductory book on SQL, and since 98% of the columns you encounter will be simple data types, this book covers only the character, date, and numeric data types.

Character Data

Character data can be stored as either fixed-length or variable-length strings; the difference is that fixed-length strings are right-padded with spaces and always consume the same number of bytes, and variable-length strings are not right-padded with spaces and don't always consume the same number of bytes. When defining a character column, you must specify the maximum size of any string to be stored in the column. For example, if you want to store strings up to 20 characters in length, you could use either of the following definitions:

```
char(20)    /* fixed-length */
varchar(20) /* variable-length */
```

The maximum length for `char` columns is currently 255 bytes, whereas `varchar` columns can be up to 65,535 bytes. If you need to store longer strings (such as emails, XML

documents, etc.), then you will want to use one of the text types (`mediumtext` and `longtext`), which I cover later in this section. In general, you should use the `char` type when all strings to be stored in the column are of the same length, such as state abbreviations, and the `varchar` type when strings to be stored in the column are of varying lengths. Both `char` and `varchar` are used in a similar fashion in all the major database servers.

Oracle Database is an exception when it comes to the use of `varchar`. Oracle users should use the `varchar2` type when defining variable-length character columns.

Character sets

For languages that use the Latin alphabet, such as English, there is a sufficiently small number of characters such that only a single byte is needed to store each character. Other languages, such as Japanese and Korean, contain large numbers of characters, thus requiring multiple bytes of storage for each character. Such character sets are therefore called *multibyte character sets*.

MySQL can store data using various character sets, both single- and multibyte. To view the supported character sets in your server, you can use the `show` command, as in:

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | cp1252 West European        | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |      1 |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |      1 |
| ascii    | US ASCII                    | ascii_general_ci    |      1 |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |      3 |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |      2 |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |      1 |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |      1 |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |      2 |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |      1 |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |      2 |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |      1 |
| cp1250   | Windows Central European    | cp1250_general_ci   |      1 |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |      2 |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |      1 |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci |      1 |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
| cp866    | DOS Russian                 | cp866_general_ci    |      1 |
```

```
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |      1 |
| macce    | Mac Central European        | macce_general_ci    |      1 |
| macroman | Mac West European           | macroman_general_ci |      1 |
| cp852    | DOS Central European        | cp852_general_ci    |      1 |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |      1 |
| cp1251   | Windows Cyrillic            | cp1251_general_ci   |      1 |
| cp1256   | Windows Arabic              | cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic              | cp1257_general_ci   |      1 |
| binary   | Binary pseudo charset       | binary              |      1 |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |      1 |
| cp932    | SJIS for Windows Japanese   | cp932_japanese_ci   |      2 |
| eucjpms  | UJIS for Windows Japanese   | eucjpms_japanese_ci |      3 |
+----------+-----------------------------+---------------------+--------+
36 rows in set (0.11 sec)
```

If the value in the fourth column, `maxlen`, is greater than 1, then the character set is a multibyte character set.

When I installed the MySQL server, the `latin1` character set was automatically chosen as the default character set. However, you may choose to use a different character set for each character column in your database, and you can even store different character sets within the same table. To choose a character set other than the default when defining a column, simply name one of the supported character sets after the type definition, as in:

```
varchar(20) character set utf8
```

With MySQL, you may also set the default character set for your entire database:

```
create database foreign_sales character set utf8;
```

While this is as much information regarding character sets as I'm willing to discuss in an introductory book, there is a great deal more to the topic of internationalization than what is shown here. If you plan to deal with multiple or unfamiliar character sets, you may want to pick up a book such as Andy Deitsch and David Czarnecki's *Java Internationalization (http://oreilly.com/catalog/9780596000196/)* (O'Reilly) or Richard Gillam's *Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard* (Addison-Wesley).

Text data

If you need to store data that might exceed the 64 KB limit for `varchar` columns, you will need to use one of the text types.

Table 2-2 shows the available text types and their maximum sizes.

Table 2-2. MySQL text types

Text type	Maximum number of bytes
`Tinytext`	255
`Text`	65,535
`Mediumtext`	16,777,215
`Longtext`	4,294,967,295

When choosing to use one of the text types, you should be aware of the following:

- If the data being loaded into a text column exceeds the maximum size for that type, the data will be truncated.
- Trailing spaces will not be removed when data is loaded into the column.
- When using `text` columns for sorting or grouping, only the first 1,024 bytes are used, although this limit may be increased if necessary.
- The different text types are unique to MySQL. SQL Server has a single `text` type for large character data, whereas DB2 and Oracle use a data type called `clob`, for Character Large Object.
- Now that MySQL allows up to 65,535 bytes for `varchar` columns (it was limited to 255 bytes in version 4), there isn't any particular need to use the `tinytext` or `text` type.

If you are creating a column for free-form data entry, such as a `notes` column to hold data about customer interactions with your company's customer service department, then `varchar` will probably be adequate. If you are storing documents, however, you should choose either the `mediumtext` or `longtext` type.

Oracle Database allows up to 2,000 bytes for `char` columns and 4,000 bytes for `varchar2` columns. SQL Server can handle up to 8,000 bytes for both `char` and `varchar` data.

Numeric Data

Although it might seem reasonable to have a single numeric data type called "numeric," there are actually several different numeric data types that reflect the various ways in which numbers are used, as illustrated here:

A column indicating whether a customer order has been shipped
: This type of column, referred to as a *Boolean*, would contain a `0` to indicate `false` and a `1` to indicate `true`.

A system-generated primary key for a transaction table
: This data would generally start at `1` and increase in increments of one up to a potentially very large number.

An item number for a customer's electronic shopping basket
: The values for this type of column would be positive whole numbers between 1 and, at most, 200 (for shopaholics).

Positional data for a circuit board drill machine
: High-precision scientific or manufacturing data often requires accuracy to eight decimal points.

To handle these types of data (and more), MySQL has several different numeric data types. The most commonly used numeric types are those used to store whole numbers. When specifying one of these types, you may also specify that the data is *unsigned*, which tells the server that all data stored in the column will be greater than or equal to zero. Table 2-3 shows the five different data types used to store whole-number integers.

Table 2-3. MySQL integer types

Type	Signed range	Unsigned range
`Tinyint`	−128 to 127	0 to 255
`Smallint`	−32,768 to 32,767	0 to 65,535
`Mediumint`	−8,388,608 to 8,388,607	0 to 16,777,215
`Int`	−2,147,483,648 to 2,147,483,647	0 to 4,294,967,295
`Bigint`	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615

When you create a column using one of the integer types, MySQL will allocate an appropriate amount of space to store the data, which ranges from one byte for a `tinyint` to eight bytes for a `bigint`. Therefore, you should try to choose a type that will be large enough to hold the biggest number you can envision being stored in the column without needlessly wasting storage space.

For floating-point numbers (such as 3.1415927), you may choose from the numeric types shown in Table 2-4.

Table 2-4. MySQL floating-point types

Type	Numeric range
`Float(`*`p`*`,`*`s`*`)`	−3.402823466E+38 to −1.175494351E-38
	and 1.175494351E-38 to 3.402823466E+38
`Double(`*`p`*`,`*`s`*`)`	−1.7976931348623157E+308 to −2.2250738585072014E-308
	and 2.2250738585072014E-308 to 1.7976931348623157E+308

When using a floating-point type, you can specify a *precision* (the total number of allowable digits both to the left and to the right of the decimal point) and a *scale* (the number of allowable digits to the right of the decimal point), but they are not required. These values are represented in Table 2-4 as *`p`* and `s`. If you specify a precision and scale for your floating-point column, remember that the data stored in the column will be

rounded if the number of digits exceeds the scale and/or precision of the column. For example, a column defined as `float(4,2)` will store a total of four digits, two to the left of the decimal and two to the right of the decimal. Therefore, such a column would handle the numbers 27.44 and 8.19 just fine, but the number 17.8675 would be rounded to 17.87, and attempting to store the number 178.375 in your `float(4,2)` column would generate an error.

Like the integer types, floating-point columns can be defined as `unsigned`, but this designation only prevents negative numbers from being stored in the column rather than altering the range of data that may be stored in the column.

Temporal Data

Along with strings and numbers, you will almost certainly be working with information about dates and/or times. This type of data is referred to as *temporal*, and some examples of temporal data in a database include:

- The future date that a particular event is expected to happen, such as shipping a customer's order
- The date that a customer's order was shipped
- The date and time that a user modified a particular row in a table
- An employee's birth date
- The year corresponding to a row in a `yearly_sales` fact table in a data warehouse
- The elapsed time needed to complete a wiring harness on an automobile assembly line

MySQL includes data types to handle all of these situations. Table 2-5 shows the temporal data types supported by MySQL.

Table 2-5. MySQL temporal types

Type	Default format	Allowable values
`Date`	YYYY-MM-DD	`1000-01-01` to `9999-12-31`
`Datetime`	YYYY-MM-DD HH:MI:SS	`1000-01-01 00:00:00` to `9999-12-31 23:59:59`
`Timestamp`	YYYY-MM-DD HH:MI:SS	`1970-01-01 00:00:00` to `2037-12-31 23:59:59`
`Year`	YYYY	`1901` to `2155`
`Time`	HHH:MI:SS	`-838:59:59` to `838:59:59`

While database servers store temporal data in various ways, the purpose of a format string (second column of Table 2-5) is to show how the data will be represented when retrieved, along with how a date string should be constructed when inserting or updating a temporal column. Thus, if you wanted to insert the date March 23, 2005 into a `date` column using the default format `YYYY-MM-DD`, you would use the string

'2005-03-23'. Chapter 7 fully explores how temporal data is constructed and displayed.

Each database server allows a different range of dates for temporal columns. Oracle Database accepts dates ranging from 4712 BC to 9999 AD, while SQL Server only handles dates ranging from 1753 AD to 9999 AD (unless you are using SQL Server 2008's new `datetime2` data type, which allows for dates ranging from 1 AD to 9999 AD). MySQL falls in between Oracle and SQL Server and can store dates from 1000 AD to 9999 AD. Although this might not make any difference for most systems that track current and future events, it is important to keep in mind if you are storing historical dates.

Table 2-6 describes the various components of the date formats shown in Table 2-5.

Table 2-6. Date format components

Component	Definition	Range
YYYY	Year, including century	1000 to 9999
MM	Month	01 (January) to 12 (December)
DD	Day	01 to 31
HH	Hour	00 to 23
HHH	Hours (elapsed)	-838 to 838
MI	Minute	00 to 59
SS	Second	00 to 59

Here's how the various temporal types would be used to implement the examples shown earlier:

- Columns to hold the expected future shipping date of a customer order and an employee's birth date would use the `date` type, since it is unnecessary to know at what time a person was born and unrealistic to schedule a future shipment down to the second.
- A column to hold information about when a customer order was actually shipped would use the `datetime` type, since it is important to track not only the date that the shipment occurred but the time as well.
- A column that tracks when a user last modified a particular row in a table would use the `timestamp` type. The `timestamp` type holds the same information as the `datetime` type (year, month, day, hour, minute, second), but a `timestamp` column will automatically be populated with the current date/time by the MySQL server when a row is added to a table or when a row is later modified.
- A column holding just year data would use the `year` type.

- Columns that hold data regarding the length of time needed to complete a task would use the `time` type. For this type of data, it would be unnecessary and confusing to store a date component, since you are interested only in the number of hours/minutes/seconds needed to complete the task. This information could be derived using two `datetime` columns (one for the task start date/time and the other for the task completion date/time) and subtracting one from the other, but it is simpler to use a single `time` column.

Chapter 7 explores how to work with each of these temporal data types.

Table Creation

Now that you have a firm grasp on what data types may be stored in a MySQL database, it's time to see how to use these types in table definitions. Let's start by defining a table to hold information about a person.

Step 1: Design

A good way to start designing a table is to do a bit of brainstorming to see what kind of information would be helpful to include. Here's what I came up with after thinking for a short time about the types of information that describe a person:

- Name
- Gender
- Birth date
- Address
- Favorite foods

This is certainly not an exhaustive list, but it's good enough for now. The next step is to assign column names and data types. Table 2-7 shows my initial attempt.

Table 2-7. Person table, first pass

Column	Type	Allowable values
`Name`	`Varchar(40)`	
`Gender`	`Char(1)`	`M`, `F`
`Birth_date`	`Date`	
`Address`	`Varchar(100)`	
`Favorite_foods`	`Varchar(200)`	

The `name`, `address`, and `favorite_foods` columns are of type `varchar` and allow for free-form data entry. The `gender` column allows a single character which should equal only `M` or `F`. The `birth_date` column is of type `date`, since a time component is not needed.

Step 2: Refinement

In Chapter 1, you were introduced to the concept of *normalization*, which is the process of ensuring that there are no duplicate (other than foreign keys) or compound columns in your database design. In looking at the columns in the `person` table a second time, the following issues arise:

- The `name` column is actually a compound object consisting of a first name and a last name.
- Since multiple people can have the same name, gender, birth date, and so forth, there are no columns in the `person` table that guarantee uniqueness.
- The `address` column is also a compound object consisting of street, city, state/province, country, and postal code.
- The `favorite_foods` column is a list containing `0`, `1`, or more independent items. It would be best to create a separate table for this data that includes a foreign key to the `person` table so that you know to which person a particular food may be attributed.

After taking these issues into consideration, Table 2-8 gives a normalized version of the `person` table.

Table 2-8. Person table, second pass

Column	Type	Allowable values
Person_id	Smallint (unsigned)	
First_name	Varchar(20)	
Last_name	Varchar(20)	
Gender	Char(1)	M, F
Birth_date	Date	
Street	Varchar(30)	
City	Varchar(20)	
State	Varchar(20)	
Country	Varchar(20)	
Postal_code	Varchar(20)	

Now that the `person` table has a primary key (`person_id`) to guarantee uniqueness, the next step is to build a `favorite_food` table that includes a foreign key to the `person` table. Table 2-9 shows the result.

Table 2-9. Favorite_food table

Column	Type
Person_id	Smallint (unsigned)
Food	Varchar(20)

The `person_id` and `food` columns comprise the primary key of the `favorite_food` table, and the `person_id` column is also a foreign key to the `person` table.

How Much Is Enough?

Moving the `favorite_foods` column out of the `person` table was definitely a good idea, but are we done yet? What happens, for example, if one person lists "pasta" as a favorite food while another person lists "spaghetti"? Are they the same thing? In order to prevent this problem, you might decide that you want people to choose their favorite foods from a list of options, in which case you should create a `food` table with `food_id` and `food_name` columns, and then change the `favorite_food` table to contain a foreign key to the `food` table. While this design would be fully normalized, you might decide that you simply want to store the values that the user has entered, in which case you may leave the table as is.

Step 3: Building SQL Schema Statements

Now that the design is complete for the two tables holding information about people and their favorite foods, the next step is to generate SQL statements to create the tables in the database. Here is the statement to create the `person` table:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender CHAR(1),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Everything in this statement should be fairly self-explanatory except for the last item; when you define your table, you need to tell the database server what column or columns will serve as the primary key for the table. You do this by creating a *constraint* on the table. You can add several types of constraints to a table definition. This constraint is a *primary key constraint*. It is created on the `person_id` column and given the name `pk_person`.

While on the topic of constraints, there is another type of constraint that would be useful for the `person` table. In Table 2-7, I added a third column to show the allowable values for certain columns (such as `'M'` and `'F'` for the `gender` column). Another type of constraint called a *check constraint* constrains the allowable values for a particular column. MySQL allows a check constraint to be attached to a column definition, as in the following:

```
gender CHAR(1) CHECK (gender IN ('M','F')),
```

While check constraints operate as expected on most database servers, the MySQL server allows check constraints to be defined but does not enforce them. However, MySQL does provide another character data type called `enum` that merges the check constraint into the data type definition. Here's what it would look like for the `gender` column definition:

```
gender ENUM('M','F'),
```

Here's how the `person` table definition looks with an `enum` data type for the `gender` column:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender ENUM('M','F'),
  birth_date DATE,
  street VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Later in this chapter, you will see what happens if you try to add data to a column that violates its check constraint (or, in the case of MySQL, its enumeration values).

You are now ready to run the `create table` statement using the `mysql` command-line tool. Here's what it looks like:

```
mysql> CREATE TABLE person
    ->  (person_id SMALLINT UNSIGNED,
    ->   fname VARCHAR(20),
    ->   lname VARCHAR(20),
    ->   gender ENUM('M','F'),
    ->   birth_date DATE,
    ->   street VARCHAR(30),
    ->   city VARCHAR(20),
    ->   state VARCHAR(20),
    ->   country VARCHAR(20),
    ->   postal_code VARCHAR(20),
    ->   CONSTRAINT pk_person PRIMARY KEY (person_id)
    ->  );
 Query OK, 0 rows affected (0.27 sec)
```

After processing the `create table` statement, the MySQL server returns the message "Query OK, 0 rows affected," which tells me that the statement had no syntax errors. If you want to make sure that the `person` table does, in fact, exist, you can use the `describe` command (or `desc` for short) to look at the table definition:

```
mysql> DESC person;
+-------------+----------------------+------+-----+---------+-------+
| Field       | Type                 | Null | Key | Default | Extra |
+-------------+----------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| fname       | varchar(20)          | YES  |     | NULL    |       |
| lname       | varchar(20)          | YES  |     | NULL    |       |
| gender      | enum('M','F')        | YES  |     | NULL    |       |
| birth_date  | date                 | YES  |     | NULL    |       |
| street      | varchar(30)          | YES  |     | NULL    |       |
| city        | varchar(20)          | YES  |     | NULL    |       |
| state       | varchar(20)          | YES  |     | NULL    |       |
| country     | varchar(20)          | YES  |     | NULL    |       |
| postal_code | varchar(20)          | YES  |     | NULL    |       |
+-------------+----------------------+------+-----+---------+-------+
10 rows in set (0.06 sec)
```

Columns 1 and 2 of the `describe` output are self-explanatory. Column 3 shows whether a particular column can be omitted when data is inserted into the table. I purposefully left this topic out of the discussion for now (see the sidebar "What Is Null?" on page 29 for a short discourse), but we explore it fully in Chapter 4. The fourth column shows whether a column takes part in any keys (primary or foreign); in this case, the `person_id` column is marked as the primary key. Column 5 shows whether a particular column will be populated with a default value if you omit the column when inserting data into the table. The `person_id` column shows a default value of `0`, although this would work only once, since each row in the `person` table must contain a unique value for this column (since it is the primary key). The sixth column (called "Extra") shows any other pertinent information that might apply to a column.

What Is Null?

In some cases, it is not possible or applicable to provide a value for a particular column in your table. For example, when adding data about a new customer order, the `ship_date` column cannot yet be determined. In this case, the column is said to be *null* (note that I do not say that it *equals* null), which indicates the absence of a value. Null is used for various cases where a value cannot be supplied, such as:

- Not applicable
- Unknown
- Empty set

When designing a table, you may specify which columns are allowed to be null (the default), and which columns are not allowed to be null (designated by adding the keywords `not null` after the type definition).

Now that you've created the person table, your next step is to create the favorite_food table:

```
mysql> CREATE TABLE favorite_food
    ->  (person_id SMALLINT UNSIGNED,
    ->   food VARCHAR(20),
    ->   CONSTRAINT pk_favorite_food PRIMARY KEY (person_id, food),
    ->   CONSTRAINT fk_fav_food_person_id FOREIGN KEY (person_id)
    ->     REFERENCES person (person_id)
    ->  );
Query OK, 0 rows affected (0.10 sec)
```

This should look very similar to the create table statement for the person table, with the following exceptions:

- Since a person can have more than one favorite food (which is the reason this table was created in the first place), it takes more than just the person_id column to guarantee uniqueness in the table. This table, therefore, has a two-column primary key: person_id and food.
- The favorite_food table contains another type of constraint called a *foreign key constraint*. This constrains the values of the person_id column in the favorite_food table to include *only* values found in the person table. With this constraint in place, I will not be able to add a row to the favorite_food table indicating that person_id 27 likes pizza if there isn't already a row in the person table having a person_id of 27.

If you forget to create the foreign key constraint when you first create the table, you can add it later via the alter table statement.

Describe shows the following after executing the create table statement:

```
mysql> DESC favorite_food;
+-------------+---------------------+------+-----+---------+-------+
| Field       | Type                | Null | Key | Default | Extra |
+-------------+---------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| food        | varchar(20)         |      | PRI |         |       |
+-------------+---------------------+------+-----+---------+-------+
```

Now that the tables are in place, the next logical step is to add some data.

Populating and Modifying Tables

With the person and favorite_food tables in place, you can now begin to explore the four SQL data statements: insert, update, delete, and select.

Inserting Data

Since there is not yet any data in the `person` and `favorite_food` tables, the first of the four SQL data statements to be explored will be the `insert` statement. There are three main components to an `insert` statement:

- The name of the table into which to add the data
- The names of the columns in the table to be populated
- The values with which to populate the columns

You are not required to provide data for every column in the table (unless all the columns in the table have been defined as `not null`). In some cases, those columns that are not included in the initial `insert` statement will be given a value later via an `update` statement. In other cases, a column may never receive a value for a particular row of data (such as a customer order that is canceled before being shipped, thus rendering the `ship_date` column inapplicable).

Generating numeric key data

Before inserting data into the `person` table, it would be useful to discuss how values are generated for numeric primary keys. Other than picking a number out of thin air, you have a couple of options:

- Look at the largest value currently in the table and add one.
- Let the database server provide the value for you.

Although the first option may seem valid, it proves problematic in a multiuser environment, since two users might look at the table at the same time and generate the same value for the primary key. Instead, all database servers on the market today provide a safe, robust method for generating numeric keys. In some servers, such as the Oracle Database, a separate schema object is used (called a *sequence*); in the case of MySQL, however, you simply need to turn on the *auto-increment* feature for your primary key column. Normally, you would do this at table creation, but doing it now provides the opportunity to learn another SQL schema statement, `alter table`, which is used to modify the definition of an existing table:

```
ALTER TABLE person MODIFY person_id SMALLINT UNSIGNED AUTO_INCREMENT;
```

This statement essentially redefines the `person_id` column in the `person` table. If you describe the table, you will now see the auto-increment feature listed under the "Extra" column for `person_id`:

```
mysql> DESC person;
+-------------+--------------------------+------+-----+---------+----------------+
| Field       | Type                     | Null | Key | Default | Extra          |
+-------------+--------------------------+------+-----+---------+----------------+
| person_id   | smallint(5) unsigned     |      | PRI | NULL    | auto_increment |
| .           |                          |      |     |         |                |
| .           |                          |      |     |         |                |
| .           |                          |      |     |         |                |
```

When you insert data into the `person` table, simply provide a `null` value for the `person_id` column, and MySQL will populate the column with the next available number (by default, MySQL starts at 1 for auto-increment columns).

The insert statement

Now that all the pieces are in place, it's time to add some data. The following statement creates a row in the `person` table for William Turner:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (null, 'William','Turner', 'M', '1972-05-27');
Query OK, 1 row affected (0.01 sec)
```

The feedback ("Query OK, 1 row affected") tells you that your statement syntax was proper, and that one row was added to the database (since it was an `insert` statement). You can look at the data just added to the table by issuing a `select` statement:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.06 sec)
```

As you can see, the MySQL server generated a value of `1` for the primary key. Since there is only a single row in the `person` table, I neglected to specify which row I am interested in and simply retrieved all the rows in the table. If there were more than one row in the table, however, I could add a `where` clause to specify that I want to retrieve data only for the row having a value of `1` for the `person_id` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE person_id = 1;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

While this query specifies a particular primary key value, you can use any column in the table to search for rows, as shown by the following query, which finds all rows with a value of `'Turner'` for the `lname` column:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE lname = 'Turner';
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

Before moving on, a couple of things about the earlier `insert` statement are worth mentioning:

- Values were not provided for any of the address columns. This is fine, since `null`s are allowed for those columns.
- The value provided for the `birth_date` column was a string. As long as you match the required format shown in Table 2-5, MySQL will convert the string to a date for you.
- The column names and the values provided must correspond in number and type. If you name seven columns and provide only six values, or if you provide values that cannot be converted to the appropriate data type for the corresponding column, you will receive an error.

William has also provided information about his favorite three foods, so here are three `insert` statements to store his food preferences:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'pizza');
Query OK, 1 row affected (0.01 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'cookies');
Query OK, 1 row affected (0.00 sec)
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'nachos');
Query OK, 1 row affected (0.01 sec)
```

Here's a query that retrieves William's favorite foods in alphabetical order using an `order by` clause:

```
mysql> SELECT food
    -> FROM favorite_food
    -> WHERE person_id = 1
    -> ORDER BY food;
+---------+
| food    |
+---------+
| cookies |
| nachos  |
| pizza   |
```

```
+---------+
3 rows in set (0.02 sec)
```

The `order by` clause tells the server how to sort the data returned by the query. Without the `order by` clause, there is no guarantee that the data in the table will be retrieved in any particular order.

So that William doesn't get lonely, you can execute another `insert` statement to add Susan Smith to the `person` table:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date,
    ->   street, city, state, country, postal_code)
    -> VALUES (null, 'Susan','Smith', 'F', '1975-11-02',
    ->   '23 Maple St.', 'Arlington', 'VA', 'USA', '20220');
 Query OK, 1 row affected (0.01 sec)
```

Since Susan was kind enough to provide her address, we included five more columns than when William's data was inserted. If you query the table again, you will see that Susan's row has been assigned the value `2` for its primary key value:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
|         2 | Susan   | Smith  | 1975-11-02 |
+-----------+---------+--------+------------+
2 rows in set (0.00 sec)
```

Can I Get That in XML?

If you will be working with XML data, you will be happy to know that most database servers provide a simple way to generate XML output from a query. With MySQL, for example, you can use the `--xml` option when invoking the `mysql` tool, and all your output will automatically be formatted using XML. Here's what the favorite-food data looks like as an XML document:

```
C:\database> mysql -u lrngsql -p --xml bank
Enter password: xxxxxx
Welcome to the MySQL Monitor...

Mysql> SELECT * FROM favorite_food;
<?xml version="1.0"?>

<resultset statement="select * from favorite_food"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="person_id">1</field>
        <field name="food">cookies</field>
  </row>
  <row>
        <field name="person_id">1</field>
        <field name="food">nachos</field>
  </row>
```

```
  <row>
        <field name="person_id">1</field>
        <field name="food">pizza</field>
  </row>
</resultset>
3 rows in set (0.00 sec)
```

With SQL Server, you don't need to configure your command-line tool; you just need to add the `for xml` clause to the end of your query, as in:

```
SELECT * FROM favorite_food
FOR XML AUTO, ELEMENTS
```

Updating Data

When the data for William Turner was initially added to the table, data for the various address columns was omitted from the `insert` statement. The next statement shows how these columns can be populated via an `update` statement:

```
mysql> UPDATE person
    -> SET street = '1225 Tremont St.',
    ->   city = 'Boston',
    ->   state = 'MA',
    ->   country = 'USA',
    ->   postal_code = '02138'
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The server responded with a two-line message: the "Rows matched: 1" item tells you that the condition in the `where` clause matched a single row in the table, and the "Changed: 1" item tells you that a single row in the table has been modified. Since the `where` clause specifies the primary key of William's row, this is exactly what you would expect to have happen.

Depending on the conditions in your `where` clause, it is also possible to modify more than one row using a single statement. Consider, for example, what would happen if your `where` clause looked as follows:

```
WHERE person_id < 10
```

Since both William and Susan have a `person_id` value less than 10, both of their rows would be modified. If you leave off the `where` clause altogether, your `update` statement will modify every row in the table.

Deleting Data

It seems that William and Susan aren't getting along very well together, so one of them has got to go. Since William was there first, Susan will get the boot courtesy of the `delete` statement:

```
mysql> DELETE FROM person
    -> WHERE person_id = 2;
Query OK, 1 row affected (0.01 sec)
```

Again, the primary key is being used to isolate the row of interest, so a single row is deleted from the table. Similar to the `update` statement, more than one row can be deleted depending on the conditions in your `where` clause, and all rows will be deleted if the `where` clause is omitted.

When Good Statements Go Bad

So far, all of the SQL data statements shown in this chapter have been well formed and have played by the rules. Based on the table definitions for the `person` and `favorite_food` tables, however, there are lots of ways that you can run afoul when inserting or modifying data. This section shows you some of the common mistakes that you might come across and how the MySQL server will respond.

Nonunique Primary Key

Because the table definitions include the creation of primary key constraints, MySQL will make sure that duplicate key values are not inserted into the tables. The next statement attempts to bypass the auto-increment feature of the `person_id` column and create another row in the `person` table with a `person_id` of `1`:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (1, 'Charles','Fulton', 'M', '1968-01-15');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

There is nothing stopping you (with the current schema objects, at least) from creating two rows with identical names, addresses, birth dates, and so on, as long as they have different values for the `person_id` column.

Nonexistent Foreign Key

The table definition for the `favorite_food` table includes the creation of a foreign key constraint on the `person_id` column. This constraint ensures that all values of `person_id` entered into the `favorite_food` table exist in the `person` table. Here's what would happen if you tried to create a row that violates this constraint:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (999, 'lasagna');
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
fails ('bank'.'favorite_food', CONSTRAINT 'fk_fav_food_person_id' FOREIGN KEY
('person_id') REFERENCES 'person' ('person_id'))
```

In this case, the `favorite_food` table is considered the *child* and the `person` table is considered the *parent*, since the `favorite_food` table is dependent on the `person` table

for some of its data. If you plan to enter data into both tables, you will need to create a row in `parent` before you can enter data into `favorite_food`.

Foreign key constraints are enforced only if your tables are created using the InnoDB storage engine. We discuss MySQL's storage engines in Chapter 12.

Column Value Violations

The `gender` column in the `person` table is restricted to the values `'M'` for male and `'F'` for female. If you mistakenly attempt to set the value of the column to any other value, you will receive the following response:

```
mysql> UPDATE person
    -> SET gender = 'Z'
    -> WHERE person_id = 1;
ERROR 1265 (01000): Data truncated for column 'gender' at row 1
```

The error message is a bit confusing, but it gives you the general idea that the server is unhappy about the value provided for the `gender` column.

Invalid Date Conversions

If you construct a string with which to populate a `date` column, and that string does not match the expected format, you will receive another error. Here's an example that uses a date format that does not match the default date format of "YYYY-MM-DD":

```
mysql> UPDATE person
    -> SET birth_date = 'DEC-21-1980'
    -> WHERE person_id = 1;
ERROR 1292 (22007): Incorrect date value: 'DEC-21-1980' for column 'birth_date'
at row 1
```

In general, it is always a good idea to explicitly specify the format string rather than relying on the default format. Here's another version of the statement that uses the `str_to_date` function to specify which format string to use:

```
mysql> UPDATE person
    -> SET birth_date = str_to_date('DEC-21-1980' , '%b-%d-%Y')
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.12 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Not only is the database server happy, but William is happy as well (we just made him eight years younger, without the need for expensive cosmetic surgery!).

Earlier in the chapter, when I discussed the various temporal data types, I showed date-formatting strings such as "YYYY-MM-DD". While many database servers use this style of formatting, MySQL uses `%Y` to indicate a four-character year. Here are a few more formatters that you might need when converting strings to `datetime`s in MySQL:

```
%a The short weekday name, such as Sun, Mon, ...
%b The short month name, such as Jan, Feb, ...
%c The numeric month (0..12)
%d The numeric day of the month (00..31)
%f The number of microseconds (000000..999999)
%H The hour of the day, in 24-hour format (00..23)
%h The hour of the day, in 12-hour format (01..12)
%i The minutes within the hour (00..59)
%j The day of year (001..366)
%M The full month name (January..December)
%m The numeric month
%p AM or PM
%s The number of seconds (00..59)
%W The full weekday name (Sunday..Saturday)
%w The numeric day of the week (0=Sunday..6=Saturday)
%Y The four-digit year
```

The Bank Schema

For the remainder of the book, you use a group of tables that model a community bank. Some of the tables include `Employee`, `Branch`, `Account`, `Customer`, `Product`, and `Transaction`. The entire schema and example data should have been created when you followed the final steps at the beginning of the chapter for loading the MySQL server and generating the sample data. To see a diagram of the tables and their columns and relationships, see Appendix A.

Table 2-10 shows all the tables used in the bank schema along with short definitions.

Table 2-10. Bank schema definitions

Table name	Definition
`Account`	A particular product opened for a particular customer
`Branch`	A location at which banking transactions are conducted
`Business`	A corporate customer (subtype of the `Customer` table)
`Customer`	A person or corporation known to the bank
`Department`	A group of bank employees implementing a particular banking function
`Employee`	A person working for the bank
`Individual`	A noncorporate customer (subtype of the `Customer` table)
`Officer`	A person allowed to transact business for a corporate customer
`Product`	A banking service offered to customers
`Product_type`	A group of products having a similar function
`Transaction`	A change made to an account balance

Feel free to experiment with the tables as much as you want, including adding your own tables to expand the bank's business functions. You can always drop the database and re-create it from the downloaded file if you want to make sure your sample data is intact.

If you want to see the tables available in your database, you can use the `show tables` command, as in:

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_bank |
+----------------+
| account        |
| branch         |
| business       |
| customer       |
| department     |
| employee       |
| favorite_food  |
| individual     |
| officer        |
| person         |
| product        |
| product_type   |
| transaction    |
+----------------+
13 rows in set (0.10 sec)
```

Along with the 11 tables in the bank schema, the table listing also includes the two tables created in this chapter: `person` and `favorite_food`. These tables will not be used in later chapters, so feel free to drop them by issuing the following commands:

```
mysql> DROP TABLE favorite_food;
Query OK, 0 rows affected (0.56 sec)
mysql> DROP TABLE person;
Query OK, 0 rows affected (0.05 sec)
```

If you want to look at the columns in a table, you can use the `describe` command. Here's an example of the `describe` output for the `customer` table:

```
mysql> DESC customer;
+--------------+------------------+------+-----+---------+----------------+
| Field        | Type             | Null | Key | Default | Extra          |
+--------------+------------------+------+-----+---------+----------------+
| cust_id      | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| fed_id       | varchar(12)      | NO   |     | NULL    |                |
| cust_type_cd | enum('I','B')    | NO   |     | NULL    |                |
| address      | varchar(30)      | YES  |     | NULL    |                |
| city         | varchar(20)      | YES  |     | NULL    |                |
| state        | varchar(20)      | YES  |     | NULL    |                |
| postal_code  | varchar(10)      | YES  |     | NULL    |                |
+--------------+------------------+------+-----+---------+----------------+
7 rows in set (0.03 sec)
```

The more comfortable you are with the example database, the better you will understand the examples and, consequently, the concepts in the following chapters.

CHAPTER 3
Query Primer

So far, you have seen a few examples of database queries (a.k.a. `select` statements) sprinkled throughout the first two chapters. Now it's time to take a closer look at the different parts of the `select` statement and how they interact.

Query Mechanics

Before dissecting the `select` statement, it might be interesting to look at how queries are executed by the MySQL server (or, for that matter, any database server). If you are using the `mysql` command-line tool (which I assume you are), then you have already logged in to the MySQL server by providing your username and password (and possibly a hostname if the MySQL server is running on a different computer). Once the server has verified that your username and password are correct, a *database connection* is generated for you to use. This connection is held by the application that requested it (which, in this case, is the `mysql` tool) until the application releases the connection (i.e., as a result of your typing `quit`) or the server closes the connection (i.e., when the server is shut down). Each connection to the MySQL server is assigned an identifier, which is shown to you when you first log in:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 6.0.3-alpha-community MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

In this case, my connection ID is `11`. This information might be useful to your database administrator if something goes awry, such as a malformed query that runs for hours, so you might want to jot it down.

Once the server has verified your username and password and issued you a connection, you are ready to execute queries (along with other SQL statements). Each time a query is sent to the server, the server checks the following things prior to statement execution:

- Do you have permission to execute the statement?
- Do you have permission to access the desired data?
- Is your statement syntax correct?

If your statement passes these three tests, then your query is handed to the *query optimizer*, whose job it is to determine the most efficient way to execute your query. The optimizer will look at such things as the order in which to join the tables named in your `from` clause and what indexes are available, and then picks an *execution plan*, which the server uses to execute your query.

Understanding and influencing how your database server chooses execution plans is a fascinating topic that many of you will wish to explore. For those readers using MySQL, you might consider reading Baron Schwartz et al.'s *High Performance MySQL (http://oreilly.com/catalog/9780596101718/)* (O'Reilly). Among other things, you will learn how to generate indexes, analyze execution plans, influence the optimizer via query hints, and tune your server's startup parameters. If you are using Oracle Database or SQL Server, dozens of tuning books are available.

Once the server has finished executing your query, the *result set* is returned to the calling application (which is, once again, the `mysql` tool). As I mentioned in Chapter 1, a result set is just another table containing rows and columns. If your query fails to yield any results, the `mysql` tool will show you the message found at the end of the following example:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname = 'Bkadfl';
Empty set(0.00 sec)
```

If the query returns one or more rows, the `mysql` tool will format the results by adding column headers and by constructing boxes around the columns using the -, |, and + symbols, as shown in the next example:

```
mysql> SELECT fname, lname
    -> FROM employee;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| Michael  | Smith     |
| Susan    | Barker    |
| Robert   | Tyler     |
| Susan    | Hawthorne |
| John     | Gooding   |
| Helen    | Fleming   |
| Chris    | Tucker    |
| Sarah    | Parker    |
| Jane     | Grossman  |
```

```
| Paula    | Roberts   |
| Thomas   | Ziegler   |
| Samantha | Jameson   |
| John     | Blake     |
| Cindy    | Mason     |
| Frank    | Portman   |
| Theresa  | Markham   |
| Beth     | Fowler    |
| Rick     | Tulman    |
+----------+-----------+
18 rows in set (0.00 sec)
```

This query returns the first and last names of all the employees in the `employee` table. After the last row of data is displayed, the `mysql` tool displays a message telling you how many rows were returned, which, in this case, is 18.

Query Clauses

Several components or *clauses* make up the `select` statement. While only one of them is mandatory when using MySQL (the `select` clause), you will usually include at least two or three of the six available clauses. Table 3-1 shows the different clauses and their purposes.

Table 3-1. Query clauses

Clause name	Purpose
`Select`	Determines which columns to include in the query's result set
`From`	Identifies the tables from which to draw data and how the tables should be joined
`Where`	Filters out unwanted data
`Group by`	Used to group rows together by common column values
`Having`	Filters out unwanted groups
`Order by`	Sorts the rows of the final result set by one or more columns

All of the clauses shown in Table 3-1 are included in the ANSI specification; additionally, several other clauses are unique to MySQL that we explore in Appendix B. The following sections delve into the uses of the six major query clauses.

The select Clause

Even though the `select` clause is the first clause of a `select` statement, it is one of the last clauses that the database server evaluates. The reason for this is that before you can determine what to include in the final result set, you need to know all of the possible columns that *could* be included in the final result set. In order to fully understand the role of the `select` clause, therefore, you will need to understand a bit about the `from` clause. Here's a query to get started:

```
mysql> SELECT *
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       1 | Operations     |
|       2 | Loans          |
|       3 | Administration |
+---------+----------------+
3 rows in set (0.04 sec)
```

In this query, the `from` clause lists a single table (`department`), and the `select` clause indicates that *all* columns (designated by *) in the `department` table should be included in the result set. This query could be described in English as follows:

Show me all the columns and all the rows in the `department` *table*.

In addition to specifying all the columns via the asterisk character, you can explicitly name the columns you are interested in, such as:

```
mysql> SELECT dept_id, name
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       1 | Operations     |
|       2 | Loans          |
|       3 | Administration |
+---------+----------------+
3 rows in set (0.01 sec)
```

The results are identical to the first query, since all the columns in the `department` table (`dept_id` and `name`) are named in the `select` clause. You can choose to include only a subset of the columns in the `department` table as well:

```
mysql> SELECT name
    -> FROM department;
+----------------+
| name           |
+----------------+
| Operations     |
| Loans          |
| Administration |
+----------------+
3 rows in set (0.00 sec)
```

The job of the `select` clause, therefore, is the following:

The `select` *clause determines which of all possible columns should be included in the query's result set*.

If you were limited to including only columns from the table or tables named in the `from` clause, things would be rather dull. However, you can spice things up by including in your `select` clause such things as:

- Literals, such as numbers or strings
- Expressions, such as `transaction.amount * -1`
- Built-in function calls, such as `ROUND(transaction.amount, 2)`
- User-defined function calls

The next query demonstrates the use of a table column, a literal, an expression, and a built-in function call in a single query against the `employee` table:

```
mysql> SELECT emp_id,
    ->  'ACTIVE',
    ->  emp_id * 3.14159,
    ->  UPPER(lname)
    -> FROM employee;
+--------+--------+-----------------+--------------+
| emp_id | ACTIVE | emp_id * 3.14159 | UPPER(lname) |
+--------+--------+-----------------+--------------+
|      1 | ACTIVE |          3.14159 | SMITH        |
|      2 | ACTIVE |          6.28318 | BARKER       |
|      3 | ACTIVE |          9.42477 | TYLER        |
|      4 | ACTIVE |         12.56636 | HAWTHORNE    |
|      5 | ACTIVE |         15.70795 | GOODING      |
|      6 | ACTIVE |         18.84954 | FLEMING      |
|      7 | ACTIVE |         21.99113 | TUCKER       |
|      8 | ACTIVE |         25.13272 | PARKER       |
|      9 | ACTIVE |         28.27431 | GROSSMAN     |
|     10 | ACTIVE |         31.41590 | ROBERTS      |
|     11 | ACTIVE |         34.55749 | ZIEGLER      |
|     12 | ACTIVE |         37.69908 | JAMESON      |
|     13 | ACTIVE |         40.84067 | BLAKE        |
|     14 | ACTIVE |         43.98226 | MASON        |
|     15 | ACTIVE |         47.12385 | PORTMAN      |
|     16 | ACTIVE |         50.26544 | MARKHAM      |
|     17 | ACTIVE |         53.40703 | FOWLER       |
|     18 | ACTIVE |         56.54862 | TULMAN       |
+--------+--------+-----------------+--------------+
18 rows in set (0.05 sec)
```

We cover expressions and built-in functions in detail later, but I wanted to give you a feel for what kinds of things can be included in the `select` clause. If you only need to execute a built-in function or evaluate a simple expression, you can skip the `from` clause entirely. Here's an example:

```
mysql> SELECT VERSION(),
    ->   USER(),
    ->   DATABASE();
+----------------------+------------------+------------+
| version()            | user()           | database() |
+----------------------+------------------+------------+
| 6.0.3-alpha-community | lrngsql@localhost | bank       |
+----------------------+------------------+------------+
1 row in set (0.05 sec)
```

Since this query simply calls three built-in functions and doesn't retrieve data from any tables, there is no need for a `from` clause.

Column Aliases

Although the `mysql` tool will generate labels for the columns returned by your queries, you may want to assign your own labels. While you might want to assign a new label to a column from a table (if it is poorly or ambiguously named), you will almost certainly want to assign your own labels to those columns in your result set that are generated by expressions or built-in function calls. You can do so by adding a *column alias* after each element of your `select` clause. Here's the previous query against the `employee` table with column aliases applied to three of the columns:

```
mysql> SELECT emp_id,
    ->   'ACTIVE' status,
    ->   emp_id * 3.14159 empid_x_pi,
    ->   UPPER(lname) last_name_upper
    -> FROM employee;
+--------+--------+-----------+-----------------+
| emp_id | status | empid_x_pi | last_name_upper |
+--------+--------+-----------+-----------------+
|      1 | ACTIVE |    3.14159 | SMITH           |
|      2 | ACTIVE |    6.28318 | BARKER          |
|      3 | ACTIVE |    9.42477 | TYLER           |
|      4 | ACTIVE |   12.56636 | HAWTHORNE       |
|      5 | ACTIVE |   15.70795 | GOODING         |
|      6 | ACTIVE |   18.84954 | FLEMING         |
|      7 | ACTIVE |   21.99113 | TUCKER          |
|      8 | ACTIVE |   25.13272 | PARKER          |
|      9 | ACTIVE |   28.27431 | GROSSMAN        |
|     10 | ACTIVE |   31.41590 | ROBERTS         |
|     11 | ACTIVE |   34.55749 | ZIEGLER         |
|     12 | ACTIVE |   37.69908 | JAMESON         |
|     13 | ACTIVE |   40.84067 | BLAKE           |
|     14 | ACTIVE |   43.98226 | MASON           |
|     15 | ACTIVE |   47.12385 | PORTMAN         |
|     16 | ACTIVE |   50.26544 | MARKHAM         |
|     17 | ACTIVE |   53.40703 | FOWLER          |
|     18 | ACTIVE |   56.54862 | TULMAN          |
+--------+--------+-----------+-----------------+
18 rows in set (0.00 sec)
```

If you look at the column headers, you can see that the second, third, and fourth columns now have reasonable names instead of simply being labeled with the function or expression that generated the column. If you look at the `select` clause, you can see how the column aliases `status`, `empid_x_pi`, and `last_name_upper` are added after the second, third, and fourth columns. I think you will agree that the output is easier to understand with column aliases in place, and it would be easier to work with programmatically if you were issuing the query from within Java or C# rather than interactively via the

mysql tool. In order to make your column aliases stand out even more, you also have the option of using the as keyword before the alias name, as in:

```
mysql> SELECT emp_id,
    ->   'ACTIVE' AS status,
    ->   emp_id * 3.14159 AS empid_x_pi,
    ->   UPPER(lname) AS last_name_upper
    -> FROM employee;
```

Many people feel that including the optional as keyword improves readability, although I have chosen not to use it for the examples in this book.

Removing Duplicates

In some cases, a query might return duplicate rows of data. For example, if you were to retrieve the IDs of all customers that have accounts, you would see the following:

```
mysql> SELECT cust_id
    -> FROM account;
+---------+
| cust_id |
+---------+
|       1 |
|       1 |
|       1 |
|       2 |
|       2 |
|       3 |
|       3 |
|       4 |
|       4 |
|       4 |
|       5 |
|       6 |
|       6 |
|       7 |
|       8 |
|       8 |
|       9 |
|       9 |
|       9 |
|      10 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
24 rows in set (0.00 sec)
```

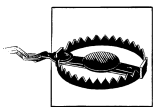
Since some customers have more than one account, you will see the same customer ID once for each account owned by that customer. What you probably want in this case is the *distinct* set of customers that have accounts, instead of seeing the customer ID

for each row in the `account` table. You can achieve this by adding the keyword `distinct` directly after the `select` keyword, as demonstrated by the following:

```
mysql> SELECT DISTINCT cust_id
    -> FROM account;
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       5 |
|       6 |
|       7 |
|       8 |
|       9 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
13 rows in set (0.01 sec)
```

The result set now contains 13 rows, one for each distinct customer, rather than 24 rows, one for each account.

If you do not want the server to remove duplicate data, or you are sure there will be no duplicates in your result set, you can specify the `ALL` keyword instead of specifying `DISTINCT`. However, the `ALL` keyword is the default and never needs to be explicitly named, so most programmers do not include `ALL` in their queries.



Keep in mind that generating a distinct set of results requires the data to be sorted, which can be time-consuming for large result sets. Don't fall into the trap of using `DISTINCT` just to be sure there are no duplicates; instead, take the time to understand the data you are working with so that you will know whether duplicates are possible.

The from Clause

Thus far, you have seen queries whose `from` clauses contain a single table. Although most SQL books will define the `from` clause as simply a list of one or more tables, I would like to broaden the definition as follows:

The `from` clause defines the tables used by a query, along with the means of linking the tables together.

This definition is composed of two separate but related concepts, which we explore in the following sections.

Tables

When confronted with the term *table*, most people think of a set of related rows stored in a database. While this does describe one type of table, I would like to use the word in a more general way by removing any notion of how the data might be stored and concentrating on just the set of related rows. Three different types of tables meet this relaxed definition:

- Permanent tables (i.e., created using the `create table` statement)
- Temporary tables (i.e., rows returned by a subquery)
- Virtual tables (i.e., created using the `create view` statement)

Each of these table types may be included in a query's `from` clause. By now, you should be comfortable with including a permanent table in a `from` clause, so I briefly describe the other types of tables that can be referenced in a `from` clause.

Subquery-generated tables

A subquery is a query contained within another query. Subqueries are surrounded by parentheses and can be found in various parts of a `select` statement; within the `from` clause, however, a subquery serves the role of generating a temporary table that is visible from all other query clauses and can interact with other tables named in the `from` clause. Here's a simple example:

```
mysql> SELECT e.emp_id, e.fname, e.lname
    -> FROM (SELECT emp_id, fname, lname, start_date, title
    ->       FROM employee) e;
+--------+----------+-----------+
| emp_id | fname    | lname     |
+--------+----------+-----------+
|      1 | Michael  | Smith     |
|      2 | Susan    | Barker    |
|      3 | Robert   | Tyler     |
|      4 | Susan    | Hawthorne |
|      5 | John     | Gooding   |
|      6 | Helen    | Fleming   |
|      7 | Chris    | Tucker    |
|      8 | Sarah    | Parker    |
|      9 | Jane     | Grossman  |
|     10 | Paula    | Roberts   |
|     11 | Thomas   | Ziegler   |
|     12 | Samantha | Jameson   |
|     13 | John     | Blake     |
|     14 | Cindy    | Mason     |
|     15 | Frank    | Portman   |
|     16 | Theresa  | Markham   |
|     17 | Beth     | Fowler    |
|     18 | Rick     | Tulman    |
+--------+----------+-----------+
18 rows in set (0.00 sec)
```

In this example, a subquery against the `employee` table returns five columns, and the *containing query* references three of the five available columns. The subquery is referenced by the containing query via its alias, which, in this case, is `e`. This is a simplistic and not particularly useful example of a subquery in a `from` clause; you will find detailed coverage of subqueries in Chapter 9.

Views

A view is a query that is stored in the data dictionary. It looks and acts like a table, but there is no data associated with a view (this is why I call it a *virtual* table). When you issue a query against a view, your query is merged with the view definition to create a final query to be executed.

To demonstrate, here's a view definition that queries the `employee` table and includes a call to a built-in function:

```
mysql> CREATE VIEW employee_vw AS
    -> SELECT emp_id, fname, lname,
    ->   YEAR(start_date) start_year
    -> FROM employee;
Query OK, 0 rows affected (0.10 sec)
```

When the view is created, no additional data is generated or stored: the server simply tucks away the `select` statement for future use. Now that the view exists, you can issue queries against it, as in:

```
mysql> SELECT emp_id, start_year
    -> FROM employee_vw;
+--------+------------+
| emp_id | start_year |
+--------+------------+
|      1 |       2005 |
|      2 |       2006 |
|      3 |       2005 |
|      4 |       2006 |
|      5 |       2007 |
|      6 |       2008 |
|      7 |       2008 |
|      8 |       2006 |
|      9 |       2006 |
|     10 |       2006 |
|     11 |       2004 |
|     12 |       2007 |
|     13 |       2004 |
|     14 |       2006 |
|     15 |       2007 |
|     16 |       2005 |
|     17 |       2006 |
|     18 |       2006 |
+--------+------------+
18 rows in set (0.07 sec)
```

Views are created for various reasons, including to hide columns from users and to simplify complex database designs.

Table Links

The second deviation from the simple `from` clause definition is the mandate that if more than one table appears in the `from` clause, the conditions used to *link* the tables must be included as well. This is not a requirement of MySQL or any other database server, but it is the ANSI-approved method of joining multiple tables, and it is the most portable across the various database servers. We explore joining multiple tables in depth in Chapters 5 and 10, but here's a simple example in case I have piqued your curiosity:

```
mysql> SELECT employee.emp_id, employee.fname,
    ->   employee.lname, department.name dept_name
    -> FROM employee INNER JOIN department
    ->   ON employee.dept_id = department.dept_id;
+--------+----------+-----------+----------------+
| emp_id | fname    | lname     | dept_name      |
+--------+----------+-----------+----------------+
|      1 | Michael  | Smith     | Administration |
|      2 | Susan    | Barker    | Administration |
|      3 | Robert   | Tyler     | Administration |
|      4 | Susan    | Hawthorne | Operations     |
|      5 | John     | Gooding   | Loans          |
|      6 | Helen    | Fleming   | Operations     |
|      7 | Chris    | Tucker    | Operations     |
|      8 | Sarah    | Parker    | Operations     |
|      9 | Jane     | Grossman  | Operations     |
|     10 | Paula    | Roberts   | Operations     |
|     11 | Thomas   | Ziegler   | Operations     |
|     12 | Samantha | Jameson   | Operations     |
|     13 | John     | Blake     | Operations     |
|     14 | Cindy    | Mason     | Operations     |
|     15 | Frank    | Portman   | Operations     |
|     16 | Theresa  | Markham   | Operations     |
|     17 | Beth     | Fowler    | Operations     |
|     18 | Rick     | Tulman    | Operations     |
+--------+----------+-----------+----------------+
18 rows in set (0.05 sec)
```

The previous query displays data from both the `employee` table (`emp_id`, `fname`, `lname`) and the `department` table (`name`), so both tables are included in the `from` clause. The mechanism for linking the two tables (referred to as a *join*) is the employee's department affiliation stored in the `employee` table. Thus, the database server is instructed to use the value of the `dept_id` column in the `employee` table to look up the associated department name in the `department` table. Join conditions for two tables are found in the `on` subclause of the `from` clause; in this case, the join condition is `ON employee.dept_id = department.dept_id`. Again, please refer to Chapter 5 for a thorough discussion of joining multiple tables.

Defining Table Aliases

When multiple tables are joined in a single query, you need a way to identify which table you are referring to when you reference columns in the `select`, `where`, `group by`, `having`, and `order by` clauses. You have two choices when referencing a table outside the `from` clause:

- Use the entire table name, such as `employee.emp_id`.
- Assign each table an *alias* and use the alias throughout the query.

In the previous query, I chose to use the entire table name in the `select` and `on` clauses. Here's what the same query looks like using table aliases:

```
SELECT e.emp_id, e.fname, e.lname,
  d.name dept_name
FROM employee e INNER JOIN department d
  ON e.dept_id = d.dept_id;
```

If you look closely at the `from` clause, you will see that the `employee` table is assigned the alias `e`, and the `department` table is assigned the alias `d`. These aliases are then used in the `on` clause when defining the join condition as well as in the `select` clause when specifying the columns to include in the result set. I hope you will agree that using aliases makes for a more compact statement without causing confusion (as long as your choices for alias names are reasonable). Additionally, you may use the `as` keyword with your table aliases, similar to what was demonstrated earlier for column aliases:

```
SELECT e.emp_id, e.fname, e.lname,
  d.name dept_name
FROM employee AS e INNER JOIN department AS d
  ON e.dept_id = d.dept_id;
```

I have found that roughly half of the database developers I have worked with use the `as` keyword with their column and table aliases, and half do not.

The where Clause

The queries shown thus far in the chapter have selected every row from the `employee`, `department`, or `account` table (except for the demonstration of `distinct` earlier in the chapter). Most of the time, however, you will not wish to retrieve *every* row from a table but will want a way to filter out those rows that are not of interest. This is a job for the `where` clause.

The `where` *clause is the mechanism for filtering out unwanted rows from your result set.*

For example, perhaps you are interested in retrieving data from the `employee` table, but only for those employees who are employed as head tellers. The following query employs a `where` clause to retrieve *only* the four head tellers:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller';
+--------+---------+---------+------------+-------------+
| emp_id | fname   | lname   | start_date | title       |
+--------+---------+---------+------------+-------------+
|      6 | Helen   | Fleming | 2008-03-17 | Head Teller |
|     10 | Paula   | Roberts | 2006-07-27 | Head Teller |
|     13 | John    | Blake   | 2004-05-11 | Head Teller |
|     16 | Theresa | Markham | 2005-03-15 | Head Teller |
+--------+---------+---------+------------+-------------+
4 rows in set (1.17 sec)
```

In this case the `where` clause filtered out 14 of the 18 employee rows. This `where` clause contains a single *filter condition*, but you can include as many conditions as required; individual conditions are separated using operators such as `and`, `or`, and `not` (see Chapter 4 for a complete discussion of the `where` clause and filter conditions). Here's an extension of the previous query that includes a second condition stating that only those employees with a start date later than January 1, 2006 should be included:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   AND start_date > '2006-01-01';
+--------+-------+---------+------------+-------------+
| emp_id | fname | lname   | start_date | title       |
+--------+-------+---------+------------+-------------+
|      6 | Helen | Fleming | 2008-03-17 | Head Teller |
|     10 | Paula | Roberts | 2006-07-27 | Head Teller |
+--------+-------+---------+------------+-------------+
2 rows in set (0.01 sec)
```

The first condition (`title = 'Head Teller'`) filtered out 14 of 18 employee rows, and the second condition (`start_date > '2006-01-01'`) filtered out an additional 2 rows, leaving 2 rows in the final result set. Let's see what would happen if you change the operator separating the two conditions from `and` to `or`:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   OR start_date > '2006-01-01';
+--------+----------+-----------+------------+--------------------+
| emp_id | fname    | lname     | start_date | title              |
+--------+----------+-----------+------------+--------------------+
|      2 | Susan    | Barker    | 2006-09-12 | Vice President     |
|      4 | Susan    | Hawthorne | 2006-04-24 | Operations Manager |
|      5 | John     | Gooding   | 2007-11-14 | Loan Manager       |
|      6 | Helen    | Fleming   | 2008-03-17 | Head Teller        |
|      7 | Chris    | Tucker    | 2008-09-15 | Teller             |
|      8 | Sarah    | Parker    | 2006-12-02 | Teller             |
|      9 | Jane     | Grossman  | 2006-05-03 | Teller             |
|     10 | Paula    | Roberts   | 2006-07-27 | Head Teller        |
|     12 | Samantha | Jameson   | 2007-01-08 | Teller             |
|     13 | John     | Blake     | 2004-05-11 | Head Teller        |
```

```
|     14 | Cindy    | Mason     | 2006-08-09 | Teller             |
|     15 | Frank    | Portman   | 2007-04-01 | Teller             |
|     16 | Theresa  | Markham   | 2005-03-15 | Head Teller        |
|     17 | Beth     | Fowler    | 2006-06-29 | Teller             |
|     18 | Rick     | Tulman    | 2006-12-12 | Teller             |
+--------+----------+-----------+------------+--------------------+
15 rows in set (0.00 sec)
```

Looking at the output, you can see that all four head tellers are included in the result set, along with any other employee who started working for the bank after January 1, 2006. At least one of the two conditions is true for 15 of the 18 employees in the `employee` table. Thus, when you separate conditions using the `and` operator, *all* conditions must evaluate to `true` to be included in the result set; when you use `or`, however, only *one* of the conditions needs to evaluate to `true` for a row to be included.

So, what should you do if you need to use both `and` and `or` operators in your `where` clause? Glad you asked. You should use parentheses to group conditions together. The next query specifies that only those employees who are head tellers *and* began working for the company after January 1, 2006 *or* those employees who are tellers *and* began working after January 1, 2007 be included in the result set:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE (title = 'Head Teller' AND start_date > '2006-01-01')
    ->   OR (title = 'Teller' AND start_date > '2007-01-01');
+--------+----------+---------+------------+-------------+
| emp_id | fname    | lname   | start_date | title       |
+--------+----------+---------+------------+-------------+
|      6 | Helen    | Fleming | 2008-03-17 | Head Teller |
|      7 | Chris    | Tucker  | 2008-09-15 | Teller      |
|     10 | Paula    | Roberts | 2006-07-27 | Head Teller |
|     12 | Samantha | Jameson | 2007-01-08 | Teller      |
|     15 | Frank    | Portman | 2007-04-01 | Teller      |
+--------+----------+---------+------------+-------------+
5 rows in set (0.00 sec)
```

You should always use parentheses to separate groups of conditions when mixing different operators so that you, the database server, and anyone who comes along later to modify your code will be on the same page.

The group by and having Clauses

All the queries thus far have retrieved raw data without any manipulation. Sometimes, however, you will want to find trends in your data that will require the database server to cook the data a bit before you retrieve your result set. One such mechanism is the `group by` clause, which is used to group data by column values. For example, rather than looking at a list of employees and the departments to which they are assigned, you might want to look at a list of departments along with the number of employees assigned to each department. When using the `group by` clause, you may also use the `having`

clause, which allows you to filter group data in the same way the `where` clause lets you filter raw data.

Here's a quick look at a query that counts all the employees in each department and returns the names of those departments having more than two employees:

```
mysql> SELECT d.name, count(e.emp_id) num_employees
    -> FROM department d INNER JOIN employee e
    ->   ON d.dept_id = e.dept_id
    -> GROUP BY d.name
    -> HAVING count(e.emp_id) > 2;
+----------------+---------------+
| name           | num_employees |
+----------------+---------------+
| Administration |             3 |
| Operations     |            14 |
+----------------+---------------+
2 rows in set (0.00 sec)
```

I wanted to briefly mention these two clauses so that they don't catch you by surprise later in the book, but they are a bit more advanced than the other four `select` clauses. Therefore, I ask that you wait until Chapter 8 for a full description of how and when to use `group by` and `having`.

The order by Clause

In general, the rows in a result set returned from a query are not in any particular order. If you want your result set in a particular order, you will need to instruct the server to sort the results using the `order by` clause:

The `order by` *clause is the mechanism for sorting your result set using either raw column data or expressions based on column data.*

For example, here's another look at an earlier query against the `account` table:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|          10 | CHK        |
|          10 | SAV        |
|          10 | CD         |
|          10 | CHK        |
|          10 | SAV        |
|          13 | CHK        |
|          13 | MM         |
|           1 | CHK        |
|           1 | SAV        |
|           1 | MM         |
|          16 | CHK        |
|           1 | CHK        |
|           1 | CD         |
```

```
|          10 | CD          |
|          16 | CHK         |
|          16 | SAV         |
|           1 | CHK         |
|           1 | MM          |
|           1 | CD          |
|          16 | CHK         |
|          16 | BUS         |
|          10 | BUS         |
|          16 | CHK         |
|          13 | SBL         |
+-------------+-------------+
24 rows in set (0.00 sec)
```

If you are trying to analyze data for each employee, it would be helpful to sort the results by the `open_emp_id` column; to do so, simply add this column to the `order by` clause:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id;
+-------------+-------------+
| open_emp_id | product_cd  |
+-------------+-------------+
|           1 | CHK         |
|           1 | SAV         |
|           1 | MM          |
|           1 | CHK         |
|           1 | CD          |
|           1 | CHK         |
|           1 | MM          |
|           1 | CD          |
|          10 | CHK         |
|          10 | SAV         |
|          10 | CD          |
|          10 | CHK         |
|          10 | SAV         |
|          10 | CD          |
|          10 | BUS         |
|          13 | CHK         |
|          13 | MM          |
|          13 | SBL         |
|          16 | CHK         |
|          16 | CHK         |
|          16 | SAV         |
|          16 | CHK         |
|          16 | BUS         |
|          16 | CHK         |
+-------------+-------------+
24 rows in set (0.00 sec)
```

It is now easier to see what types of accounts each employee opened. However, it might be even better if you could ensure that the account types were shown in the same order for each distinct employee; you can accomplish this by adding the `product_cd` column after the `open_emp_id` column in the `order by` clause:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id, product_cd;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|           1 | CD         |
|           1 | CD         |
|           1 | CHK        |
|           1 | CHK        |
|           1 | CHK        |
|           1 | MM         |
|           1 | MM         |
|           1 | SAV        |
|          10 | BUS        |
|          10 | CD         |
|          10 | CD         |
|          10 | CHK        |
|          10 | CHK        |
|          10 | SAV        |
|          10 | SAV        |
|          13 | CHK        |
|          13 | MM         |
|          13 | SBL        |
|          16 | BUS        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | SAV        |
+-------------+------------+
24 rows in set (0.00 sec)
```

The result set has now been sorted first by employee ID and then by account type. The order in which columns appear in your `order by` clause does make a difference.

Ascending Versus Descending Sort Order

When sorting, you have the option of specifying *ascending* or *descending* order via the `asc` and `desc` keywords. The default is ascending, so you will need to add the `desc` keyword, only if you want to use a descending sort. For example, the following query lists all accounts sorted by available balance with the highest balance listed at the top:

```
mysql> SELECT account_id, product_cd, open_date, avail_balance
    -> FROM account
    -> ORDER BY avail_balance DESC;
+------------+------------+------------+---------------+
| account_id | product_cd | open_date  | avail_balance |
+------------+------------+------------+---------------+
|         29 | SBL        | 2004-02-22 |      50000.00 |
|         28 | CHK        | 2003-07-30 |      38552.05 |
|         24 | CHK        | 2002-09-30 |      23575.12 |
|         15 | CD         | 2004-12-28 |      10000.00 |
|         27 | BUS        | 2004-03-22 |       9345.55 |
```

```
|         22 | MM         | 2004-10-28 |       9345.55 |
|         12 | MM         | 2004-09-30 |       5487.09 |
|         17 | CD         | 2004-01-12 |       5000.00 |
|         18 | CHK        | 2001-05-23 |       3487.19 |
|          3 | CD         | 2004-06-30 |       3000.00 |
|          4 | CHK        | 2001-03-12 |       2258.02 |
|         13 | CHK        | 2004-01-27 |       2237.97 |
|          8 | MM         | 2002-12-15 |       2212.50 |
|         23 | CD         | 2004-06-30 |       1500.00 |
|          1 | CHK        | 2000-01-15 |       1057.75 |
|          7 | CHK        | 2002-11-23 |       1057.75 |
|         11 | SAV        | 2000-01-15 |        767.77 |
|         10 | CHK        | 2003-09-12 |        534.12 |
|          2 | SAV        | 2000-01-15 |        500.00 |
|         19 | SAV        | 2001-05-23 |        387.99 |
|          5 | SAV        | 2001-03-12 |        200.00 |
|         21 | CHK        | 2003-07-30 |        125.67 |
|         14 | CHK        | 2002-08-24 |        122.37 |
|         25 | BUS        | 2002-10-01 |          0.00 |
+------------+------------+------------+---------------+
24 rows in set (0.05 sec)
```

Descending sorts are commonly used for ranking queries, such as "show me the top five account balances." MySQL includes a `limit` clause that allows you to sort your data and then discard all but the first *X* rows; see Appendix B for a discussion of the `limit` clause, along with other non-ANSI extensions.

Sorting via Expressions

Sorting your results using column data is all well and good, but sometimes you might need to sort by something that is not stored in the database, and possibly doesn't appear anywhere in your query. You can add an expression to your `order by` clause to handle such situations. For example, perhaps you would like to sort your customer data by the last three digits of the customer's federal ID number (which is either a Social Security number for individuals or a corporate ID for businesses):

```
mysql> SELECT cust_id, cust_type_cd, city, state, fed_id
    -> FROM customer
    -> ORDER BY RIGHT(fed_id, 3);
+---------+--------------+------------+-------+-------------+
| cust_id | cust_type_cd | city       | state | fed_id      |
+---------+--------------+------------+-------+-------------+
|       1 | I            | Lynnfield  | MA    | 111-11-1111 |
|      10 | B            | Salem      | NH    | 04-1111111  |
|       2 | I            | Woburn     | MA    | 222-22-2222 |
|      11 | B            | Wilmington | MA    | 04-2222222  |
|       3 | I            | Quincy     | MA    | 333-33-3333 |
|      12 | B            | Salem      | NH    | 04-3333333  |
|      13 | B            | Quincy     | MA    | 04-4444444  |
|       4 | I            | Waltham    | MA    | 444-44-4444 |
|       5 | I            | Salem      | NH    | 555-55-5555 |
|       6 | I            | Waltham    | MA    | 666-66-6666 |
|       7 | I            | Wilmington | MA    | 777-77-7777 |
```

```
|       8 | I           | Salem      | NH    | 888-88-8888 |
|       9 | I           | Newton     | MA    | 999-99-9999 |
+---------+-------------+------------+-------+-------------+
13 rows in set (0.24 sec)
```

This query uses the built-in function `right()` to extract the last three characters of the `fed_id` column and then sorts the rows based on this value.

Sorting via Numeric Placeholders

If you are sorting using the columns in your `select` clause, you can opt to reference the columns by their *position* in the `select` clause rather than by name. For example, if you want to sort using the second and fifth columns returned by a query, you could do the following:

```
mysql> SELECT emp_id, title, start_date, fname, lname
    -> FROM employee
    -> ORDER BY 2, 5;
+--------+--------------------+------------+----------+-----------+
| emp_id | title              | start_date | fname    | lname     |
+--------+--------------------+------------+----------+-----------+
|     13 | Head Teller        | 2004-05-11 | John     | Blake     |
|      6 | Head Teller        | 2008-03-17 | Helen    | Fleming   |
|     16 | Head Teller        | 2005-03-15 | Theresa  | Markham   |
|     10 | Head Teller        | 2006-07-27 | Paula    | Roberts   |
|      5 | Loan Manager       | 2007-11-14 | John     | Gooding   |
|      4 | Operations Manager | 2006-04-24 | Susan    | Hawthorne |
|      1 | President          | 2005-06-22 | Michael  | Smith     |
|     17 | Teller             | 2006-06-29 | Beth     | Fowler    |
|      9 | Teller             | 2006-05-03 | Jane     | Grossman  |
|     12 | Teller             | 2007-01-08 | Samantha | Jameson   |
|     14 | Teller             | 2006-08-09 | Cindy    | Mason     |
|      8 | Teller             | 2006-12-02 | Sarah    | Parker    |
|     15 | Teller             | 2007-04-01 | Frank    | Portman   |
|      7 | Teller             | 2008-09-15 | Chris    | Tucker    |
|     18 | Teller             | 2006-12-12 | Rick     | Tulman    |
|     11 | Teller             | 2004-10-23 | Thomas   | Ziegler   |
|      3 | Treasurer          | 2005-02-09 | Robert   | Tyler     |
|      2 | Vice President     | 2006-09-12 | Susan    | Barker    |
+--------+--------------------+------------+----------+-----------+
18 rows in set (0.00 sec)
```

You might want to use this feature sparingly, since adding a column to the `select` clause without changing the numbers in the `order by` clause can lead to unexpected results. Personally, I may reference columns positionally when writing ad hoc queries, but I always reference columns by name when writing code.

Test Your Knowledge

The following exercises are designed to strengthen your understanding of the `select` statement and its various clauses. Please see Appendix C for solutions.

Exercise 3-1

Retrieve the employee ID, first name, and last name for all bank employees. Sort by last name and then by first name.

Exercise 3-2

Retrieve the account ID, customer ID, and available balance for all accounts whose status equals `'ACTIVE'` and whose available balance is greater than $2,500.

Exercise 3-3

Write a query against the `account` table that returns the IDs of the employees who opened the accounts (use the `account.open_emp_id` column). Include a single row for each distinct employee.

Exercise 3-4

Fill in the blanks (denoted by <#>) for this multi-data-set query to achieve the results shown here:

```
mysql> SELECT p.product_cd, a.cust_id, a.avail_balance
    -> FROM product p INNER JOIN account <1>
    ->   ON p.product_cd = <2>
    -> WHERE p.<3> = 'ACCOUNT'
    -> ORDER BY <4>, <5>;
+------------+---------+---------------+
| product_cd | cust_id | avail_balance |
+------------+---------+---------------+
| CD         |       1 |       3000.00 |
| CD         |       6 |      10000.00 |
| CD         |       7 |       5000.00 |
| CD         |       9 |       1500.00 |
| CHK        |       1 |       1057.75 |
| CHK        |       2 |       2258.02 |
| CHK        |       3 |       1057.75 |
| CHK        |       4 |        534.12 |
| CHK        |       5 |       2237.97 |
| CHK        |       6 |        122.37 |
| CHK        |       8 |       3487.19 |
| CHK        |       9 |        125.67 |
| CHK        |      10 |      23575.12 |
| CHK        |      12 |      38552.05 |
| MM         |       3 |       2212.50 |
```

```
| MM          |       4 |       5487.09 |
| MM          |       9 |       9345.55 |
| SAV         |       1 |        500.00 |
| SAV         |       2 |        200.00 |
| SAV         |       4 |        767.77 |
| SAV         |       8 |        387.99 |
+-------------+---------+---------------+
21 rows in set (0.09 sec)
```

CHAPTER 4
Filtering

Sometimes you will want to work with every row in a table, such as:

- Purging all data from a table used to stage new data warehouse feeds
- Modifying all rows in a table after a new column has been added
- Retrieving all rows from a message queue table

In cases like these, your SQL statements won't need to have a `where` clause, since you don't need to exclude any rows from consideration. Most of the time, however, you will want to narrow your focus to a subset of a table's rows. Therefore, all the SQL data statements (except the `insert` statement) include an optional `where` clause to house *filter conditions* used to restrict the number of rows acted on by the SQL statement. Additionally, the `select` statement includes a `having` clause in which filter conditions pertaining to grouped data may be included. This chapter explores the various types of filter conditions that you can employ in the `where` clauses of `select`, `update`, and `delete` statements; we explore the use of filter conditions in the `having` clause of a `select` statement in Chapter 8.

Condition Evaluation

A `where` clause may contain one or more *conditions*, separated by the operators `and` and `or`. If multiple conditions are separated only by the `and` operator, then all the conditions must evaluate to `true` for the row to be included in the result set. Consider the following `where` clause:

```
WHERE title = 'Teller' AND start_date < '2007-01-01'
```

Given these two conditions, only tellers who began working for the bank prior to 2007 will be included (or, to look at it another way, any employee who is either not a teller or began working for the bank in 2007 or later will be removed from consideration). Although this example uses only two conditions, no matter how many conditions are in your `where` clause, if they are separated by the `and` operator they must *all* evaluate to `true` for the row to be included in the result set.

If all conditions in the `where` clause are separated by the `or` operator, however, only *one* of the conditions must evaluate to `true` for the row to be included in the result set. Consider the following two conditions:

```
WHERE title = 'Teller' OR start_date < '2007-01-01'
```

There are now various ways for a given `employee` row to be included in the result set:

- The employee is a teller and was employed prior to 2007.
- The employee is a teller and was employed after January 1, 2007.
- The employee is something other than a teller but was employed prior to 2007.

Table 4-1 shows the possible outcomes for a `where` clause containing two conditions separated by the `or` operator.

Table 4-1. Two-condition evaluation using or

Intermediate result	Final result
`WHERE true OR true`	`True`
`WHERE true OR false`	`True`
`WHERE false OR true`	`True`
`WHERE false OR false`	`False`

In the case of the preceding example, the only way for a row to be excluded from the result set is if the employee is not a teller and was employed on or after January 1, 2007.

Using Parentheses

If your `where` clause includes three or more conditions using both the `and` and `or` operators, you should use parentheses to make your intent clear, both to the database server and to anyone else reading your code. Here's a `where` clause that extends the previous example by checking to make sure that the employee is still employed by the bank:

```
WHERE end_date IS NULL
  AND (title = 'Teller' OR start_date < '2007-01-01')
```

There are now three conditions; for a row to make it to the final result set, the first condition must evaluate to `true`, and either the second *or* third condition (or both) must evaluate to `true`. Table 4-2 shows the possible outcomes for this `where` clause.

Table 4-2. Three-condition evaluation using and, or

Intermediate result	Final result
`WHERE true AND (true OR true)`	`True`
`WHERE true AND (true OR false)`	`True`
`WHERE true AND (false OR true)`	`True`
`WHERE true AND (false OR false)`	`False`

Intermediate result	Final result
WHERE false AND (true OR true)	False
WHERE false AND (true OR false)	False
WHERE false AND (false OR true)	False
WHERE false AND (false OR false)	False

As you can see, the more conditions you have in your `where` clause, the more combinations there are for the server to evaluate. In this case, only three of the eight combinations yield a final result of `true`.

Using the not Operator

Hopefully, the previous three-condition example is fairly easy to understand. Consider the following modification, however:

```
WHERE end_date IS NULL
  AND NOT (title = 'Teller' OR start_date < '2007-01-01')
```

Did you spot the change from the previous example? I added the `not` operator after the `and` operator on the second line. Now, instead of looking for nonterminated employees who either are tellers or began working for the bank prior to 2007, I am looking for nonterminated employees who both are nontellers and began working for the bank in 2007 or later. Table 4-3 shows the possible outcomes for this example.

Table 4-3. Three-condition evaluation using and, or, and not

Intermediate result	Final result
WHERE true AND NOT (true OR true)	False
WHERE true AND NOT (true OR false)	False
WHERE true AND NOT (false OR true)	False
WHERE true AND NOT (false OR false)	True
WHERE false AND NOT (true OR true)	False
WHERE false AND NOT (true OR false)	False
WHERE false AND NOT (false OR true)	False
WHERE false AND NOT (false OR false)	False

While it is easy for the database server to handle, it is typically difficult for a person to evaluate a `where` clause that includes the `not` operator, which is why you won't encounter it very often. In this case, you can rewrite the `where` clause to avoid using the `not` operator:

```
WHERE end_date IS NULL
  AND title != 'Teller' AND start_date >= '2007-01-01'
```

While I'm sure that the server doesn't have a preference, you probably have an easier time understanding this version of the `where` clause.

Building a Condition

Now that you have seen how the server evaluates multiple conditions, let's take a step back and look at what comprises a single condition. A condition is made up of one or more *expressions* coupled with one or more *operators*. An expression can be any of the following:

- A number
- A column in a table or view
- A string literal, such as `'Teller'`
- A built-in function, such as `concat('Learning', ' ', 'SQL')`
- A subquery
- A list of expressions, such as `('Teller', 'Head Teller', 'Operations Manager')`

The operators used within conditions include:

- Comparison operators, such as `=`, `!=`, `<`, `>`, `<>`, `LIKE`, `IN`, and `BETWEEN`
- Arithmetic operators, such as `+`, `-`, `*`, and `/`

The following section demonstrates how you can combine these expressions and operators to manufacture the various types of conditions.

Condition Types

There are many different ways to filter out unwanted data. You can look for specific values, sets of values, or ranges of values to include or exclude, or you can use various pattern-searching techniques to look for partial matches when dealing with string data. The next four subsections explore each of these condition types in detail.

Equality Conditions

A large percentage of the filter conditions that you write or come across will be of the form '*`column = expression`*' as in:

```
title = 'Teller'
fed_id = '111-11-1111'
amount = 375.25
dept_id = (SELECT dept_id FROM department WHERE name = 'Loans')
```

Conditions such as these are called *equality conditions* because they equate one expression to another. The first three examples equate a column to a literal (two strings and a number), and the fourth example equates a column to the value returned from

a subquery. The following query uses two equality conditions; one in the `on` clause (a join condition), and the other in the `where` clause (a filter condition):

```
mysql> SELECT pt.name product_type, p.name product
    -> FROM product p INNER JOIN product_type pt
    ->   ON p.product_type_cd = pt.product_type_cd
    -> WHERE pt.name = 'Customer Accounts';
+------------------+-----------------------+
| product_type     | product               |
+------------------+-----------------------+
| Customer Accounts | certificate of deposit |
| Customer Accounts | checking account      |
| Customer Accounts | money market account  |
| Customer Accounts | savings account       |
+------------------+-----------------------+
4 rows in set (0.08 sec)
```

This query shows all products that are *customer account* types.

Inequality conditions

Another fairly common type of condition is the *inequality condition*, which asserts that two expressions are *not* equal. Here's the previous query with the filter condition in the `where` clause changed to an inequality condition:

```
mysql> SELECT pt.name product_type, p.name product
    -> FROM product p INNER JOIN product_type pt
    ->   ON p.product_type_cd = pt.product_type_cd
    -> WHERE pt.name <> 'Customer Accounts';
+-------------------------------+-------------------------+
| product_type                  | product                 |
+-------------------------------+-------------------------+
| Individual and Business Loans | auto loan               |
| Individual and Business Loans | business line of credit |
| Individual and Business Loans | home mortgage           |
| Individual and Business Loans | small business loan     |
+-------------------------------+-------------------------+
4 rows in set (0.00 sec)
```

This query shows all products that are *not* customer account types. When building inequality conditions, you may choose to use either the `!=` or `<>` operator.

Data modification using equality conditions

Equality/inequality conditions are commonly used when modifying data. For example, let's say that the bank has a policy of removing old account rows once per year. Your task is to remove rows from the `account` table that were closed in 2002. Here's one way to tackle it:

```
DELETE FROM account
WHERE status = 'CLOSED' AND YEAR(close_date) = 2002;
```

This statement includes two equality conditions: one to find only closed accounts, and another to check for those accounts closed in 2002.

When crafting examples of `delete` and `update` statements, I try to write each statement such that no rows are modified. That way, when you execute the statements, your data will remain unchanged, and your output from `select` statements will always match that shown in this book.

Since MySQL sessions are in auto-commit mode by default (see Chapter 12), you would not be able to roll back (undo) any changes made to the example data if one of my statements modified the data. You may, of course, do whatever you want with the example data, including wiping it clean and rerunning the scripts I have provided, but I try to leave it intact.

Range Conditions

Along with checking that an expression is equal to (or not equal to) another expression, you can build conditions that check whether an expression falls within a certain range. This type of condition is common when working with numeric or temporal data. Consider the following query:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date < '2007-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     11 | Thomas  | Ziegler   | 2004-10-23 |
|     13 | John    | Blake     | 2004-05-11 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
13 rows in set (0.15 sec)
```

This query finds all employees hired prior to 2007. Along with specifying an upper limit for the start date, you may also want to specify a lower range for the start date:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date < '2007-01-01'
    ->   AND start_date >= '2005-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
```

```
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
11 rows in set (0.00 sec)
```

This version of the query retrieves all employees hired in 2005 or 2006.

The between operator

When you have *both* an upper and lower limit for your range, you may choose to use a single condition that utilizes the `between` operator rather than using two separate conditions, as in:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date BETWEEN '2005-01-01' AND '2007-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2005-06-22 |
|      2 | Susan   | Barker    | 2006-09-12 |
|      3 | Robert  | Tyler     | 2005-02-09 |
|      4 | Susan   | Hawthorne | 2006-04-24 |
|      8 | Sarah   | Parker    | 2006-12-02 |
|      9 | Jane    | Grossman  | 2006-05-03 |
|     10 | Paula   | Roberts   | 2006-07-27 |
|     14 | Cindy   | Mason     | 2006-08-09 |
|     16 | Theresa | Markham   | 2005-03-15 |
|     17 | Beth    | Fowler    | 2006-06-29 |
|     18 | Rick    | Tulman    | 2006-12-12 |
+--------+---------+-----------+------------+
11 rows in set (0.03 sec)
```

When using the `between` operator, there are a couple of things to keep in mind. You should always specify the lower limit of the range first (after `between`) and the upper limit of the range second (after `and`). Here's what happens if you mistakenly specify the upper limit first:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date BETWEEN '2007-01-01' AND '2005-01-01';
Empty set (0.00 sec)
```

As you can see, no data is returned. This is because the server is, in effect, generating two conditions from your single condition using the `<=` and `>=` operators, as in:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> FROM employee
    -> WHERE start_date >= '2007-01-01'
    ->   AND start_date <= '2005-01-01';
Empty set (0.00 sec)
```

Since it is impossible to have a date that is *both* greater than January 1, 2007 and less than January 1, 2005, the query returns an empty set. This brings me to the second pitfall when using `between`, which is to remember that your upper and lower limits are *inclusive*, meaning that the values you provide are included in the range limits. In this case, I want to specify `2005-01-01` as the lower end of the range and `2006-12-31` as the upper end, rather than `2007-01-01`. Even though there probably weren't any employees who started working for the bank on New Year's Day 2007, it is best to specify exactly what you want.

Along with dates, you can also build conditions to specify ranges of numbers. Numeric ranges are fairly easy to grasp, as demonstrated by the following:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE avail_balance BETWEEN 3000 AND 5000;
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          3 | CD         |       1 |       3000.00 |
|         17 | CD         |       7 |       5000.00 |
|         18 | CHK        |       8 |       3487.19 |
+------------+------------+---------+---------------+
3 rows in set (0.10 sec)
```

All accounts with between $3,000 and $5,000 of an available balance are returned. Again, make sure that you specify the lower amount first.

String ranges

While ranges of dates and numbers are easy to understand, you can also build conditions that search for ranges of strings, which are a bit harder to visualize. Say, for example, you are searching for customers having a Social Security number that falls within a certain range. The format for a Social Security number is "XXX-XX-XXXX," where X is a number from 0 to 9, and you want to find every customer whose Social Security number lies between "500-00-0000" and "999-99-9999." Here's what the statement would look like:

```
mysql> SELECT cust_id, fed_id
    -> FROM customer
    -> WHERE cust_type_cd = 'I'
    ->   AND fed_id BETWEEN '500-00-0000' AND '999-99-9999';
+---------+-------------+
| cust_id | fed_id      |
+---------+-------------+
|       5 | 555-55-5555 |
|       6 | 666-66-6666 |
```

```
|       7 | 777-77-7777 |
|       8 | 888-88-8888 |
|       9 | 999-99-9999 |
+---------+-------------+
5 rows in set (0.01 sec)
```

To work with string ranges, you need to know the order of the characters within your character set (the order in which the characters within a character set are sorted is called a *collation*).

Membership Conditions

In some cases, you will not be restricting an expression to a single value or range of values, but rather to a finite set of values. For example, you might want to locate all accounts whose product code is either `'CHK'`, `'SAV'`, `'CD'`, or `'MM'`:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd = 'CHK' OR product_cd = 'SAV'
    ->   OR product_cd = 'CD' OR product_cd = 'MM';
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          1 | CHK        |       1 |       1057.75 |
|          2 | SAV        |       1 |        500.00 |
|          3 | CD         |       1 |       3000.00 |
|          4 | CHK        |       2 |       2258.02 |
|          5 | SAV        |       2 |        200.00 |
|          7 | CHK        |       3 |       1057.75 |
|          8 | MM         |       3 |       2212.50 |
|         10 | CHK        |       4 |        534.12 |
|         11 | SAV        |       4 |        767.77 |
|         12 | MM         |       4 |       5487.09 |
|         13 | CHK        |       5 |       2237.97 |
|         14 | CHK        |       6 |        122.37 |
|         15 | CD         |       6 |      10000.00 |
|         17 | CD         |       7 |       5000.00 |
|         18 | CHK        |       8 |       3487.19 |
|         19 | SAV        |       8 |        387.99 |
|         21 | CHK        |       9 |        125.67 |
|         22 | MM         |       9 |       9345.55 |
|         23 | CD         |       9 |       1500.00 |
|         24 | CHK        |      10 |      23575.12 |
|         28 | CHK        |      12 |      38552.05 |
+------------+------------+---------+---------------+
21 rows in set (0.28 sec)
```

While this `where` clause (four conditions `or`'d together) wasn't too tedious to generate, imagine if the set of expressions contained 10 or 20 members. For these situations, you can use the `in` operator instead:

```
SELECT account_id, product_cd, cust_id, avail_balance
FROM account
WHERE product_cd IN ('CHK','SAV','CD','MM');
```

With the `in` operator, you can write a single condition no matter how many expressions are in the set.

Using subqueries

Along with writing your own set of expressions, such as `('CHK','SAV','CD','MM')`, you can use a subquery to generate a set for you on the fly. For example, all four product types used in the previous query have a `product_type_cd` of `'ACCOUNT'`, so why not use a subquery against the `product` table to retrieve the four product codes instead of explicitly naming them:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd IN (SELECT product_cd FROM product
    ->   WHERE product_type_cd = 'ACCOUNT');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          3 | CD         |       1 |       3000.00 |
|         15 | CD         |       6 |      10000.00 |
|         17 | CD         |       7 |       5000.00 |
|         23 | CD         |       9 |       1500.00 |
|          1 | CHK        |       1 |       1057.75 |
|          4 | CHK        |       2 |       2258.02 |
|          7 | CHK        |       3 |       1057.75 |
|         10 | CHK        |       4 |        534.12 |
|         13 | CHK        |       5 |       2237.97 |
|         14 | CHK        |       6 |        122.37 |
|         18 | CHK        |       8 |       3487.19 |
|         21 | CHK        |       9 |        125.67 |
|         24 | CHK        |      10 |      23575.12 |
|         28 | CHK        |      12 |      38552.05 |
|          8 | MM         |       3 |       2212.50 |
|         12 | MM         |       4 |       5487.09 |
|         22 | MM         |       9 |       9345.55 |
|          2 | SAV        |       1 |        500.00 |
|          5 | SAV        |       2 |        200.00 |
|         11 | SAV        |       4 |        767.77 |
|         19 | SAV        |       8 |        387.99 |
+------------+------------+---------+---------------+
21 rows in set (0.11 sec)
```

The subquery returns a set of four values, and the main query checks to see whether the value of the `product_cd` column can be found in the set that the subquery returned.

Using not in

Sometimes you want to see whether a particular expression exists within a set of expressions, and sometimes you want to see whether the expression does *not* exist. For these situations, you can use the `not in` operator:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd NOT IN ('CHK','SAV','CD','MM');
+------------+------------+---------+--------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+--------------+
|         25 | BUS        |      10 |          0.00 |
|         27 | BUS        |      11 |       9345.55 |
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+--------------+
3 rows in set (0.09 sec)
```

This query finds all accounts that are *not* checking, savings, certificate of deposit, or money market accounts.

Matching Conditions

So far, you have been introduced to conditions that identify an exact string, a range of strings, or a set of strings; the final condition type deals with partial string matches. You may, for example, want to find all employees whose last name begins with *T*. You could use a built-in function to strip off the first letter of the `lname` column, as in:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE LEFT(lname, 1) = 'T';
+--------+--------+--------+
| emp_id | fname  | lname  |
+--------+--------+--------+
|      3 | Robert | Tyler  |
|      7 | Chris  | Tucker |
|     18 | Rick   | Tulman |
+--------+--------+--------+
3 rows in set (0.01 sec)
```

While the built-in function `left()` does the job, it doesn't give you much flexibility. Instead, you can use wildcard characters to build search expressions, as demonstrated in the next section.

Using wildcards

When searching for partial string matches, you might be interested in:

- Strings beginning/ending with a certain character
- Strings beginning/ending with a substring
- Strings containing a certain character anywhere within the string
- Strings containing a substring anywhere within the string
- Strings with a specific format, regardless of individual characters

You can build search expressions to identify these and many other partial string matches by using the wildcard characters shown in Table 4-4.

Table 4-4. Wildcard characters

Wildcard character	Matches
_	Exactly one character
%	Any number of characters (including 0)

The underscore character takes the place of a single character, while the percent sign can take the place of a variable number of characters. When building conditions that utilize search expressions, you use the `like` operator, as in:

```
mysql> SELECT lname
    -> FROM employee
    -> WHERE lname LIKE '_a%e%';
+-----------+
| lname     |
+-----------+
| Barker    |
| Hawthorne |
| Parker    |
| Jameson   |
+-----------+
4 rows in set (0.00 sec)
```

The search expression in the previous example specifies strings containing an *a* in the second position and followed by an *e* at any other position in the string (including the last position). Table 4-5 shows some more search expressions and their interpretations.

Table 4-5. Sample search expressions

Search expression	Interpretation
F%	Strings beginning with *F*
%t	Strings ending with *t*
%bas%	Strings containing the substring 'bas'
_ _t_	Four-character strings with a *t* in the third position
_ _ _-_ _-_ _ _ _	11-character strings with dashes in the fourth and seventh positions

You could use the last example in Table 4-5 to find customers whose federal ID matches the format used for Social Security numbers, as in:

```
mysql> SELECT cust_id, fed_id
    -> FROM customer
    -> WHERE fed_id LIKE '___-__-____';
+---------+-------------+
| cust_id | fed_id      |
+---------+-------------+
|       1 | 111-11-1111 |
|       2 | 222-22-2222 |
|       3 | 333-33-3333 |
|       4 | 444-44-4444 |
|       5 | 555-55-5555 |
```

```
|       6 | 666-66-6666 |
|       7 | 777-77-7777 |
|       8 | 888-88-8888 |
|       9 | 999-99-9999 |
+---------+-------------+
9 rows in set (0.02 sec)
```

The wildcard characters work fine for building simple search expressions; if your needs are a bit more sophisticated, however, you can use multiple search expressions, as demonstrated by the following:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname LIKE 'F%' OR lname LIKE 'G%';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

This query finds all employees whose last name begins with *F* or *G*.

Using regular expressions

If you find that the wildcard characters don't provide enough flexibility, you can use regular expressions to build search expressions. A regular expression is, in essence, a search expression on steroids. If you are new to SQL but have coded using programming languages such as Perl, then you might already be intimately familiar with regular expressions. If you have never used regular expressions, then you may want to consult Jeffrey E.F. Friedl's *Mastering Regular Expressions (http://oreilly.com/catalog/9780596528126/)* (O'Reilly), since it is far too large a topic to try to cover in this book.

Here's what the previous query (find all employees whose last name starts with *F* or *G*) would look like using the MySQL implementation of regular expressions:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname REGEXP '^[FG]';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

The `regexp` operator takes a regular expression (`'^[FG]'` in this example) and applies it to the expression on the lefthand side of the condition (the column `lname`). The query

now contains a single condition using a regular expression rather than two conditions using wildcard characters.

Oracle Database and Microsoft SQL Server also support regular expressions. With Oracle Database, you would use the `regexp_like` function instead of the `regexp` operator shown in the previous example, whereas SQL Server allows regular expressions to be used with the `like` operator.

Null: That Four-Letter Word

I put it off as long as I could, but it's time to broach a topic that tends to be met with fear, uncertainty, and dread: the `null` value. `Null` is the absence of a value; before an employee is terminated, for example, her `end_date` column in the `employee` table should be `null`. There is no value that can be assigned to the `end_date` column that would make sense in this situation. `Null` is a bit slippery, however, as there are various flavors of `null`:

Not applicable
: Such as the employee ID column for a transaction that took place at an ATM machine

Value not yet known
: Such as when the federal ID is not known at the time a customer row is created

Value undefined
: Such as when an account is created for a product that has not yet been added to the database

Some theorists argue that there should be a different expression to cover each of these (and more) situations, but most practitioners would agree that having multiple `null` values would be far too confusing.

When working with `null`, you should remember:

- An expression can *be* null, but it can never *equal* null.
- Two nulls are never equal to each other.

To test whether an expression is `null`, you need to use the `is null` operator, as demonstrated by the following:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id IS NULL;
+--------+---------+-------+-----------------+
| emp_id | fname   | lname | superior_emp_id |
+--------+---------+-------+-----------------+
|      1 | Michael | Smith |            NULL |
+--------+---------+-------+-----------------+
1 row in set (0.00 sec)
```

This query returns all employees who do not have a boss (wouldn't that be nice?). Here's the same query using `= null` instead of `is null`:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id = NULL;
Empty set (0.01 sec)
```

As you can see, the query parses and executes but does not return any rows. This is a common mistake made by inexperienced SQL programmers, and the database server will not alert you to your error, so be careful when constructing conditions that test for `null`.

If you want to see whether a value has been assigned to a column, you can use the `is not null` operator, as in:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id IS NOT NULL;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|      7 | Chris    | Tucker    |               6 |
|      8 | Sarah    | Parker    |               6 |
|      9 | Jane     | Grossman  |               6 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
17 rows in set (0.00 sec)
```

This version of the query returns the other 17 employees who, unlike Michael Smith, have a boss.

Before putting `null` aside for a while, it would be helpful to investigate one more potential pitfall. Suppose that you have been asked to identify all employees who are *not* managed by Helen Fleming (whose employee ID is 6). Your first instinct might be to do the following:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id != 6;
```

```
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
14 rows in set (0.00 sec)
```

While it is true that these 14 employees do not work for Helen Fleming, if you look carefully at the data, you will see that there is one more employee who doesn't work for Helen who is not listed here. That employee is Michael Smith, and his `superior_emp_id` column is `null` (because he's the big cheese). To answer the question correctly, therefore, you need to account for the possibility that some rows might contain a `null` in the `superior_emp_id` column:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
    -> FROM employee
    -> WHERE superior_emp_id != 6 OR superior_emp_id IS NULL;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      1 | Michael  | Smith     |            NULL |
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
15 rows in set (0.00 sec)
```

The result set now includes all 15 employees who don't work for Helen. When working with a database that you are not familiar with, it is a good idea to find out which columns

in a table allow nulls so that you can take appropriate measures with your filter conditions to keep data from slipping through the cracks.

Test Your Knowledge

The following exercises test your understanding of filter conditions. Please see Appendix C for solutions.

The following transaction data is used for the first two exercises:

Txn_id	Txn_date	Account_id	Txn_type_cd	Amount
1	2005-02-22	101	CDT	1000.00
2	2005-02-23	102	CDT	525.75
3	2005-02-24	101	DBT	100.00
4	2005-02-24	103	CDT	55
5	2005-02-25	101	DBT	50
6	2005-02-25	103	DBT	25
7	2005-02-25	102	CDT	125.37
8	2005-02-26	103	DBT	10
9	2005-02-27	101	CDT	75

Exercise 4-1

Which of the transaction IDs would be returned by the following filter conditions?

```
txn_date < '2005-02-26' AND (txn_type_cd = 'DBT' OR amount > 100)
```

Exercise 4-2

Which of the transaction IDs would be returned by the following filter conditions?

```
account_id IN (101,103) AND NOT (txn_type_cd = 'DBT' OR amount > 100)
```

Exercise 4-3

Construct a query that retrieves all accounts opened in 2002.

Exercise 4-4

Construct a query that finds all nonbusiness customers whose last name contains an *a* in the second position and an *e* anywhere after the *a*.

CHAPTER 5

Querying Multiple Tables

Back in Chapter 2, I demonstrated how related concepts are broken into separate pieces through a process known as normalization. The end result of this exercise was two tables: `person` and `favorite_food`. If, however, you want to generate a single report showing a person's name, address, *and* favorite foods, you will need a mechanism to bring the data from these two tables back together again; this mechanism is known as a *join*, and this chapter concentrates on the simplest and most common join, the *inner join*. Chapter 10 demonstrates all of the different join types.

What Is a Join?

Queries against a single table are certainly not rare, but you will find that most of your queries will require two, three, or even more tables. To illustrate, let's look at the definitions for the `employee` and `department` tables and then define a query that retrieves data from both tables:

```
mysql> DESC employee;
+--------------------+----------------------+------+-----+---------+
| Field              | Type                 | Null | Key | Default |
+--------------------+----------------------+------+-----+---------+
| emp_id             | smallint(5) unsigned | NO   | PRI | NULL    |
| fname              | varchar(20)          | NO   |     | NULL    |
| lname              | varchar(20)          | NO   |     | NULL    |
| start_date         | date                 | NO   |     | NULL    |
| end_date           | date                 | YES  |     | NULL    |
| superior_emp_id    | smallint(5) unsigned | YES  | MUL | NULL    |
| dept_id            | smallint(5) unsigned | YES  | MUL | NULL    |
| title              | varchar(20)          | YES  |     | NULL    |
| assigned_branch_id | smallint(5) unsigned | YES  | MUL | NULL    |
+--------------------+----------------------+------+-----+---------+
9 rows in set (0.11 sec)

 mysql> DESC department;
+---------+----------------------+------+-----+---------+
| Field   | Type                 | Null | Key | Default |
+---------+----------------------+------+-----+---------+
| dept_id | smallint(5) unsigned | No   | PRI | NULL    |
```

```
| name    | varchar(20)          | No   |     | NULL    |
+---------+----------------------+------+-----+---------+
2 rows in set (0.03 sec)
```

Let's say you want to retrieve the first and last names of each employee along with the name of the department to which each employee is assigned. Your query will therefore need to retrieve the `employee.fname`, `employee.lname`, and `department.name` columns. But how can you retrieve data from both tables in the same query? The answer lies in the `employee.dept_id` column, which holds the ID of the department to which each employee is assigned (in more formal terms, the `employee.dept_id` column is the *foreign key* to the `department` table). The query, which you will see shortly, instructs the server to use the `employee.dept_id` column as the *bridge* between the `employee` and `department` tables, thereby allowing columns from both tables to be included in the query's result set. This type of operation is known as a join.

Cartesian Product

The easiest way to start is to put the `employee` and `department` tables into the `from` clause of a query and see what happens. Here's a query that retrieves the employee's first and last names along with the department name, with a `from` clause naming both tables separated by the `join` keyword:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e JOIN department d;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Operations     |
| Michael  | Smith     | Loans          |
| Michael  | Smith     | Administration |
| Susan    | Barker    | Operations     |
| Susan    | Barker    | Loans          |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Operations     |
| Robert   | Tyler     | Loans          |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| Susan    | Hawthorne | Loans          |
| Susan    | Hawthorne | Administration |
| John     | Gooding   | Operations     |
| John     | Gooding   | Loans          |
| John     | Gooding   | Administration |
| Helen    | Fleming   | Operations     |
| Helen    | Fleming   | Loans          |
| Helen    | Fleming   | Administration |
| Chris    | Tucker    | Operations     |
| Chris    | Tucker    | Loans          |
| Chris    | Tucker    | Administration |
| Sarah    | Parker    | Operations     |
| Sarah    | Parker    | Loans          |
| Sarah    | Parker    | Administration |
| Jane     | Grossman  | Operations     |
```

```
| Jane     | Grossman  | Loans          |
| Jane     | Grossman  | Administration |
| Paula    | Roberts   | Operations     |
| Paula    | Roberts   | Loans          |
| Paula    | Roberts   | Administration |
| Thomas   | Ziegler   | Operations     |
| Thomas   | Ziegler   | Loans          |
| Thomas   | Ziegler   | Administration |
| Samantha | Jameson   | Operations     |
| Samantha | Jameson   | Loans          |
| Samantha | Jameson   | Administration |
| John     | Blake     | Operations     |
| John     | Blake     | Loans          |
| John     | Blake     | Administration |
| Cindy    | Mason     | Operations     |
| Cindy    | Mason     | Loans          |
| Cindy    | Mason     | Administration |
| Frank    | Portman   | Operations     |
| Frank    | Portman   | Loans          |
| Frank    | Portman   | Administration |
| Theresa  | Markham   | Operations     |
| Theresa  | Markham   | Loans          |
| Theresa  | Markham   | Administration |
| Beth     | Fowler    | Operations     |
| Beth     | Fowler    | Loans          |
| Beth     | Fowler    | Administration |
| Rick     | Tulman    | Operations     |
| Rick     | Tulman    | Loans          |
| Rick     | Tulman    | Administration |
+----------+-----------+----------------+
54 rows in set (0.23 sec)
```

Hmmm...there are only 18 employees and 3 different departments, so how did the result set end up with 54 rows? Looking more closely, you can see that the set of 18 employees is repeated three times, with all the data identical except for the department name. Because the query didn't specify *how* the two tables should be joined, the database server generated the *Cartesian product*, which is *every* permutation of the two tables (18 employees × 3 departments = 54 permutations). This type of join is known as a *cross join*, and it is rarely used (on purpose, at least). Cross joins are one of the join types that we study in Chapter 10.

Inner Joins

To modify the previous query so that only 18 rows are included in the result set (one for each employee), you need to describe how the two tables are related. Earlier, I showed that the `employee.dept_id` column serves as the link between the two tables, so this information needs to be added to the `on` subclause of the `from` clause:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e JOIN department d
    ->   ON e.dept_id = d.dept_id;
```

```
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| John     | Gooding   | Loans          |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
+----------+-----------+----------------+
18 rows in set (0.00 sec)
```

Instead of 54 rows, you now have the expected 18 rows due to the addition of the `on` subclause, which instructs the server to join the `employee` and `department` tables by using the `dept_id` column to traverse from one table to the other. For example, Susan Hawthorne's row in the `employee` table contains a value of `1` in the `dept_id` column (not shown in the example). The server uses this value to look up the row in the `department` table having a value of `1` in its `dept_id` column and then retrieves the value `'Operations'` from the `name` column in that row.

If a value exists for the `dept_id` column in one table but *not* the other, then the join fails for the rows containing that value and those rows are excluded from the result set. This type of join is known as an *inner join*, and it is the most commonly used type of join. To clarify, if the `department` table contains a fourth row for the marketing department, but no employees have been assigned to that department, then the marketing department would not be included in the result set. Likewise, if some of the employees had been assigned to department ID 99, which doesn't exist in the `department` table, then these employees would be left out of the result set. If you want to include all rows from one table or the other regardless of whether a match exists, you need to specify an *outer join*, but we cover this later in the book.

In the previous example, I did not specify in the `from` clause which type of join to use. However, when you wish to join two tables using an inner join, you should explicitly specify this in your `from` clause; here's the same example, with the addition of the join type (note the keyword `INNER`):

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e INNER JOIN department d
```

```
    ->   ON e.dept_id = d.dept_id;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| John     | Gooding   | Loans          |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
+----------+-----------+----------------+
18 rows in set (0.00 sec)
```

If you do not specify the type of join, then the server will do an inner join by default. As you will see later in the book, however, there are several types of joins, so you should get in the habit of specifying the exact type of join that you require.

If the names of the columns used to join the two tables are identical, which is true in the previous query, you can use the `using` subclause instead of the `on` subclause, as in:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e INNER JOIN department d
    ->   USING (dept_id);
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| John     | Gooding   | Loans          |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
```

```
| Rick     | Tulman    | Operations     |
+----------+-----------+----------------+
18 rows in set (0.01 sec)
```

Since `using` is a shorthand notation that you can use in only a specific situation, I prefer always to use the `on` subclause to avoid confusion.

The ANSI Join Syntax

The notation used throughout this book for joining tables was introduced in the SQL92 version of the ANSI SQL standard. All the major databases (Oracle Database, Microsoft SQL Server, MySQL, IBM DB2 Universal Database, and Sybase Adaptive Server) have adopted the SQL92 join syntax. Because most of these servers have been around since before the release of the SQL92 specification, they all include an older join syntax as well. For example, all these servers would understand the following variation of the previous query:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e, department d
    -> WHERE e.dept_id = d.dept_id;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Operations     |
| John     | Gooding   | Loans          |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
+----------+-----------+----------------+
18 rows in set (0.01 sec)
```

This older method of specifying joins does not include the `on` subclause; instead, tables are named in the `from` clause separated by commas, and join conditions are included in the `where` clause. While you may decide to ignore the SQL92 syntax in favor of the older join syntax, the ANSI join syntax has the following advantages:

- Join conditions and filter conditions are separated into two different clauses (the `on` subclause and the `where` clause, respectively), making a query easier to understand.

- The join conditions for each pair of tables are contained in their own `on` clause, making it less likely that part of a join will be mistakenly omitted.
- Queries that use the SQL92 join syntax are portable across database servers, whereas the older syntax is slightly different across the different servers.

The benefits of the SQL92 join syntax are easier to identify for complex queries that include both join and filter conditions. Consider the following query, which returns all accounts opened by experienced tellers (hired prior to 2007) currently assigned to the Woburn branch:

```
mysql> SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
    -> FROM account a, branch b, employee e
    -> WHERE a.open_emp_id = e.emp_id
    ->   AND e.start_date < '2007-01-01'
    ->   AND e.assigned_branch_id = b.branch_id
    ->   AND (e.title = 'Teller' OR e.title = 'Head Teller')
    ->   AND b.name = 'Woburn Branch';
+------------+---------+------------+------------+
| account_id | cust_id | open_date  | product_cd |
+------------+---------+------------+------------+
|          1 |       1 | 2000-01-15 | CHK        |
|          2 |       1 | 2000-01-15 | SAV        |
|          3 |       1 | 2004-06-30 | CD         |
|          4 |       2 | 2001-03-12 | CHK        |
|          5 |       2 | 2001-03-12 | SAV        |
|         17 |       7 | 2004-01-12 | CD         |
|         27 |      11 | 2004-03-22 | BUS        |
+------------+---------+------------+------------+
7 rows in set (0.00 sec)
```

With this query, it is not so easy to determine which conditions in the `where` clause are join conditions and which are filter conditions. It is also not readily apparent which type of join is being employed (to identify the type of join, you would need to look closely at the join conditions in the `where` clause to see whether any special characters are employed), nor is it easy to determine whether any join conditions have been mistakenly left out. Here's the same query using the SQL92 join syntax:

```
mysql> SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
    -> FROM account a INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    ->   INNER JOIN branch b
    ->   ON e.assigned_branch_id = b.branch_id
    -> WHERE e.start_date < '2007-01-01'
    ->   AND (e.title = 'Teller' OR e.title = 'Head Teller')
    ->   AND b.name = 'Woburn Branch';
+------------+---------+------------+------------+
| account_id | cust_id | open_date  | product_cd |
+------------+---------+------------+------------+
|          1 |       1 | 2000-01-15 | CHK        |
|          2 |       1 | 2000-01-15 | SAV        |
|          3 |       1 | 2004-06-30 | CD         |
|          4 |       2 | 2001-03-12 | CHK        |
|          5 |       2 | 2001-03-12 | SAV        |
```

```
|         17 |       7 | 2004-01-12 | CD         |
|         27 |      11 | 2004-03-22 | BUS        |
+------------+---------+------------+------------+
7 rows in set (0.05 sec)
```

Hopefully, you will agree that the version using SQL92 join syntax is easier to understand.

Joining Three or More Tables

Joining three tables is similar to joining two tables, but with one slight wrinkle. With a two-table join, there are two tables and one join type in the `from` clause, and a single `on` subclause to define how the tables are joined. With a three-table join, there are three tables and two join types in the `from` clause, and two `on` subclauses. Here's another example of a query with a two-table join:

```
mysql> SELECT a.account_id, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+
| account_id | fed_id     |
+------------+------------+
|         24 | 04-1111111 |
|         25 | 04-1111111 |
|         27 | 04-2222222 |
|         28 | 04-3333333 |
|         29 | 04-4444444 |
+------------+------------+
5 rows in set (0.15 sec)
```

This query, which returns the account ID and federal tax number for all business accounts, should look fairly straightforward by now. If, however, you add the `employee` table to the query to also retrieve the name of the teller who opened each account, it looks as follows:

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+---------+---------+
| account_id | fed_id     | fname   | lname   |
+------------+------------+---------+---------+
|         24 | 04-1111111 | Theresa | Markham |
|         25 | 04-1111111 | Theresa | Markham |
|         27 | 04-2222222 | Paula   | Roberts |
|         28 | 04-3333333 | Theresa | Markham |
|         29 | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.00 sec)
```

Now three tables, two join types, and two `on` subclauses are listed in the `from` clause, so things have gotten quite a bit busier. At first glance, the order in which the tables are named might cause you to think that the `employee` table is being joined to the `customer` table, since the `account` table is named first, followed by the `customer` table, and then the `employee` table. If you switch the order in which the first two tables appear, however, you will get the exact same results:

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM customer c INNER JOIN account a
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+---------+---------+
| account_id | fed_id     | fname   | lname   |
+------------+------------+---------+---------+
|         24 | 04-1111111 | Theresa | Markham |
|         25 | 04-1111111 | Theresa | Markham |
|         27 | 04-2222222 | Paula   | Roberts |
|         28 | 04-3333333 | Theresa | Markham |
|         29 | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.09 sec)
```

The `customer` table is now listed first, followed by the `account` table and then the `employee` table. Since the `on` subclauses haven't changed, the results are the same. For the sake of completeness, here's the same query one last time, but with the table order completely reversed (`employee` to `account` to `customer`):

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM employee e INNER JOIN account a
    ->   ON e.emp_id = a.open_emp_id
    ->   INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+---------+---------+
| account_id | fed_id     | fname   | lname   |
+------------+------------+---------+---------+
|         24 | 04-1111111 | Theresa | Markham |
|         25 | 04-1111111 | Theresa | Markham |
|         27 | 04-2222222 | Paula   | Roberts |
|         28 | 04-3333333 | Theresa | Markham |
|         29 | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.00 sec)
```

Does Join Order Matter?

If you are confused about why all three versions of the `account`/`employee`/`customer` query yield the same results, keep in mind that SQL is a nonprocedural language, meaning that you describe what you want to retrieve and which database objects need to be involved, but it is up to the database server to determine how best to execute your query. Using statistics gathered from your database objects, the server must pick one of three tables as a starting point (the chosen table is thereafter known as the *driving table*), and then decide in which order to join the remaining tables. Therefore, the order in which tables appear in your `from` clause is not significant.

If, however, you believe that the tables in your query should always be joined in a particular order, you can place the tables in the desired order and then specify the keyword `STRAIGHT_JOIN` in MySQL, request the `FORCE ORDER` option in SQL Server, or use either the `ORDERED` or the `LEADING` optimizer hint in Oracle Database. For example, to tell the MySQL server to use the `customer` table as the driving table and to then join the `account` and `employee` tables, you could do the following:

```
mysql> SELECT STRAIGHT_JOIN a.account_id, c.fed_id, e.fname, e.lname
    -> FROM customer c INNER JOIN account a
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
```

One way to think of a query that uses three or more tables is as a snowball rolling down a hill. The first two tables get the ball rolling, and each subsequent table gets tacked on to the snowball as it heads downhill. You can think of the snowball as the *intermediate result set*, which is picking up more and more columns as subsequent tables are joined. Therefore, the `employee` table is not really being joined to the `account` table, but rather the intermediate result set created when the `customer` and `account` tables were joined. (In case you were wondering why I chose a snowball analogy, I wrote this chapter in the midst of a New England winter: 110 inches so far, and more coming tomorrow. Oh joy.)

Using Subqueries As Tables

You have already seen several examples of queries that use three tables, but there is one variation worth mentioning: what to do if some of the data sets are generated by subqueries. Subqueries is the focus of Chapter 9, but I already introduced the concept of a subquery in the `from` clause in the previous chapter. Here's another version of an earlier query (find all accounts opened by experienced tellers currently assigned to the Woburn branch) that joins the `account` table to subqueries against the `branch` and `employee` tables:

```
1 SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
2 FROM account a INNER JOIN
3   (SELECT emp_id, assigned_branch_id
```

```
4     FROM employee
5     WHERE start_date < '2007-01-01'
6       AND (title = 'Teller' OR title = 'Head Teller')) e
7   ON a.open_emp_id = e.emp_id
8   INNER JOIN
9    (SELECT branch_id
10     FROM branch
11     WHERE name = 'Woburn Branch') b
12   ON e.assigned_branch_id = b.branch_id;
```

The first subquery, which starts on line 3 and is given the alias `e`, finds all experienced tellers. The second subquery, which starts on line 9 and is given the alias `b`, finds the ID of the Woburn branch. First, the `account` table is joined to the experienced-teller subquery using the employee ID and then the table that results is joined to the Woburn branch subquery using the branch ID. The results are the same as those of the previous version of the query (try it and see for yourself), but the queries look very different from one another.

There isn't really anything shocking here, but it might take a minute to figure out what's going on. Notice, for example, the lack of a `where` clause in the main query; since all the filter conditions are against the `employee` and `branch` tables, the filter conditions are all inside the subqueries, so there is no need for any filter conditions in the main query. One way to visualize what is going on is to run the subqueries and look at the result sets. Here are the results of the first subquery against the `employee` table:

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE start_date < '2007-01-01'
    ->   AND (title = 'Teller' OR title = 'Head Teller');
+--------+--------------------+
| emp_id | assigned_branch_id |
+--------+--------------------+
|      8 |                  1 |
|      9 |                  1 |
|     10 |                  2 |
|     11 |                  2 |
|     13 |                  3 |
|     14 |                  3 |
|     16 |                  4 |
|     17 |                  4 |
|     18 |                  4 |
+--------+--------------------+
9 rows in set (0.03 sec)
```

Thus, this result set consists of a set of employee IDs and their corresponding branch IDs. When they are joined to the `account` table via the `emp_id` column, you now have an intermediate result set consisting of all rows from the `account` table with the additional column holding the branch ID of the employee that opened each account. Here are the results of the second subquery against the `branch` table:

```
mysql> SELECT branch_id
    -> FROM branch
    -> WHERE name = 'Woburn Branch';
```

```
+-----------+
| branch_id |
+-----------+
|         2 |
+-----------+
1 row in set (0.02 sec)
```

This query returns a single row containing a single column: the ID of the Woburn branch. This table is joined to the `assigned_branch_id` column of the intermediate result set, causing all accounts opened by non-Woburn-based employees to be filtered out of the final result set.

Using the Same Table Twice

If you are joining multiple tables, you might find that you need to join the same table more than once. In the sample database, for example, there are foreign keys to the `branch` table from both the `account` table (the branch at which the account was opened) and the `employee` table (the branch at which the employee works). If you want to include *both* branches in your result set, you can include the `branch` table twice in the `from` clause, joined once to the `employee` table and once to the `account` table. For this to work, you will need to give each instance of the `branch` table a different alias so that the server knows which one you are referring to in the various clauses, as in:

```
mysql> SELECT a.account_id, e.emp_id,
    ->   b_a.name open_branch, b_e.name emp_branch
    -> FROM account a INNER JOIN branch b_a
    ->   ON a.open_branch_id = b_a.branch_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    ->   INNER JOIN branch b_e
    ->   ON e.assigned_branch_id = b_e.branch_id
    -> WHERE a.product_cd = 'CHK';
+------------+--------+---------------+---------------+
| account_id | emp_id | open_branch   | emp_branch    |
+------------+--------+---------------+---------------+
|         10 |      1 | Headquarters  | Headquarters  |
|         14 |      1 | Headquarters  | Headquarters  |
|         21 |      1 | Headquarters  | Headquarters  |
|          1 |     10 | Woburn Branch | Woburn Branch |
|          4 |     10 | Woburn Branch | Woburn Branch |
|          7 |     13 | Quincy Branch | Quincy Branch |
|         13 |     16 | So. NH Branch | So. NH Branch |
|         18 |     16 | So. NH Branch | So. NH Branch |
|         24 |     16 | So. NH Branch | So. NH Branch |
|         28 |     16 | So. NH Branch | So. NH Branch |
+------------+--------+---------------+---------------+
10 rows in set (0.16 sec)
```

This query shows who opened each checking account, what branch it was opened at, and to which branch the employee who opened the account is currently assigned. The `branch` table is included twice, with aliases `b_a` and `b_e`. By assigning different aliases

to each instance of the branch table, the server is able to understand which instance you are referring to: the one joined to the account table, or the one joined to the employee table. Therefore, this is one example of a query that *requires* the use of table aliases.

Self-Joins

Not only can you include the same table more than once in the same query, but you can actually join a table to itself. This might seem like a strange thing to do at first, but there are valid reasons for doing so. The employee table, for example, includes a *self-referencing foreign key*, which means that it includes a column (superior_emp_id) that points to the primary key within the same table. This column points to the employee's manager (unless the employee is the head honcho, in which case the column is null). Using a *self-join*, you can write a query that lists every employee's name along with the name of his or her manager:

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e INNER JOIN employee e_mgr
    ->   ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
+----------+-----------+-----------+-----------+
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
| Helen    | Fleming   | Susan     | Hawthorne |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| Paula    | Roberts   | Susan     | Hawthorne |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| John     | Blake     | Susan     | Hawthorne |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| Theresa  | Markham   | Susan     | Hawthorne |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
+----------+-----------+-----------+-----------+
17 rows in set (0.00 sec)
```

This query includes two instances of the employee table: one to provide employee names (with the table alias e), and the other to provide manager names (with the table alias e_mgr). The on subclause uses these aliases to join the employee table to itself via the superior_emp_id foreign key. This is another example of a query for which table aliases are required; otherwise, the server wouldn't know whether you are referring to an employee or an employee's manager.

While there are 18 rows in the `employee` table, the query returned only 17 rows; the president of the bank, Michael Smith, has no superior (his `superior_emp_id` column is `null`), so the join failed for his row. To include Michael Smith in the result set, you would need to use an outer join, which we cover in Chapter 10.

Equi-Joins Versus Non-Equi-Joins

All of the multitable queries shown thus far have employed *equi-joins*, meaning that values from the two tables must match for the join to succeed. An equi-join always employs an equals sign, as in:

```
ON e.assigned_branch_id = b.branch_id
```

While the majority of your queries will employ equi-joins, you can also join your tables via ranges of values, which are referred to as *non-equi-joins*. Here's an example of a query that joins by a range of values:

```
SELECT e.emp_id, e.fname, e.lname, e.start_date
FROM employee e INNER JOIN product p
  ON e.start_date >= p.date_offered
  AND e.start_date <= p.date_retired
WHERE p.name = 'no-fee checking';
```

This query joins two tables that have no foreign key relationships. The intent is to find all employees who began working for the bank while the No-Fee Checking product was being offered. Thus, an employee's start date must be between the date the product was offered and the date the product was retired.

You may also find a need for a *self-non-equi-join*, meaning that a table is joined to itself using a non-equi-join. For example, let's say that the operations manager has decided to have a chess tournament for all bank tellers. You have been asked to create a list of all the pairings. You might try joining the `employee` table to itself for all tellers (`title = 'Teller'`) and return all rows where the `emp_id`s don't match (since a person can't play chess against himself):

```
mysql> SELECT e1.fname, e1.lname, 'VS' vs, e2.fname, e2.lname
    -> FROM employee e1 INNER JOIN employee e2
    ->   ON e1.emp_id != e2.emp_id
    -> WHERE e1.title = 'Teller' AND e2.title = 'Teller';
+---------+----------+----+---------+----------+
| fname   | lname    | vs | fname   | lname    |
+---------+----------+----+---------+----------+
| Sarah   | Parker   | VS | Chris   | Tucker   |
| Jane    | Grossman | VS | Chris   | Tucker   |
| Thomas  | Ziegler  | VS | Chris   | Tucker   |
| Samantha| Jameson  | VS | Chris   | Tucker   |
| Cindy   | Mason    | VS | Chris   | Tucker   |
| Frank   | Portman  | VS | Chris   | Tucker   |
| Beth    | Fowler   | VS | Chris   | Tucker   |
| Rick    | Tulman   | VS | Chris   | Tucker   |
| Chris   | Tucker   | VS | Sarah   | Parker   |
```

```
| Jane     | Grossman | VS | Sarah    | Parker   |
| Thomas   | Ziegler  | VS | Sarah    | Parker   |
| Samantha | Jameson  | VS | Sarah    | Parker   |
| Cindy    | Mason    | VS | Sarah    | Parker   |
| Frank    | Portman  | VS | Sarah    | Parker   |
| Beth     | Fowler   | VS | Sarah    | Parker   |
| Rick     | Tulman   | VS | Sarah    | Parker   |
...
| Chris    | Tucker   | VS | Rick     | Tulman   |
| Sarah    | Parker   | VS | Rick     | Tulman   |
| Jane     | Grossman | VS | Rick     | Tulman   |
| Thomas   | Ziegler  | VS | Rick     | Tulman   |
| Samantha | Jameson  | VS | Rick     | Tulman   |
| Cindy    | Mason    | VS | Rick     | Tulman   |
| Frank    | Portman  | VS | Rick     | Tulman   |
| Beth     | Fowler   | VS | Rick     | Tulman   |
+----------+----------+----+----------+----------+
72 rows in set (0.01 sec)
```

You're on the right track, but the problem here is that for each pairing (e.g., Sarah Parker versus Chris Tucker), there is also a reverse pairing (e.g., Chris Tucker versus Sarah Parker). One way to achieve the desired results is to use the join condition `e1.emp_id < e2.emp_id` so that each teller is paired only with those tellers having a higher employee ID (you can also use `e1.emp_id > e2.emp_id` if you wish):

```
mysql> SELECT e1.fname, e1.lname, 'VS' vs, e2.fname, e2.lname
    -> FROM employee e1 INNER JOIN employee e2
    ->   ON e1.emp_id < e2.emp_id
    -> WHERE e1.title = 'Teller' AND e2.title = 'Teller';
+----------+----------+----+----------+----------+
| fname    | lname    | vs | fname    | lname    |
+----------+----------+----+----------+----------+
| Chris    | Tucker   | VS | Sarah    | Parker   |
| Chris    | Tucker   | VS | Jane     | Grossman |
| Sarah    | Parker   | VS | Jane     | Grossman |
| Chris    | Tucker   | VS | Thomas   | Ziegler  |
| Sarah    | Parker   | VS | Thomas   | Ziegler  |
| Jane     | Grossman | VS | Thomas   | Ziegler  |
| Chris    | Tucker   | VS | Samantha | Jameson  |
| Sarah    | Parker   | VS | Samantha | Jameson  |
| Jane     | Grossman | VS | Samantha | Jameson  |
| Thomas   | Ziegler  | VS | Samantha | Jameson  |
| Chris    | Tucker   | VS | Cindy    | Mason    |
| Sarah    | Parker   | VS | Cindy    | Mason    |
| Jane     | Grossman | VS | Cindy    | Mason    |
| Thomas   | Ziegler  | VS | Cindy    | Mason    |
| Samantha | Jameson  | VS | Cindy    | Mason    |
| Chris    | Tucker   | VS | Frank    | Portman  |
| Sarah    | Parker   | VS | Frank    | Portman  |
| Jane     | Grossman | VS | Frank    | Portman  |
| Thomas   | Ziegler  | VS | Frank    | Portman  |
| Samantha | Jameson  | VS | Frank    | Portman  |
| Cindy    | Mason    | VS | Frank    | Portman  |
| Chris    | Tucker   | VS | Beth     | Fowler   |
| Sarah    | Parker   | VS | Beth     | Fowler   |
```

```
| Jane     | Grossman | VS | Beth     | Fowler   |
| Thomas   | Ziegler  | VS | Beth     | Fowler   |
| Samantha | Jameson  | VS | Beth     | Fowler   |
| Cindy    | Mason    | VS | Beth     | Fowler   |
| Frank    | Portman  | VS | Beth     | Fowler   |
| Chris    | Tucker   | VS | Rick     | Tulman   |
| Sarah    | Parker   | VS | Rick     | Tulman   |
| Jane     | Grossman | VS | Rick     | Tulman   |
| Thomas   | Ziegler  | VS | Rick     | Tulman   |
| Samantha | Jameson  | VS | Rick     | Tulman   |
| Cindy    | Mason    | VS | Rick     | Tulman   |
| Frank    | Portman  | VS | Rick     | Tulman   |
| Beth     | Fowler   | VS | Rick     | Tulman   |
+----------+----------+----+----------+----------+
36 rows in set (0.00 sec)
```

You now have a list of 36 pairings, which is the correct number when choosing pairs of 9 distinct things.

Join Conditions Versus Filter Conditions

You are now familiar with the concept that join conditions belong in the `on` subclause, while filter conditions belong in the `where` clause. However, SQL is flexible as to where you place your conditions, so you will need to take care when constructing your queries. For example, the following query joins two tables using a single join condition, and also includes a single filter condition in the `where` clause:

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         24 | CHK        | 04-1111111 |
|         25 | BUS        | 04-1111111 |
|         27 | BUS        | 04-2222222 |
|         28 | CHK        | 04-3333333 |
|         29 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.01 sec)
```

That was pretty straightforward, but what happens if you mistakenly put the filter condition in the `on` subclause instead of in the `where` clause?

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->     AND c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         24 | CHK        | 04-1111111 |
```

```
|         25 | BUS        | 04-1111111 |
|         27 | BUS        | 04-2222222 |
|         28 | CHK        | 04-3333333 |
|         29 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.01 sec)
```

As you can see, the second version, which has *both* conditions in the `on` subclause and has no `where` clause, generates the same results. What if both conditions are placed in the `where` clause but the `from` clause still uses the ANSI join syntax?

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    -> WHERE a.cust_id = c.cust_id
    ->   AND c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         24 | CHK        | 04-1111111 |
|         25 | BUS        | 04-1111111 |
|         27 | BUS        | 04-2222222 |
|         28 | CHK        | 04-3333333 |
|         29 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.01 sec)
```

Once again, the MySQL server has generated the same result set. It will be up to you to put your conditions in the proper place so that your queries are easy to understand and maintain.

Test Your Knowledge

The following exercises are designed to test your understanding of inner joins. Please see Appendix C for the solutions to these exercises.

Exercise 5-1

Fill in the blanks (denoted by `<#>`) for the following query to obtain the results that follow:

```
mysql> SELECT e.emp_id, e.fname, e.lname, b.name
    -> FROM employee e INNER JOIN <1> b
    ->   ON e.assigned_branch_id = b.<2>;
+--------+----------+-----------+---------------+
| emp_id | fname    | lname     | name          |
+--------+----------+-----------+---------------+
|      1 | Michael  | Smith     | Headquarters  |
|      2 | Susan    | Barker    | Headquarters  |
|      3 | Robert   | Tyler     | Headquarters  |
|      4 | Susan    | Hawthorne | Headquarters  |
```

```
|      5 | John     | Gooding   | Headquarters  |
|      6 | Helen    | Fleming   | Headquarters  |
|      7 | Chris    | Tucker    | Headquarters  |
|      8 | Sarah    | Parker    | Headquarters  |
|      9 | Jane     | Grossman  | Headquarters  |
|     10 | Paula    | Roberts   | Woburn Branch |
|     11 | Thomas   | Ziegler   | Woburn Branch |
|     12 | Samantha | Jameson   | Woburn Branch |
|     13 | John     | Blake     | Quincy Branch |
|     14 | Cindy    | Mason     | Quincy Branch |
|     15 | Frank    | Portman   | Quincy Branch |
|     16 | Theresa  | Markham   | So. NH Branch |
|     17 | Beth     | Fowler    | So. NH Branch |
|     18 | Rick     | Tulman    | So. NH Branch |
+--------+----------+-----------+---------------+
18 rows in set (0.03 sec)
```

Exercise 5-2

Write a query that returns the account ID for each nonbusiness customer (`customer.cust_type_cd = 'I'`) with the customer's federal ID (`customer.fed_id`) and the name of the product on which the account is based (`product.name`).

Exercise 5-3

Construct a query that finds all employees whose supervisor is assigned to a different department. Retrieve the employees' ID, first name, and last name.

CHAPTER 6

Working with Sets

Although you can interact with the data in a database one row at a time, relational databases are really all about sets. You have seen how you can create tables via queries or subqueries, make them persistent via `insert` statements, and bring them together via joins; this chapter explores how you can combine multiple tables using various set operators.

Set Theory Primer

In many parts of the world, basic set theory is included in elementary-level math curriculums. Perhaps you recall looking at something like what is shown in Figure 6-1.

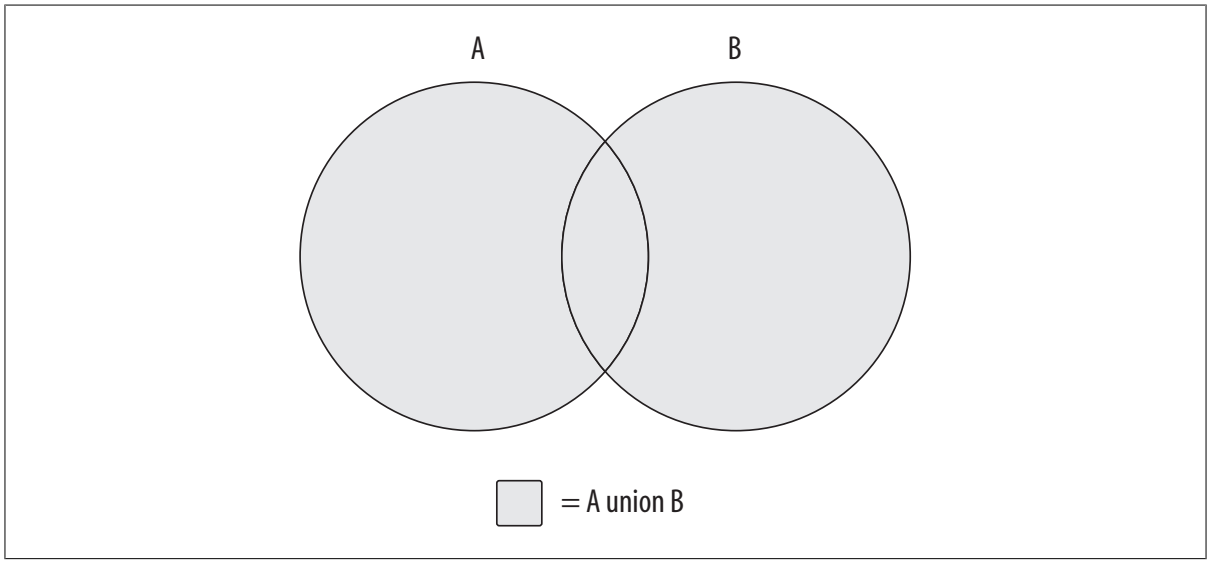


Figure 6-1. The union operation

The shaded area in Figure 6-1 represents the *union* of sets A and B, which is the combination of the two sets (with any overlapping regions included only once). Is this starting to look familiar? If so, then you'll finally get a chance to put that knowledge to use; if not, don't worry, because it's easy to visualize using a couple of diagrams.

Using circles to represent two data sets (A and B), imagine a subset of data that is common to both sets; this common data is represented by the overlapping area shown in Figure 6-1. Since set theory is rather uninteresting without an overlap between data sets, I use the same diagram to illustrate each set operation. There is another set operation that is concerned *only* with the overlap between two data sets; this operation is known as the *intersection* and is demonstrated in Figure 6-2.

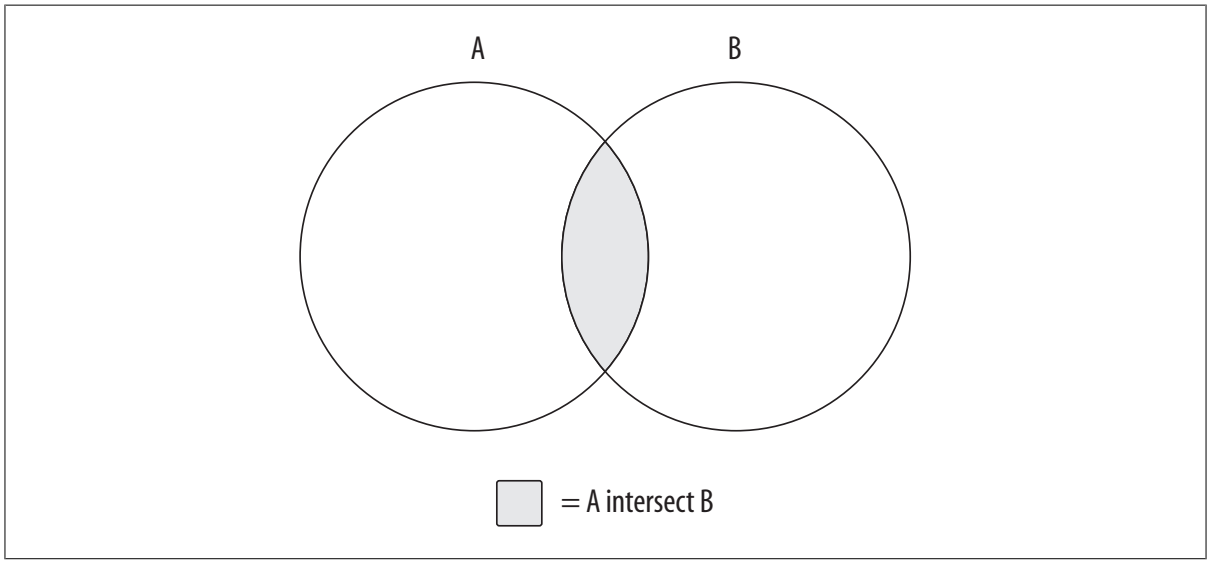


Figure 6-2. The intersection operation

The data set generated by the intersection of sets A and B is just the area of overlap between the two sets. If the two sets have no overlap, then the intersection operation yields the empty set.

The third and final set operation, which is demonstrated in Figure 6-3, is known as the *except* operation.

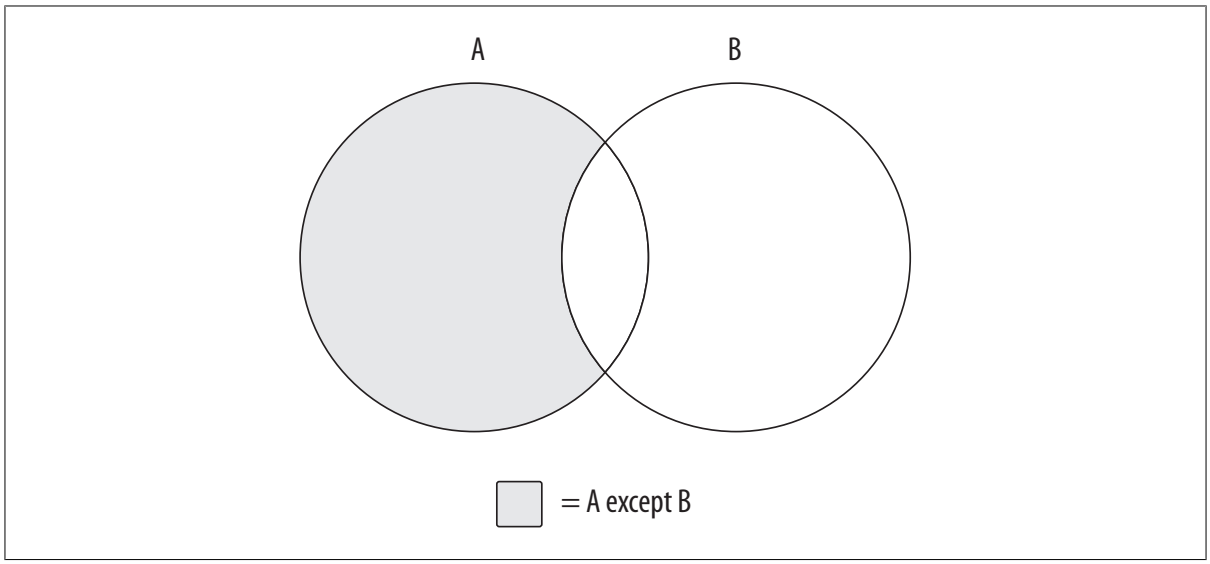


Figure 6-3. The except operation

Figure 6-3 shows the results of `A except B`, which is the whole of set A minus any overlap with set B. If the two sets have no overlap, then the operation `A except B` yields the whole of set A.

Using these three operations, or by combining different operations together, you can generate whatever results you need. For example, imagine that you want to build a set demonstrated by Figure 6-4.

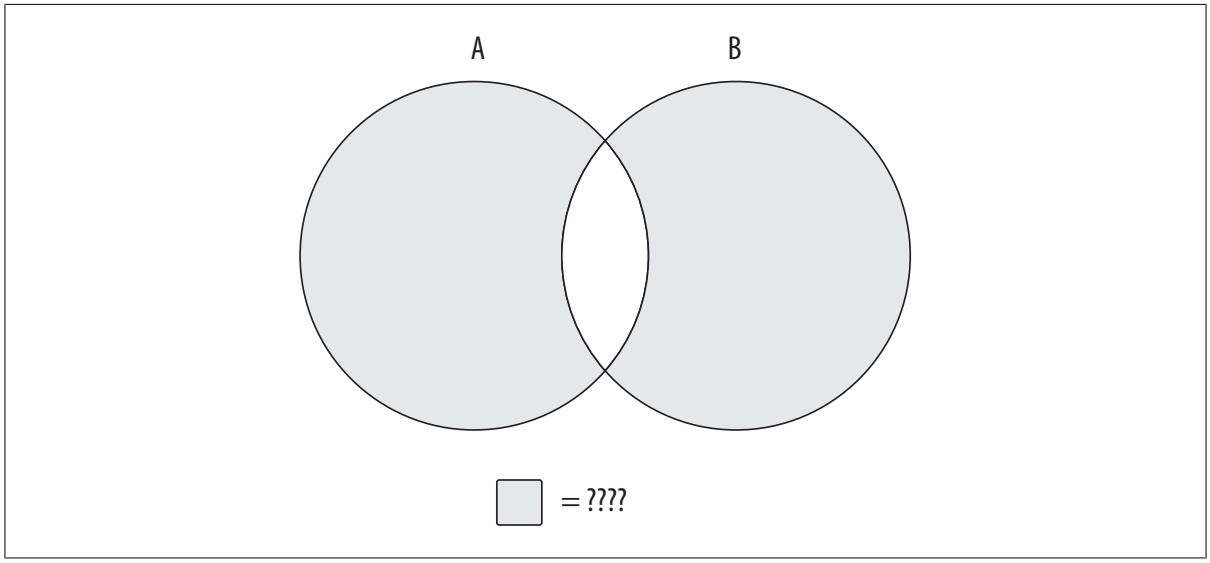


Figure 6-4. Mystery data set

The data set you are looking for includes all of sets A and B *without* the overlapping region. You can't achieve this outcome with just one of the three operations shown earlier; instead, you will need to first build a data set that encompasses all of sets A and B, and then utilize a second operation to remove the overlapping region. If the combined set is described as `A union B`, and the overlapping region is described as `A intersect B`, then the operation needed to generate the data set represented by Figure 6-4 would look as follows:

```
(A union B) except (A intersect B)
```

Of course, there are often multiple ways to achieve the same results; you could reach a similar outcome using the following operation:

```
(A except B) union (B except A)
```

While these concepts are fairly easy to understand using diagrams, the next sections show you how these concepts are applied to a relational database using the SQL set operators.

Set Theory in Practice

The circles used in the previous section's diagrams to represent data sets don't convey anything about what the data sets comprise. When dealing with actual data, however,

there is a need to describe the composition of the data sets involved if they are to be combined. Imagine, for example, what would happen if you tried to generate the union of the `product` table and the `customer` table, whose table definitions are as follows:

```
mysql> DESC product;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| product_cd      | varchar(10) | NO   | PRI | NULL    |       |
| name            | varchar(50) | NO   |     | NULL    |       |
| product_type_cd | varchar(10) | NO   | MUL | NULL    |       |
| date_offered    | date        | YES  |     | NULL    |       |
| date_retired    | date        | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
5 rows in set (0.23 sec)
 mysql> DESC customer;
+--------------+------------------+------+-----+---------+----------------+
| Field        | Type             | Null | Key | Default | Extra          |
+--------------+------------------+------+-----+---------+----------------+
| cust_id      | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| fed_id       | varchar(12)      | NO   |     | NULL    |                |
| cust_type_cd | enum('I','B')    | NO   |     | NULL    |                |
| address      | varchar(30)      | YES  |     | NULL    |                |
| city         | varchar(20)      | YES  |     | NULL    |                |
| state        | varchar(20)      | YES  |     | NULL    |                |
| postal_code  | varchar(10)      | YES  |     | NULL    |                |
+--------------+------------------+------+-----+---------+----------------+
7 rows in set (0.04 sec)
```

When combined, the first column in the table that results would be the combination of the `product.product_cd` and `customer.cust_id` columns, the second column would be the combination of the `product.name` and `customer.fed_id` columns, and so forth. While some of the column pairs are easy to combine (e.g., two numeric columns), it is unclear how other column pairs should be combined, such as a numeric column with a string column or a string column with a date column. Additionally, the sixth and seventh columns of the combined tables would include data from only the `customer` table's sixth and seventh columns, since the `product` table has only five columns. Clearly, there needs to be some commonality between two tables that you wish to combine.

Therefore, when performing set operations on two data sets, the following guidelines must apply:

- Both data sets must have the same number of columns.
- The data types of each column across the two data sets must be the same (or the server must be able to convert one to the other).

With these rules in place, it is easier to envision what "overlapping data" means in practice; each column pair from the two sets being combined must contain the same string, number, or date for rows in the two tables to be considered the same.

You perform a set operation by placing a *set operator* between two `select` statements, as demonstrated by the following:

```
mysql> SELECT 1 num, 'abc' str
    -> UNION
    -> SELECT 9 num, 'xyz' str;
+-----+-----+
| num | str |
+-----+-----+
|   1 | abc |
|   9 | xyz |
+-----+-----+
2 rows in set (0.02 sec)
```

Each of the individual queries yields a data set consisting of a single row having a numeric column and a string column. The set operator, which in this case is `union`, tells the database server to combine all rows from the two sets. Thus, the final set includes two rows of two columns. This query is known as a *compound query* because it comprises multiple, otherwise-independent queries. As you will see later, compound queries may include *more* than two queries if multiple set operations are needed to attain the final results.

Set Operators

The SQL language includes three set operators that allow you to perform each of the various set operations described earlier in the chapter. Additionally, each set operator has two flavors, one that includes duplicates and another that removes duplicates (but not necessarily *all* of the duplicates). The following subsections define each operator and demonstrate how they are used.

The union Operator

The `union` and `union all` operators allow you to combine multiple data sets. The difference between the two is that `union` sorts the combined set and removes duplicates, whereas `union all` does not. With `union all`, the number of rows in the final data set will always equal the sum of the number of rows in the sets being combined. This operation is the simplest set operation to perform (from the server's point of view), since there is no need for the server to check for overlapping data. The following example demonstrates how you can use the `union all` operator to generate a full set of customer data from the two customer subtype tables:

```
mysql> SELECT 'IND' type_cd, cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business;
+---------+---------+-----------------------+
| type_cd | cust_id | name                  |
+---------+---------+-----------------------+
```

```
| IND      |       1 | Hadley                 |
| IND      |       2 | Tingley                |
| IND      |       3 | Tucker                 |
| IND      |       4 | Hayward                |
| IND      |       5 | Frasier                |
| IND      |       6 | Spencer                |
| IND      |       7 | Young                  |
| IND      |       8 | Blake                  |
| IND      |       9 | Farley                 |
| BUS      |      10 | Chilton Engineering    |
| BUS      |      11 | Northeast Cooling Inc. |
| BUS      |      12 | Superior Auto Body     |
| BUS      |      13 | AAA Insurance Inc.     |
+----------+---------+------------------------+
13 rows in set (0.04 sec)
```

The query returns all 13 customers, with nine rows coming from the `individual` table and the other four coming from the `business` table. While the `business` table includes a single column to hold the company name, the `individual` table includes two name columns, one each for the person's first and last names. In this case, I chose to include only the last name from the `individual` table.

Just to drive home the point that the `union all` operator doesn't remove duplicates, here's the same query as the previous example but with an additional query against the `business` table:

```
mysql> SELECT 'IND' type_cd, cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business
    -> UNION ALL
    -> SELECT 'BUS' type_cd, cust_id, name
    -> FROM business;
+----------+---------+------------------------+
| type_cd | cust_id | name                   |
+----------+---------+------------------------+
| IND      |       1 | Hadley                 |
| IND      |       2 | Tingley                |
| IND      |       3 | Tucker                 |
| IND      |       4 | Hayward                |
| IND      |       5 | Frasier                |
| IND      |       6 | Spencer                |
| IND      |       7 | Young                  |
| IND      |       8 | Blake                  |
| IND      |       9 | Farley                 |
| BUS      |      10 | Chilton Engineering    |
| BUS      |      11 | Northeast Cooling Inc. |
| BUS      |      12 | Superior Auto Body     |
| BUS      |      13 | AAA Insurance Inc.     |
| BUS      |      10 | Chilton Engineering    |
| BUS      |      11 | Northeast Cooling Inc. |
| BUS      |      12 | Superior Auto Body     |
| BUS      |      13 | AAA Insurance Inc.     |
```

```
+---------+---------+-----------------------+
17 rows in set (0.01 sec)
```

This compound query includes three `select` statements, two of which are identical. As you can see by the results, the four rows from the `business` table are included twice (customer IDs 10, 11, 12, and 13).

While you are unlikely to repeat the same query twice in a compound query, here is another compound query that returns duplicate data:

```
mysql> SELECT emp_id
    -> FROM employee
    -> WHERE assigned_branch_id = 2
    ->   AND (title = 'Teller' OR title = 'Head Teller')
    -> UNION ALL
    -> SELECT DISTINCT open_emp_id
    -> FROM account
    -> WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
|     12 |
|     10 |
+--------+
4 rows in set (0.01 sec)
```

The first query in the compound statement retrieves all tellers assigned to the Woburn branch, whereas the second query returns the distinct set of tellers who opened accounts at the Woburn branch. Of the four rows in the result set, one of them is a duplicate (employee ID 10). If you would like your combined table to *exclude* duplicate rows, you need to use the `union` operator instead of `union all`:

```
mysql> SELECT emp_id
    -> FROM employee
    -> WHERE assigned_branch_id = 2
    ->   AND (title = 'Teller' OR title = 'Head Teller')
    -> UNION
    -> SELECT DISTINCT open_emp_id
    -> FROM account
    -> WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
|     12 |
+--------+
3 rows in set (0.01 sec)
```

For this version of the query, only the three distinct rows are included in the result set, rather than the four rows (three distinct, one duplicate) returned when using `union all`.

The intersect Operator

The ANSI SQL specification includes the `intersect` operator for performing intersections. Unfortunately, version 6.0 of MySQL does not implement the `intersect` operator. If you are using Oracle or SQL Server 2008, you will be able to use `intersect`; since I am using MySQL for all examples in this book, however, the result sets for the example queries in this section are fabricated and cannot be executed with any versions up to and including version 6.0. I also refrain from showing the MySQL prompt (`mysql>`), since the statements are not being executed by the MySQL server.

If the two queries in a compound query return nonoverlapping data sets, then the intersection will be an empty set. Consider the following query:

```
SELECT emp_id, fname, lname
FROM employee
INTERSECT
SELECT cust_id, fname, lname
FROM individual;
Empty set (0.04 sec)
```

The first query returns the ID and name of each employee, while the second query returns the ID and name of each customer. These sets are completely nonoverlapping, so the intersection of the two sets yields the empty set.

The next step is to identify two queries that *do* have overlapping data and then apply the `intersect` operator. For this purpose, I use the same query used to demonstrate the difference between `union` and `union all`, except this time using `intersect`:

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
INTERSECT
SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     10 |
+--------+
1 row in set (0.01 sec)
```

The intersection of these two queries yields employee ID 10, which is the only value found in both queries' result sets.

Along with the `intersect` operator, which removes any duplicate rows found in the overlapping region, the ANSI SQL specification calls for an `intersect all` operator, which does not remove duplicates. The only database server that currently implements the `intersect all` operator is IBM's DB2 Universal Server.

The except Operator

The ANSI SQL specification includes the `except` operator for performing the except operation. Once again, unfortunately, version 6.0 of MySQL does not implement the `except` operator, so the same rules apply for this section as for the previous section.

If you are using Oracle Database, you will need to use the non-ANSI-compliant `minus` operator instead.

The `except` operator returns the first table minus any overlap with the second table. Here's the example from the previous section, but using `except` instead of `intersect`:

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
EXCEPT
SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
2 rows in set (0.01 sec)
```

In this version of the query, the result set consists of the three rows from the first query minus employee ID 10, which is found in the result sets from both queries. There is also an `except all` operator specified in the ANSI SQL specification, but once again, only IBM's DB2 Universal Server has implemented the `except all` operator.

The `except all` operator is a bit tricky, so here's an example to demonstrate how duplicate data is handled. Let's say you have two data sets that look as follows:

Set A

```
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
|     12 |
|     10 |
|     10 |
+--------+
```

Set B

```
+--------+
| emp_id |
+--------+
|     10 |
|     10 |
+--------+
```

The operation `A except B` yields the following:

```
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
```

If you change the operation to `A except all B`, you will see the following:

```
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
|     12 |
+--------+
```

Therefore, the difference between the two operations is that `except` removes all occurrences of duplicate data from set A, whereas `except all` only removes one occurrence of duplicate data from set A for every occurrence in set B.

Set Operation Rules

The following sections outline some rules that you must follow when working with compound queries.

Sorting Compound Query Results

If you want the results of your compound query to be sorted, you can add an `order by` clause after the last query. When specifying column names in the `order by` clause, you will need to choose from the column names in the first query of the compound query. Frequently, the column names are the same for both queries in a compound query, but this does not need to be the case, as demonstrated by the following:

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE title = 'Teller'
    -> UNION
    -> SELECT open_emp_id, open_branch_id
    -> FROM account
    -> WHERE product_cd = 'SAV'
    -> ORDER BY emp_id;
```

```
+--------+--------------------+
| emp_id | assigned_branch_id |
+--------+--------------------+
|      1 |                  1 |
|      7 |                  1 |
|      8 |                  1 |
|      9 |                  1 |
|     10 |                  2 |
|     11 |                  2 |
|     12 |                  2 |
|     14 |                  3 |
|     15 |                  3 |
|     16 |                  4 |
|     17 |                  4 |
|     18 |                  4 |
+--------+--------------------+
12 rows in set (0.04 sec)
```

The column names specified in the two queries are different in this example. If you specify a column name from the second query in your `order by` clause, you will see the following error:

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE title = 'Teller'
    -> UNION
    -> SELECT open_emp_id, open_branch_id
    -> FROM account
    -> WHERE product_cd = 'SAV'
    -> ORDER BY open_emp_id;
ERROR 1054 (42S22): Unknown column 'open_emp_id' in 'order clause'
```

I recommend giving the columns in both queries identical column aliases in order to avoid this issue.

Set Operation Precedence

If your compound query contains more than two queries using different set operators, you need to think about the order in which to place the queries in your compound statement to achieve the desired results. Consider the following three-query compound statement:

```
mysql> SELECT cust_id
    -> FROM account
    -> WHERE product_cd IN ('SAV', 'MM')
    -> UNION ALL
    -> SELECT a.cust_id
    -> FROM account a INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    -> WHERE b.name = 'Woburn Branch'
    -> UNION
    -> SELECT cust_id
    -> FROM account
    -> WHERE avail_balance BETWEEN 500 AND 2500;
```

```
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       8 |
|       9 |
|       7 |
|      11 |
|       5 |
+---------+
9 rows in set (0.00 sec)
```

This compound query includes three queries that return sets of nonunique customer IDs; the first and second queries are separated with the `union all` operator, while the second and third queries are separated with the `union` operator. While it might not seem to make much difference where the `union` and `union all` operators are placed, it does, in fact, make a difference. Here's the same compound query with the set operators reversed:

```
mysql> SELECT cust_id
    -> FROM account
    -> WHERE product_cd IN ('SAV', 'MM')
    -> UNION
    -> SELECT a.cust_id
    -> FROM account a INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    -> WHERE b.name = 'Woburn Branch'
    -> UNION ALL
    -> SELECT cust_id
    -> FROM account
    -> WHERE avail_balance BETWEEN 500 AND 2500;
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       8 |
|       9 |
|       7 |
|      11 |
|       1 |
|       1 |
|       2 |
|       3 |
|       3 |
|       4 |
|       4 |
|       5 |
|       9 |
```

```
+---------+
17 rows in set (0.00 sec)
```

Looking at the results, it's obvious that it *does* make a difference how the compound query is arranged when using different set operators. In general, compound queries containing three or more queries are evaluated in order from top to bottom, but with the following caveats:

- The ANSI SQL specification calls for the `intersect` operator to have precedence over the other set operators.
- You may dictate the order in which queries are combined by enclosing multiple queries in parentheses.

However, since MySQL does not yet implement `intersect` or allow parentheses in compound queries, you will need to carefully arrange the queries in your compound query so that you achieve the desired results. If you are using a different database server, you can wrap adjoining queries in parentheses to override the default top-to-bottom processing of compound queries, as in:

```
(SELECT cust_id
 FROM account
 WHERE product_cd IN ('SAV', 'MM')
 UNION ALL
 SELECT a.cust_id
 FROM account a INNER JOIN branch b
   ON a.open_branch_id = b.branch_id
 WHERE b.name = 'Woburn Branch')
INTERSECT
(SELECT cust_id
 FROM account
 WHERE avail_balance BETWEEN 500 AND 2500
 EXCEPT
 SELECT cust_id
 FROM account
 WHERE product_cd = 'CD'
  AND avail_balance < 1000);
```

For this compound query, the first and second queries would be combined using the `union all` operator, then the third and fourth queries would be combined using the `except` operator, and finally, the results from these two operations would be combined using the `intersect` operator to generate the final result set.

Test Your Knowledge

The following exercises are designed to test your understanding of set operations. See Appendix C for answers to these exercises.

Exercise 6-1

If set A = {L M N O P} and set B = {P Q R S T}, what sets are generated by the following operations?

- `A union B`
- `A union all B`
- `A intersect B`
- `A except B`

Exercise 6-2

Write a compound query that finds the first and last names of all individual customers along with the first and last names of all employees.

Exercise 6-3

Sort the results from Exercise 6-2 by the `lname` column.

CHAPTER 7

Data Generation, Conversion, and Manipulation

As I mentioned in the Preface, this book strives to teach generic SQL techniques that can be applied across multiple database servers. This chapter, however, deals with the generation, conversion, and manipulation of string, numeric, and temporal data, and the SQL language does not include commands covering this functionality. Rather, built-in functions are used to facilitate data generation, conversion, and manipulation, and while the SQL standard does specify some functions, the database vendors often do not comply with the function specifications.

Therefore, my approach for this chapter is to show you some of the common ways in which data is manipulated within SQL statements, and then demonstrate some of the built-in functions implemented by Microsoft SQL Server, Oracle Database, and MySQL. Along with reading this chapter, I strongly recommend you purchase a reference guide covering all the functions implemented by your server. If you work with more than one database server, there are several reference guides that cover multiple servers, such as Kevin Kline et al.'s *SQL in a Nutshell* (*http://oreilly.com/catalog/9780596518844/*) and Jonathan Gennick's *SQL Pocket Guide (http://oreilly.com/catalog/9780596526887/)*, both from O'Reilly.

Working with String Data

When working with string data, you will be using one of the following character data types:

`CHAR`
: Holds fixed-length, blank-padded strings. MySQL allows `CHAR` values up to 255 characters in length, Oracle Database permits up to 2,000 characters, and SQL Server allows up to 8,000 characters.

`varchar`
: Holds variable-length strings. MySQL permits up to 65,535 characters in a `varchar` column, Oracle Database (via the `varchar2` type) allows up to 4,000 characters, and SQL Server allows up to 8,000 characters.

`text` *(MySQL and SQL Server) or* `CLOB` *(Character Large Object; Oracle Database)*
: Holds very large variable-length strings (generally referred to as documents in this context). MySQL has multiple text types (`tinytext`, `text`, `mediumtext`, and `long text`) for documents up to 4 GB in size. SQL Server has a single `text` type for documents up to 2 GB in size, and Oracle Database includes the `CLOB` data type, which can hold documents up to a whopping 128 TB. SQL Server 2005 also includes the `varchar(max)` data type and recommends its use instead of the `text` type, which will be removed from the server in some future release.

To demonstrate how you can use these various types, I use the following table for some of the examples in this section:

```
CREATE TABLE string_tbl
 (char_fld CHAR(30),
  vchar_fld VARCHAR(30),
  text_fld TEXT
 );
```

The next two subsections show how you can generate and manipulate string data.

String Generation

The simplest way to populate a character column is to enclose a string in quotes, as in:

```
mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
    -> VALUES ('This is char data',
    ->   'This is varchar data',
    ->   'This is text data');
Query OK, 1 row affected (0.00 sec)
```

When inserting string data into a table, remember that if the length of the string exceeds the maximum size for the character column (either the designated maximum or the maximum allowed for the data type), the server will throw an exception. Although this is the default behavior for all three servers, you can configure MySQL and SQL Server to silently truncate the string instead of throwing an exception. To demonstrate how MySQL handles this situation, the following `update` statement attempts to modify the `vchar_fld` column, whose maximum length is defined as 30, with a string that is 46 characters in length:

```
mysql> UPDATE string_tbl
    -> SET vchar_fld = 'This is a piece of extremely long varchar data';
ERROR 1406 (22001): Data too long for column 'vchar_fld' at row 1
```

With MySQL 6.0, the default behavior is now "strict" mode, which means that exceptions are thrown when problems arise, whereas in older versions of the server the string would have been truncated and a warning issued. If you would rather have the engine

truncate the string and issue a warning instead of raising an exception, you can opt to be in ANSI mode. The following example shows how to check which mode you are in, and then how to change the mode using the `SET` command:

```
mysql> SELECT @@session.sql_mode;
+-------------------------------------------------------------+
| @@session.sql_mode                                          |
+-------------------------------------------------------------+
| STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SET sql_mode='ansi';
Query OK, 0 rows affected (0.08 sec)

mysql> SELECT @@session.sql_mode;
+-------------------------------------------------------------+
| @@session.sql_mode                                          |
+-------------------------------------------------------------+
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,ANSI |
+-------------------------------------------------------------+
1 row in set (0.00 sec)
```

If you rerun the previous `UPDATE` statement, you will find that the column has been modified, but the following warning is generated:

```
mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------+
| Level   | Code | Message                                       |
+---------+------+-----------------------------------------------+
| Warning | 1265 | Data truncated for column 'vchar_fld' at row 1 |
+---------+------+-----------------------------------------------+
1 row in set (0.00 sec)
```

If you retrieve the `vchar_fld` column, you will see that the string has indeed been truncated:

```
mysql> SELECT vchar_fld
    -> FROM string_tbl;
+--------------------------------+
| vchar_fld                      |
+--------------------------------+
| This is a piece of extremely l |
+--------------------------------+
1 row in set (0.05 sec)
```

As you can see, only the first 30 characters of the 46-character string made it into the `vchar_fld` column. The best way to avoid string truncation (or exceptions, in the case of Oracle Database or MySQL in strict mode) when working with `varchar` columns is to set the upper limit of a column to a high enough value to handle the longest strings that might be stored in the column (keeping in mind that the server allocates only enough space to store the string, so it is not wasteful to set a high upper limit for `varchar` columns).

Including single quotes

Since strings are demarcated by single quotes, you will need to be alert for strings that include single quotes or apostrophes. For example, you won't be able to insert the following string because the server will think that the apostrophe in the word *doesn't* marks the end of the string:

```
UPDATE string_tbl
SET text_fld = 'This string doesn't work';
```

To make the server ignore the apostrophe in the word *doesn't*, you will need to add an *escape* to the string so that the server treats the apostrophe like any other character in the string. All three servers allow you to escape a single quote by adding another single quote directly before, as in:

```
mysql> UPDATE string_tbl
    -> SET text_fld = 'This string didn''t work, but it does now';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Oracle Database and MySQL users may also choose to escape a single quote by adding a backslash character immediately before, as in:

```
UPDATE string_tbl SET text_fld =
  'This string didn\'t work, but it does now'
```

If you retrieve a string for use in a screen or report field, you don't need to do anything special to handle embedded quotes:

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+----------------------------------------+
| text_fld                               |
+----------------------------------------+
| This string didn't work, but it does now |
+----------------------------------------+
1 row in set (0.00 sec)
```

However, if you are retrieving the string to add to a file that another program will read, you may want to include the escape as part of the retrieved string. If you are using MySQL, you can use the built-in function `quote()`, which places quotes around the entire string *and* adds escapes to any single quotes/apostrophes within the string. Here's what our string looks like when retrieved via the `quote()` function:

```
mysql> SELECT quote(text_fld)
    -> FROM string_tbl;
+-----------------------------------------+
| QUOTE(text_fld)                         |
+-----------------------------------------+
| 'This string didn\'t work, but it does now' |
+-----------------------------------------+
1 row in set (0.04 sec)
```

When retrieving data for data export, you may want to use the `quote()` function for all non-system-generated character columns, such as a `customer_notes` column.

Including special characters

If your application is multinational in scope, you might find yourself working with strings that include characters that do not appear on your keyboard. When working with the French and German languages, for example, you might need to include accented characters such as é and ö. The SQL Server and MySQL servers include the built-in function `char()` so that you can build strings from any of the 255 characters in the ASCII character set (Oracle Database users can use the `chr()` function). To demonstrate, the next example retrieves a typed string and its equivalent built via individual characters:

```
mysql> SELECT 'abcdefg', CHAR(97,98,99,100,101,102,103);
+---------+-------------------------------+
| abcdefg | CHAR(97,98,99,100,101,102,103) |
+---------+-------------------------------+
| abcdefg | abcdefg                       |
+---------+-------------------------------+
1 row in set (0.01 sec)
```

Thus, the 97th character in the ASCII character set is the letter *a*. While the characters shown in the preceding example are not special, the following examples show the location of the accented characters along with other special characters, such as currency symbols:

```
mysql> SELECT CHAR(128,129,130,131,132,133,134,135,136,137);
+-----------------------------------------------+
| CHAR(128,129,130,131,132,133,134,135,136,137) |
+-----------------------------------------------+
| Çüéâäàåçêë                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(138,139,140,141,142,143,144,145,146,147);
+-----------------------------------------------+
| CHAR(138,139,140,141,142,143,144,145,146,147) |
+-----------------------------------------------+
| èïîìÄÅÉæÆô                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(148,149,150,151,152,153,154,155,156,157);
+-----------------------------------------------+
| CHAR(148,149,150,151,152,153,154,155,156,157) |
+-----------------------------------------------+
| öòÛùÿ...Ü¢£¥                                  |
+-----------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CHAR(158,159,160,161,162,163,164,165);
+---------------------------------------+
```

```
| CHAR(158,159,160,161,162,163,164,165) |
+---------------------------------------+
| fáíóúñÑ                               |
+---------------------------------------+
1 row in set (0.01 sec)
```

I am using the latin1 character set for the examples in this section. If your session is configured for a different character set, you will see a different set of characters than what is shown here. The same concepts apply, but you will need to familiarize yourself with the layout of your character set to locate specific characters.

Building strings character by character can be quite tedious, especially if only a few of the characters in the string are accented. Fortunately, you can use the `concat()` function to concatenate individual strings, some of which you can type while others you can generate via the `char()` function. For example, the following shows how to build the phrase *danke schön* using the `concat()` and `char()` functions:

```
mysql> SELECT CONCAT('danke sch', CHAR(148), 'n');
+-------------------------------------+
| CONCAT('danke sch', CHAR(148), 'n') |
+-------------------------------------+
| danke schön                         |
+-------------------------------------+
1 row in set (0.00 sec)
```

Oracle Database users can use the concatenation operator (`||`) instead of the `concat()` function, as in:

```
SELECT 'danke sch' || CHR(148) || 'n'
FROM dual;
```

SQL Server does not include a `concat()` function, so you will need to use the concatenation operator (`+`), as in:

```
SELECT 'danke sch' + CHAR(148) + 'n'
```

If you have a character and need to find its ASCII equivalent, you can use the `ascii()` function, which takes the leftmost character in the string and returns a number:

```
mysql> SELECT ASCII('ö');
+------------+
| ASCII('ö') |
+------------+
|        148 |
+------------+
1 row in set (0.00 sec)
```

Using the `char()`, `ascii()`, and `concat()` functions (or concatenation operators), you should be able to work with any Roman language even if you are using a keyboard that does not include accented or special characters.

String Manipulation

Each database server includes many built-in functions for manipulating strings. This section explores two types of string functions: those that return numbers and those that return strings. Before I begin, however, I reset the data in the `string_tbl` table to the following:

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
    -> VALUES ('This string is 28 characters',
    ->   'This string is 28 characters',
    ->   'This string is 28 characters');
Query OK, 1 row affected (0.00 sec)
```

String functions that return numbers

Of the string functions that return numbers, one of the most commonly used is the `length()` function, which returns the number of characters in the string (SQL Server users will need to use the `len()` function). The following query applies the `length()` function to each column in the `string_tbl` table:

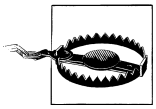
```
mysql> SELECT LENGTH(char_fld) char_length,
    ->   LENGTH(vchar_fld) varchar_length,
    ->   LENGTH(text_fld) text_length
    -> FROM string_tbl;
+-------------+---------------+-------------+
| char_length | varchar_length | text_length |
+-------------+---------------+-------------+
|          28 |             28 |          28 |
+-------------+---------------+-------------+
1 row in set (0.00 sec)
```

While the lengths of the `varchar` and `text` columns are as expected, you might have expected the length of the `char` column to be 30, since I told you that strings stored in `char` columns are right-padded with spaces. The MySQL server removes trailing spaces from `char` data when it is retrieved, however, so you will see the same results from all string functions regardless of the type of column in which the strings are stored.

Along with finding the length of a string, you might want to find the location of a substring within a string. For example, if you want to find the position at which the string `'characters'` appears in the `vchar_fld` column, you could use the `position()` function, as demonstrated by the following:

```
mysql> SELECT POSITION('characters' IN vchar_fld)
    -> FROM string_tbl;
+-----------------------------------+
| POSITION('characters' IN vchar_fld) |
+-----------------------------------+
|                                19 |
+-----------------------------------+
1 row in set (0.12 sec)
```

If the substring cannot be found, the `position()` function returns `0`.



For those of you who program in a language such as C or C++, where the first element of an array is at position 0, remember when working with databases that the first character in a string is at position 1. A return value of `0` from `position()` indicates that the substring could not be found, not that the substring was found at the first position in the string.

If you want to start your search at something other than the first character of your target string, you will need to use the `locate()` function, which is similar to the `position()` function except that it allows an optional third parameter, which is used to define the search's start position. The `locate()` function is also proprietary, whereas the `position()` function is part of the SQL:2003 standard. Here's an example asking for the position of the string `'is'` starting at the fifth character in the `vchar_fld` column:

```
mysql> SELECT LOCATE('is', vchar_fld, 5)
    -> FROM string_tbl;
+----------------------------+
| LOCATE('is', vchar_fld, 5) |
+----------------------------+
|                         13 |
+----------------------------+
1 row in set (0.02 sec)
```

Oracle Database does not include the `position()` or `locate()` function, but it does include the `instr()` function, which mimics the `position()` function when provided with two arguments and mimics the `locate()` function when provided with three arguments. SQL Server also doesn't include a `position()` or `locate()` function, but it does include the `charindx()` function, which also accepts either two or three arguments similar to Oracle's `instr()` function.

Another function that takes strings as arguments and returns numbers is the string comparison function `strcmp()`. `Strcmp()`, which is implemented only by MySQL and has no analog in Oracle Database or SQL Server, takes two strings as arguments, and returns one of the following:

- `-1` if the first string comes before the second string in sort order
- `0` if the strings are identical
- `1` if the first string comes after the second string in sort order

To illustrate how the function works, I first show the sort order of five strings using a query, and then show how the strings compare to one another using `strcmp()`. Here are the five strings that I insert into the `string_tbl` table:

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('abcd');
Query OK, 1 row affected (0.03 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('xyz');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('QRSTUV');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('qrstuv');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('12345');
Query OK, 1 row affected (0.00 sec)
```

Here are the five strings in their sort order:

```
mysql> SELECT vchar_fld
    -> FROM string_tbl
    -> ORDER BY vchar_fld;
+-----------+
| vchar_fld |
+-----------+
| 12345     |
| abcd      |
| QRSTUV    |
| qrstuv    |
| xyz       |
+-----------+
5 rows in set (0.00 sec)
```

The next query makes six comparisons among the five different strings:

```
mysql> SELECT STRCMP('12345','12345') 12345_12345,
    ->   STRCMP('abcd','xyz') abcd_xyz,
    ->   STRCMP('abcd','QRSTUV') abcd_QRSTUV,
    ->   STRCMP('qrstuv','QRSTUV') qrstuv_QRSTUV,
    ->   STRCMP('12345','xyz') 12345_xyz,
    ->   STRCMP('xyz','qrstuv') xyz_qrstuv;
+-------------+----------+-------------+---------------+-----------+------------+
| 12345_12345 | abcd_xyz | abcd_QRSTUV | qrstuv_QRSTUV | 12345_xyz | xyz_qrstuv |
+-------------+----------+-------------+---------------+-----------+------------+
|           0 |       -1 |          -1 |             0 |        -1 |          1 |
+-------------+----------+-------------+---------------+-----------+------------+
1 row in set (0.00 sec)
```

The first comparison yields `0`, which is to be expected since I compared a string to itself. The fourth comparison also yields `0`, which is a bit surprising, since the strings are composed of the same letters, with one string all uppercase and the other all lowercase. The reason for this result is that MySQL's `strcmp()` function is case-insensitive, which is something to remember when using the function. The other four comparisons yield either `-1` or `1` depending on whether the first string comes before or after the second string in sort order. For example, `strcmp('abcd','xyz')` yields `-1`, since the string `'abcd'` comes before the string `'xyz'`.

Along with the `strcmp()` function, MySQL also allows you to use the `like` and `regexp` operators to compare strings in the `select` clause. Such comparisons will yield `1` (for `true`) or `0` (for `false`). Therefore, these operators allow you to build expressions that return a number, much like the functions described in this section. Here's an example using `like`:

```
mysql> SELECT name, name LIKE '%ns' ends_in_ns
    -> FROM department;
+----------------+------------+
| name           | ends_in_ns |
+----------------+------------+
| Operations     |          1 |
| Loans          |          1 |
| Administration |          0 |
+----------------+------------+
3 rows in set (0.25 sec)
```

This example retrieves all the department names, along with an expression that returns `1` if the department name ends in "ns" or `0` otherwise. If you want to perform more complex pattern matches, you can use the `regexp` operator, as demonstrated by the following:

```
mysql> SELECT cust_id, cust_type_cd, fed_id,
    ->   fed_id REGEXP '.{3}-.{2}-.{4}' is_ss_no_format
    -> FROM customer;
+---------+--------------+-------------+-----------------+
| cust_id | cust_type_cd | fed_id      | is_ss_no_format |
+---------+--------------+-------------+-----------------+
|       1 | I            | 111-11-1111 |               1 |
|       2 | I            | 222-22-2222 |               1 |
|       3 | I            | 333-33-3333 |               1 |
|       4 | I            | 444-44-4444 |               1 |
|       5 | I            | 555-55-5555 |               1 |
|       6 | I            | 666-66-6666 |               1 |
|       7 | I            | 777-77-7777 |               1 |
|       8 | I            | 888-88-8888 |               1 |
|       9 | I            | 999-99-9999 |               1 |
|      10 | B            | 04-1111111  |               0 |
|      11 | B            | 04-2222222  |               0 |
|      12 | B            | 04-3333333  |               0 |
|      13 | B            | 04-4444444  |               0 |
+---------+--------------+-------------+-----------------+
13 rows in set (0.00 sec)
```

The fourth column of this query returns `1` if the value stored in the `fed_id` column matches the format for a Social Security number.

SQL Server and Oracle Database users can achieve similar results by building `case` expressions, which I describe in detail in Chapter 11.

String functions that return strings

In some cases, you will need to modify existing strings, either by extracting part of the string or by adding additional text to the string. Every database server includes multiple functions to help with these tasks. Before I begin, I once again reset the data in the `string_tbl` table:

```
mysql> DELETE FROM string_tbl;
Query OK, 5 rows affected (0.00 sec)

mysql> INSERT INTO string_tbl (text_fld)
    -> VALUES ('This string was 29 characters');
Query OK, 1 row affected (0.01 sec)
```

Earlier in the chapter, I demonstrated the use of the `concat()` function to help build words that include accented characters. The `concat()` function is useful in many other situations, including when you need to append additional characters to a stored string. For instance, the following example modifies the string stored in the `text_fld` column by tacking an additional phrase on the end:

```
mysql> UPDATE string_tbl
    -> SET text_fld = CONCAT(text_fld, ', but now it is longer');
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

The contents of the `text_fld` column are now as follows:

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+-----------------------------------------------------+
| text_fld                                            |
+-----------------------------------------------------+
| This string was 29 characters, but now it is longer |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

Thus, like all functions that return a string, you can use `concat()` to replace the data stored in a character column.

Another common use for the `concat()` function is to build a string from individual pieces of data. For example, the following query generates a narrative string for each bank teller:

```
mysql> SELECT CONCAT(fname, ' ', lname, ' has been a ',
    ->   title, ' since ', start_date) emp_narrative
    -> FROM employee
    -> WHERE title = 'Teller' OR title = 'Head Teller';
+-----------------------------------------------------------+
| emp_narrative                                             |
+-----------------------------------------------------------+
| Helen Fleming has been a Head Teller since 2008-03-17     |
| Chris Tucker has been a Teller since 2008-09-15           |
| Sarah Parker has been a Teller since 2006-12-02           |
| Jane Grossman has been a Teller since 2006-05-03          |
| Paula Roberts has been a Head Teller since 2006-07-27     |
```

```
| Thomas Ziegler has been a Teller since 2004-10-23       |
| Samantha Jameson has been a Teller since 2007-01-08     |
| John Blake has been a Head Teller since 2004-05-11      |
| Cindy Mason has been a Teller since 2006-08-09          |
| Frank Portman has been a Teller since 2007-04-01        |
| Theresa Markham has been a Head Teller since 2005-03-15 |
| Beth Fowler has been a Teller since 2006-06-29          |
| Rick Tulman has been a Teller since 2006-12-12          |
+---------------------------------------------------------+
13 rows in set (0.30 sec)
```

The `concat()` function can handle any expression that returns a string, and will even convert numbers and dates to string format, as evidenced by the date column (`start_date`) used as an argument. Although Oracle Database includes the `concat()` function, it will accept only two string arguments, so the previous query will not work on Oracle. Instead, you would need to use the concatenation operator (`||`) rather than a function call, as in:

```
SELECT fname || ' ' || lname || ' has been a ' ||
  title || ' since ' || start_date emp_narrative
FROM employee
WHERE title = 'Teller' OR title = 'Head Teller';
```

SQL Server does not include a `concat()` function, so you would need to use the same approach as the previous query, except that you would use SQL Server's concatenation operator (`+`) instead of `||`.

While `concat()` is useful for adding characters to the beginning or end of a string, you may also have a need to add or replace characters in the *middle* of a string. All three database servers provide functions for this purpose, but all of them are different, so I demonstrate the MySQL function and then show the functions from the other two servers.

MySQL includes the `insert()` function, which takes four arguments: the original string, the position at which to start, the number of characters to replace, and the replacement string. Depending on the value of the third argument, the function may be used to either insert or replace characters in a string. With a value of `0` for the third argument, the replacement string is inserted and any trailing characters are pushed to the right, as in:

```
mysql> SELECT INSERT('goodbye world', 9, 0, 'cruel ') string;
+---------------------+
| string              |
+---------------------+
| goodbye cruel world |
+---------------------+
1 row in set (0.00 sec)
```

In this example, all characters starting from position 9 are pushed to the right and the string `'cruel'` is inserted. If the third argument is greater than zero, then that number of characters is replaced with the replacement string, as in:

```
mysql> SELECT INSERT('goodbye world', 1, 7, 'hello') string;
+-------------+
| string      |
+-------------+
| hello world |
+-------------+
1 row in set (0.00 sec)
```

For this example, the first seven characters are replaced with the string `'hello'`. Oracle Database does not provide a single function with the flexibility of MySQL's `insert()` function, but Oracle does provide the `replace()` function, which is useful for replacing one substring with another. Here's the previous example reworked to use `replace()`:

```
SELECT REPLACE('goodbye world', 'goodbye', 'hello')
FROM dual;
```

All instances of the string `'goodbye'` will be replaced with the string `'hello'`, resulting in the string `'hello world'`. The `replace()` function will replace *every* instance of the search string with the replacement string, so you need to be careful that you don't end up with more replacements than you anticipated.

SQL Server also includes a `replace()` function with the same functionality as Oracle's, but SQL Server also includes a function called `stuff()` with similar functionality to MySQL's `insert()` function. Here's an example:

```
SELECT STUFF('hello world', 1, 5, 'goodbye cruel')
```

When executed, five characters are removed starting at position 1, and then the string `'goodbye cruel'` is inserted at the starting position, resulting in the string `'goodbye cruel world'`.

Along with inserting characters into a string, you may have a need to *extract* a substring from a string. For this purpose, all three servers include the `substring()` function (although Oracle Database's version is called `substr()`), which extracts a specified number of characters starting at a specified position. The following example extracts five characters from a string starting at the ninth position:

```
mysql> SELECT SUBSTRING('goodbye cruel world', 9, 5);
+---------------------------------------+
| SUBSTRING('goodbye cruel world', 9, 5) |
+---------------------------------------+
| cruel                                 |
+---------------------------------------+
1 row in set (0.00 sec)
```

Along with the functions demonstrated here, all three servers include many more built-in functions for manipulating string data. While many of them are designed for very specific purposes, such as generating the string equivalent of octal or hexadecimal numbers, there are many other general-purpose functions as well, such as functions that remove or add trailing spaces. For more information, consult your server's SQL reference guide, or a general-purpose SQL reference guide such as *SQL in a Nutshell* (O'Reilly).

Working with Numeric Data

Unlike string data (and temporal data, as you will see shortly), numeric data generation is quite straightforward. You can type a number, retrieve it from another column, or generate it via a calculation. All the usual arithmetic operators (+, -, *, /) are available for performing calculations, and parentheses may be used to dictate precedence, as in:

```
mysql> SELECT (37 * 59) / (78 - (8 * 6));
+----------------------------+
| (37 * 59) / (78 - (8 * 6)) |
+----------------------------+
|                      72.77 |
+----------------------------+
1 row in set (0.00 sec)
```

As I mentioned in Chapter 2, the main concern when storing numeric data is that numbers might be rounded if they are larger than the specified size for a numeric column. For example, the number 9.96 will be rounded to 10.0 if stored in a column defined as `float(3,1)`.

Performing Arithmetic Functions

Most of the built-in numeric functions are used for specific arithmetic purposes, such as determining the square root of a number. Table 7-1 lists some of the common numeric functions that take a single numeric argument and return a number.

Table 7-1. Single-argument numeric functions

Function name	Description
`Acos(`*x*`)`	Calculates the arc cosine of *x*
`Asin(`*x*`)`	Calculates the arc sine of *x*
`Atan(`*x*`)`	Calculates the arc tangent of *x*
`Cos(`*x*`)`	Calculates the cosine of *x*
`Cot(`*x*`)`	Calculates the cotangent of *x*
`Exp(`*x*`)`	Calculates e^x
`Ln(`*x*`)`	Calculates the natural log of *x*
`Sin(`*x*`)`	Calculates the sine of *x*
`Sqrt(`*x*`)`	Calculates the square root of *x*
`Tan(`*x*`)`	Calculates the tangent of *x*

These functions perform very specific tasks, and I refrain from showing examples for these functions (if you don't recognize a function by name or description, then you probably don't need it). Other numeric functions used for calculations, however, are a bit more flexible and deserve some explanation.

For example, the `modulo` operator, which calculates the remainder when one number is divided into another number, is implemented in MySQL and Oracle Database via the `mod()` function. The following example calculates the remainder when `4` is divided into `10`:

```
mysql> SELECT MOD(10,4);
+-----------+
| MOD(10,4) |
+-----------+
|         2 |
+-----------+
1 row in set (0.02 sec)
```

While the `mod()` function is typically used with integer arguments, with MySQL you can also use real numbers, as in:

```
mysql> SELECT MOD(22.75, 5);
+---------------+
| MOD(22.75, 5) |
+---------------+
|          2.75 |
+---------------+
1 row in set (0.02 sec)
```

SQL Server does not have a `mod()` function. Instead, the operator `%` is used for finding remainders. The expression `10 % 4` will therefore yield the value `2`.

Another numeric function that takes two numeric arguments is the `pow()` function (or `power()` if you are using Oracle Database or SQL Server), which returns one number raised to the power of a second number, as in:

```
mysql> SELECT POW(2,8);
+----------+
| POW(2,8) |
+----------+
|      256 |
+----------+
1 row in set (0.03 sec)
```

Thus, `pow(2,8)` is the MySQL equivalent of specifying 2^8. Since computer memory is allocated in chunks of 2^x bytes, the `pow()` function can be a handy way to determine the exact number of bytes in a certain amount of memory:

```
mysql> SELECT POW(2,10) kilobyte, POW(2,20) megabyte,
    ->   POW(2,30) gigabyte, POW(2,40) terabyte;
+----------+----------+------------+---------------+
| kilobyte | megabyte | gigabyte   | terabyte      |
+----------+----------+------------+---------------+
|     1024 |  1048576 | 1073741824 | 1099511627776 |
+----------+----------+------------+---------------+
1 row in set (0.00 sec)
```

I don't know about you, but I find it easier to remember that a gigabyte is 2^{30} bytes than to remember the number 1,073,741,824.

Controlling Number Precision

When working with floating-point numbers, you may not always want to interact with or display a number with its full precision. For example, you may store monetary transaction data with a precision to six decimal places, but you might want to round to the nearest hundredth for display purposes. Four functions are useful when limiting the precision of floating-point numbers: `ceil()`, `floor()`, `round()`, and `truncate()`. All three servers include these functions, although Oracle Database includes `trunc()` instead of `truncate()`, and SQL Server includes `ceiling()` instead of `ceil()`.

The `ceil()` and `floor()` functions are used to round either up or down to the closest integer, as demonstrated by the following:

```
mysql> SELECT CEIL(72.445), FLOOR(72.445);
+-------------+--------------+
| CEIL(72.445) | FLOOR(72.445) |
+--------------+--------------+
|           73 |            72 |
+--------------+--------------+
1 row in set (0.06 sec)
```

Thus, any number between 72 and 73 will be evaluated as 73 by the `ceil()` function and 72 by the `floor()` function. Remember that `ceil()` will round up even if the decimal portion of a number is very small, and `floor()` will round down even if the decimal portion is quite significant, as in:

```
mysql> SELECT CEIL(72.000000001), FLOOR(72.999999999);
+--------------------+---------------------+
| CEIL(72.000000001) | FLOOR(72.999999999) |
+--------------------+---------------------+
|                 73 |                  72 |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

If this is a bit too severe for your application, you can use the `round()` function to round up or down from the *midpoint* between two integers, as in:

```
mysql> SELECT ROUND(72.49999), ROUND(72.5), ROUND(72.50001);
+-----------------+------------+-----------------+
| ROUND(72.49999) | ROUND(72.5) | ROUND(72.50001) |
+-----------------+------------+-----------------+
|              72 |          73 |              73 |
+-----------------+------------+-----------------+
1 row in set (0.00 sec)
```

Using `round()`, any number whose decimal portion is halfway or more between two integers will be rounded up, whereas the number will be rounded down if the decimal portion is anything less than halfway between the two integers.

Most of the time, you will want to keep at least some part of the decimal portion of a number rather than rounding to the nearest integer; the `round()` function allows an optional second argument to specify how many digits to the right of the decimal place to round to. The next example shows how you can use the second argument to round the number 72.0909 to one, two, and three decimal places:

```
mysql> SELECT ROUND(72.0909, 1), ROUND(72.0909, 2), ROUND(72.0909, 3);
+------------------+------------------+------------------+
| ROUND(72.0909, 1) | ROUND(72.0909, 2) | ROUND(72.0909, 3) |
+------------------+------------------+------------------+
|             72.1 |            72.09 |           72.091 |
+------------------+------------------+------------------+
1 row in set (0.00 sec)
```

Like the `round()` function, the `truncate()` function allows an optional second argument to specify the number of digits to the right of the decimal, but `truncate()` simply discards the unwanted digits without rounding. The next example shows how the number 72.0909 would be truncated to one, two, and three decimal places:

```
mysql> SELECT TRUNCATE(72.0909, 1), TRUNCATE(72.0909, 2),
    ->   TRUNCATE(72.0909, 3);
+----------------------+----------------------+----------------------+
| TRUNCATE(72.0909, 1) | TRUNCATE(72.0909, 2) | TRUNCATE(72.0909, 3) |
+----------------------+----------------------+----------------------+
|                 72.0 |                72.09 |               72.090 |
+----------------------+----------------------+----------------------+
1 row in set (0.00 sec)
```

SQL Server does not include a `truncate()` function. Instead, the `round()` function allows for an optional third argument which, if present and nonzero, calls for the number to be truncated rather than rounded.

Both `truncate()` and `round()` also allow a *negative* value for the second argument, meaning that numbers to the *left* of the decimal place are truncated or rounded. This might seem like a strange thing to do at first, but there are valid applications. For example, you might sell a product that can be purchased only in units of 10. If a customer were to order 17 units, you could choose from one of the following methods to modify the customer's order quantity:

```
mysql> SELECT ROUND(17, -1), TRUNCATE(17, -1);
+---------------+-----------------+
| ROUND(17, -1) | TRUNCATE(17, -1) |
+---------------+-----------------+
|            20 |               10 |
+---------------+-----------------+
1 row in set (0.00 sec)
```

If the product in question is thumbtacks, then it might not make much difference to your bottom line whether you sold the customer 10 or 20 thumbtacks when only 17

were requested; if you are selling Rolex watches, however, your business may fare better by rounding.

Handling Signed Data

If you are working with numeric columns that allow negative values (in Chapter 2, I showed how a numeric column may be labeled *unsigned*, meaning that only positive numbers are allowed), several numeric functions might be of use. Let's say, for example, that you are asked to generate a report showing the current status of each bank account. The following query returns three columns useful for generating the report:

```
mysql> SELECT account_id, SIGN(avail_balance), ABS(avail_balance)
    -> FROM account;
+------------+---------------------+--------------------+
| account_id | SIGN(avail_balance) | ABS(avail_balance) |
+------------+---------------------+--------------------+
|          1 |                   1 |            1057.75 |
|          2 |                   1 |             500.00 |
|          3 |                   1 |            3000.00 |
|          4 |                   1 |            2258.02 |
|          5 |                   1 |             200.00 |
| ...                                                   |
|         19 |                   1 |            1500.00 |
|         20 |                   1 |           23575.12 |
|         21 |                   0 |               0.00 |
|         22 |                   1 |            9345.55 |
|         23 |                   1 |           38552.05 |
|         24 |                   1 |           50000.00 |
+------------+---------------------+--------------------+
24 rows in set (0.00 sec)
```

The second column uses the `sign()` function to return `-1` if the account balance is negative, `0` if the account balance is zero, and `1` if the account balance is positive. The third column returns the absolute value of the account balance via the `abs()` function.

Working with Temporal Data

Of the three types of data discussed in this chapter (character, numeric, and temporal), temporal data is the most involved when it comes to data generation and manipulation. Some of the complexity of temporal data is caused by the myriad ways in which a single date and time can be described. For example, the date on which I wrote this paragraph can be described in all the following ways:

- Wednesday, September 17, 2008
- 9/17/2008 2:14:56 P.M. EST
- 9/17/2008 19:14:56 GMT
- 2612008 (Julian format)
- Star date [–4] 85712.03 14:14:56 (*Star Trek* format)

While some of these differences are purely a matter of formatting, most of the complexity has to do with your frame of reference, which we explore in the next section.

Dealing with Time Zones

Because people around the world prefer that noon coincides roughly with the sun's peak at their location, there has never been a serious attempt to coerce everyone to use a universal clock. Instead, the world has been sliced into 24 imaginary sections, called *time zones*; within a particular time zone, everyone agrees on the current time, whereas people in different time zones do not. While this seems simple enough, some geographic regions shift their time by one hour twice a year (implementing what is known as *daylight saving time*) and some do not, so the time difference between two points on Earth might be four hours for one half of the year and five hours for the other half of the year. Even within a single time zone, different regions may or may not adhere to daylight saving time, causing different clocks in the same time zone to agree for one half of the year but be one hour different for the rest of the year.

While the computer age has exacerbated the issue, people have been dealing with time zone differences since the early days of naval exploration. To ensure a common point of reference for timekeeping, fifteenth-century navigators set their clocks to the time of day in Greenwich, England. This became known as *Greenwich Mean Time*, or GMT. All other time zones can be described by the number of hours' difference from GMT; for example, the time zone for the Eastern United States, known as *Eastern Standard Time*, can be described as GMT –5:00, or five hours earlier than GMT.

Today, we use a variation of GMT called *Coordinated Universal Time*, or UTC, which is based on an atomic clock (or, to be more precise, the average time of 200 atomic clocks in 50 locations worldwide, which is referred to as *Universal Time*). Both SQL Server and MySQL provide functions that will return the current UTC timestamp (`getutcdate()` for SQL Server and `utc_timestamp()` for MySQL).

Most database servers default to the time zone setting of the server on which it resides and provide tools for modifying the time zone if needed. For example, a database used to store stock exchange transactions from around the world would generally be configured to use UTC time, whereas a database used to store transactions at a particular retail establishment might use the server's time zone.

MySQL keeps two different time zone settings: a global time zone, and a session time zone, which may be different for each user logged in to a database. You can see both settings via the following query:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | SYSTEM              |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

A value of `system` tells you that the server is using the time zone setting from the server on which the database resides.

If you are sitting at a computer in Zurich, Switzerland, and you open a session across the network to a MySQL server situated in New York, you may want to change the time zone setting for your session, which you can do via the following command:

```
mysql> SET time_zone = 'Europe/Zurich';
Query OK, 0 rows affected (0.18 sec)
```

If you check the time zone settings again, you will see the following:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | Europe/Zurich       |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

All dates displayed in your session will now conform to Zurich time.

Oracle Database users can change the time zone setting for a session via the following command:

```
ALTER SESSION TIMEZONE = 'Europe/Zurich'
```

Generating Temporal Data

You can generate temporal data via any of the following means:

- Copying data from an existing `date`, `datetime`, or `time` column
- Executing a built-in function that returns a `date`, `datetime`, or `time`
- Building a string representation of the temporal data to be evaluated by the server

To use the last method, you will need to understand the various components used in formatting dates.

String representations of temporal data

Table 2-5 in Chapter 2 presented the more popular date components; to refresh your memory, Table 7-2 shows these same components.

Loading MySQL Time Zone Data

If you are running the MySQL server on a Windows platform, you will need to load time zone data manually before you can set global or session time zones. To do so, you need to follow these steps:

1. Download the time zone data from *http://dev.mysql.com/downloads/timezones.html*.
2. Shut down your MySQL server.
3. Extract the files from the downloaded ZIP file (in my case, the file was called *timezone-2006p.zip*) and place them in your MySQL installation directory under */data/mysql* (the full path for my installation was */Program Files/MySQL/MySQL Server 6.0/data/mysql*).
4. Restart your MySQL server.

   To look at the time zone data, change to the *mysql* database via the `use mysql` command, and execute the following query:

```
mysql> SELECT name FROM time_zone_name;
+----------------------------------+
| name                             |
+----------------------------------+
| Africa/Abidjan                   |
| Africa/Accra                     |
| Africa/Addis_Ababa               |
| Africa/Algiers                   |
| Africa/Asmera                    |
| Africa/Bamako                    |
| Africa/Bangui                    |
| Africa/Banjul                    |
| Africa/Bissau                    |
| Africa/Blantyre                  |
| Africa/Brazzaville               |
| Africa/Bujumbura                 |
...
| US/Alaska                        |
| US/Aleutian                      |
| US/Arizona                       |
| US/Central                       |
| US/East-Indiana                  |
| US/Eastern                       |
| US/Hawaii                        |
| US/Indiana-Starke                |
| US/Michigan                      |
| US/Mountain                      |
| US/Pacific                       |
| US/Pacific-New                   |
| US/Samoa                         |
| UTC                              |
| W-SU                             |
| WET                              |
| Zulu                             |
+----------------------------------+
546 rows in set (0.01 sec)
```

   To change your time zone setting, choose one of the names from the previous query that best matches your location.

Table 7-2. Date format components

Component	Definition	Range
YYYY	Year, including century	1000 to 9999
MM	Month	01 (January) to 12 (December)
DD	Day	01 to 31
HH	Hour	00 to 23
HHH	Hours (elapsed)	-838 to 838
MI	Minute	00 to 59
SS	Second	00 to 59

To build a string that the server can interpret as a `date`, `datetime`, or `time`, you need to put the various components together in the order shown in Table 7-3.

Table 7-3. Required date components

Type	Default format
`Date`	YYYY-MM-DD
`Datetime`	YYYY-MM-DD HH:MI:SS
`Timestamp`	YYYY-MM-DD HH:MI:SS
`Time`	HHH:MI:SS

Thus, to populate a `datetime` column with 3:30 P.M. on September 17, 2008, you will need to build the following string:

```
'2008-09-17 15:30:00'
```

If the server is expecting a `datetime` value, such as when updating a `datetime` column or when calling a built-in function that takes a `datetime` argument, you can provide a properly formatted string with the required date components, and the server will do the conversion for you. For example, here's a statement used to modify the date of a bank transaction:

```
UPDATE transaction
SET txn_date = '2008-09-17 15:30:00'
WHERE txn_id = 99999;
```

The server determines that the string provided in the `set` clause must be a `datetime` value, since the string is being used to populate a `datetime` column. Therefore, the server will attempt to convert the string for you by parsing the string into the six components (year, month, day, hour, minute, second) included in the default `datetime` format.

String-to-date conversions

If the server is *not* expecting a `datetime` value, or if you would like to represent the `datetime` using a nondefault format, you will need to tell the server to convert the string

to a `datetime`. For example, here is a simple query that returns a `datetime` value using the `cast()` function:

```
mysql> SELECT CAST('2008-09-17 15:30:00' AS DATETIME);
+---------------------------------------+
| CAST('2008-09-17 15:30:00' AS DATETIME) |
+---------------------------------------+
| 2008-09-17 15:30:00                   |
+---------------------------------------+
1 row in set (0.00 sec)
```

We cover the `cast()` function at the end of this chapter. While this example demonstrates how to build `datetime` values, the same logic applies to the `date` and `time` types as well. The following query uses the `cast()` function to generate a `date` value and a `time` value:

```
mysql> SELECT CAST('2008-09-17' AS DATE) date_field,
    ->   CAST('108:17:57' AS TIME) time_field;
+------------+------------+
| date_field | time_field |
+------------+------------+
| 2008-09-17 | 108:17:57  |
+------------+------------+
1 row in set (0.00 sec)
```

You may, of course, explicitly convert your strings even when the server is expecting a `date`, `datetime`, or `time` value, rather than letting the server do an implicit conversion.

When strings are converted to temporal values—whether explicitly or implicitly—you must provide all the date components in the required order. While some servers are quite strict regarding the date format, the MySQL server is quite lenient about the separators used between the components. For example, MySQL will accept all of the following strings as valid representations of 3:30 P.M. on September 17, 2008:

```
'2008-09-17 15:30:00'
'2008/09/17 15:30:00'
'2008,09,17,15,30,00'
'20080917153000'
```

Although this gives you a bit more flexibility, you may find yourself trying to generate a temporal value *without* the default date components; the next section demonstrates a built-in function that is far more flexible than the `cast()` function.

Functions for generating dates

If you need to generate temporal data from a string, and the string is not in the proper form to use the `cast()` function, you can use a built-in function that allows you to provide a format string along with the date string. MySQL includes the `str_to_date()` function for this purpose. Say, for example, that you pull the string `'September 17, 2008'` from a file and need to use it to update a `date` column. Since the string is not in the required YYYY-MM-DD format, you can use `str_to_date()` instead of reformatting the string so that you can use the `cast()` function, as in:

```
UPDATE individual
SET birth_date = STR_TO_DATE('September 17, 2008', '%M %d, %Y')
WHERE cust_id = 9999;
```

The second argument in the call to `str_to_date()` defines the format of the date string, with, in this case, a month name (`%M`), a numeric day (`%d`), and a four-digit numeric year (`%Y`). While there are over 30 recognized format components, Table 7-4 defines the dozen or so most commonly used components.

Table 7-4. Date format components

Format component	Description
`%M`	Month name (`January` to `December`)
`%m`	Month numeric (`01` to `12`)
`%d`	Day numeric (`01` to `31`)
`%j`	Day of year (`001` to `366`)
`%W`	Weekday name (`Sunday` to `Saturday`)
`%Y`	Year, four-digit numeric
`%y`	Year, two-digit numeric
`%H`	Hour (`00` to `23`)
`%h`	Hour (`01` to `12`)
`%i`	Minutes (`00` to `59`)
`%s`	Seconds (`00` to `59`)
`%f`	Microseconds (`000000` to `999999`)
`%p`	A.M. or P.M.

The `str_to_date()` function returns a `datetime`, `date`, or `time` value depending on the contents of the format string. For example, if the format string includes only `%H`, `%i`, and `%s`, then a `time` value will be returned.

Oracle Database users can use the `to_date()` function in the same manner as MySQL's `str_to_date()` function. SQL Server includes a `convert()` function that is not quite as flexible as MySQL and Oracle Database; rather than supplying a custom format string, your date string must conform to one of 21 predefined formats.

If you are trying to generate the *current* date/time, then you won't need to build a string, because the following built-in functions will access the system clock and return the current date and/or time as a string for you:

```
mysql> SELECT CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP();
+----------------+---------------+---------------------+
| CURRENT_DATE() | CURRENT_TIME() | CURRENT_TIMESTAMP() |
+----------------+---------------+---------------------+
| 2008-09-18     | 19:53:12      | 2008-09-18 19:53:12 |
+----------------+---------------+---------------------+
1 row in set (0.12 sec)
```

The values returned by these functions are in the default format for the temporal type being returned. Oracle Database includes `current_date()` and `current_timestamp()` but not `current_time()`, and SQL Server includes only the `current_timestamp()` function.

Manipulating Temporal Data

This section explores the built-in functions that take date arguments and return dates, strings, or numbers.

Temporal functions that return dates

Many of the built-in temporal functions take one date as an argument and return another date. MySQL's `date_add()` function, for example, allows you to add any kind of interval (e.g., days, months, years) to a specified date to generate another date. Here's an example that demonstrates how to add five days to the current date:

```
mysql> SELECT DATE_ADD(CURRENT_DATE(), INTERVAL 5 DAY);
+-----------------------------------------+
| DATE_ADD(CURRENT_DATE(), INTERVAL 5 DAY) |
+-----------------------------------------+
| 2008-09-22                              |
+-----------------------------------------+
1 row in set (0.06 sec)
```

The second argument is composed of three elements: the `interval` keyword, the desired quantity, and the type of interval. Table 7-5 shows some of the commonly used interval types.

Table 7-5. Common interval types

Interval name	Description
`Second`	Number of seconds
`Minute`	Number of minutes
`Hour`	Number of hours
`Day`	Number of days
`Month`	Number of months
`Year`	Number of years
`Minute_second`	Number of minutes and seconds, separated by ":"
`Hour_second`	Number of hours, minutes, and seconds, separated by ":"
`Year_month`	Number of years and months, separated by "-"

While the first six types listed in Table 7-5 are pretty straightforward, the last three types require a bit more explanation since they have multiple elements. For example, if you are told that transaction ID 9999 actually occurred 3 hours, 27 minutes, and 11 seconds later than what was posted to the `transaction` table, you can fix it via the following:

```
UPDATE transaction
SET txn_date = DATE_ADD(txn_date, INTERVAL '3:27:11' HOUR_SECOND)
WHERE txn_id = 9999;
```

In this example, the `date_add()` function takes the value in the `txn_date` column, adds 3 hours, 27 minutes, and 11 seconds to it, and uses the value that results to modify the `txn_date` column.

Or, if you work in HR and found out that employee ID 4789 claimed to be younger than he actually is, you could add 9 years and 11 months to his birth date, as in:

```
UPDATE employee
SET birth_date = DATE_ADD(birth_date, INTERVAL '9-11' YEAR_MONTH)
WHERE emp_id = 4789;
```

SQL Server users can accomplish the previous example using the `dateadd()` function:

```
UPDATE employee
SET birth_date =
  DATEADD(MONTH, 119, birth_date)
WHERE emp_id = 4789
```

SQL Server doesn't have combined intervals (i.e., `year_month`), so I converted 9 years, 11 months to 119 months.

Oracle Database users can use the `add_months()` function for this example, as in:

```
UPDATE employee
SET birth_date = ADD_MONTHS(birth_date, 119)
WHERE emp_id = 4789;
```

There are some cases where you want to add an interval to a date, and you know where you want to arrive but not how many days it takes to get there. For example, let's say that a bank customer logs on to the online banking system and schedules a transfer for the end of the month. Rather than writing some code that figures out what month you are currently in and looks up the number of days in that month, you can call the `last_day()` function, which does the work for you (both MySQL and Oracle Database include the `last_day()` function; SQL Server has no comparable function). If the customer asks for the transfer on September 17, 2008, you could find the last day of September via the following:

```
mysql> SELECT LAST_DAY('2008-09-17');
+-----------------------+
| LAST_DAY('2008-09-17') |
+-----------------------+
| 2008-09-30            |
+-----------------------+
1 row in set (0.10 sec)
```

Whether you provide a `date` or `datetime` value, the `last_day()` function always returns a `date`. Although this function may not seem like an enormous timesaver, the underlying logic can be tricky if you're trying to find the last day of February and need to figure out whether the current year is a leap year.

Another temporal function that returns a date is one that converts a `datetime` value from one time zone to another. For this purpose, MySQL includes the `convert_tz()` function and Oracle Database includes the `new_time()` function. If I want to convert my current local time to UTC, for example, I could do the following:

```
mysql> SELECT CURRENT_TIMESTAMP() current_est,
    ->   CONVERT_TZ(CURRENT_TIMESTAMP(), 'US/Eastern', 'UTC') current_utc;
+--------------------+--------------------+
| current_est        | current_utc        |
+--------------------+--------------------+
| 2008-09-18 20:01:25 | 2008-09-19 00:01:25 |
+--------------------+--------------------+
1 row in set (0.76 sec)
```

This function comes in handy when receiving dates in a different time zone than what is stored in your database.

Temporal functions that return strings

Most of the temporal functions that return string values are used to extract a portion of a date or time. For example, MySQL includes the `dayname()` function to determine which day of the week a certain date falls on, as in:

```
mysql> SELECT DAYNAME('2008-09-18');
+----------------------+
| DAYNAME('2008-09-18') |
+----------------------+
| Thursday             |
+----------------------+
1 row in set (0.08 sec)
```

Many such functions are included with MySQL for extracting information from date values, but I recommend that you use the `extract()` function instead, since it's easier to remember a few variations of one function than to remember a dozen different functions. Additionally, the `extract()` function is part of the SQL:2003 standard and has been implemented by Oracle Database as well as MySQL.

The `extract()` function uses the same interval types as the `date_add()` function (see Table 7-5) to define which element of the date interests you. For example, if you want to extract just the year portion of a `datetime` value, you can do the following:

```
mysql> SELECT EXTRACT(YEAR FROM '2008-09-18 22:19:05');
+-----------------------------------------+
| EXTRACT(YEAR FROM '2008-09-18 22:19:05') |
+-----------------------------------------+
|                                    2008 |
+-----------------------------------------+
1 row in set (0.00 sec)
```

SQL Server doesn't include an implementation of `extract()`, but it does include the `datepart()` function. Here's how you would extract the year from a `datetime` value using `datepart()`:

```
SELECT DATEPART(YEAR, GETDATE())
```

Temporal functions that return numbers

Earlier in this chapter, I showed you a function used to add a given interval to a date value, thus generating another date value. Another common activity when working with dates is to take *two* date values and determine the number of intervals (days, weeks, years) *between* the two dates. For this purpose, MySQL includes the function `datediff()`, which returns the number of full days between two dates. For example, if I want to know the number of days that my kids will be out of school this summer, I can do the following:

```
mysql> SELECT DATEDIFF('2009-09-03', '2009-06-24');
+-------------------------------------+
| DATEDIFF('2009-09-03', '2009-06-24') |
+-------------------------------------+
|                                  71 |
+-------------------------------------+
1 row in set (0.05 sec)
```

Thus, I will have to endure 71 days of poison ivy, mosquito bites, and scraped knees before the kids are safely back at school. The `datediff()` function ignores the time of day in its arguments. Even if I include a time-of-day, setting it to one second until midnight for the first date and to one second after midnight for the second date, those times will have no effect on the calculation:

```
mysql> SELECT DATEDIFF('2009-09-03 23:59:59', '2009-06-24 00:00:01');
+-------------------------------------------------------+
| DATEDIFF('2009-09-03 23:59:59', '2009-06-24 00:00:01') |
+-------------------------------------------------------+
|                                                    71 |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

If I switch the arguments and have the earlier date first, `datediff()` will return a negative number, as in:

```
mysql> SELECT DATEDIFF('2009-06-24', '2009-09-03');
+-------------------------------------+
| DATEDIFF('2009-06-24', '2009-09-03') |
+-------------------------------------+
|                                 -71 |
+-------------------------------------+
1 row in set (0.01 sec)
```

SQL Server also includes the `datediff()` function, but it is more flexible than the MySQL implementation in that you can specify the interval type (i.e., year, month, day, hour) instead of counting only the number of days between two dates. Here's how SQL Server would accomplish the previous example:

```
SELECT DATEDIFF(DAY, '2009-06-24', '2009-09-03')
```

Oracle Database allows you to determine the number of days between two dates simply by subtracting one date from another.

Conversion Functions

Earlier in this chapter, I showed you how to use the `cast()` function to convert a string to a `datetime` value. While every database server includes a number of proprietary functions used to convert data from one type to another, I recommend using the `cast()` function, which is included in the SQL:2003 standard and has been implemented by MySQL, Oracle Database, and Microsoft SQL Server.

To use `cast()`, you provide a value or expression, the `as` keyword, and the type to which you want the value converted. Here's an example that converts a string to an integer:

```
mysql> SELECT CAST('1456328' AS SIGNED INTEGER);
+-----------------------------------+
| CAST('1456328' AS SIGNED INTEGER) |
+-----------------------------------+
|                           1456328 |
+-----------------------------------+
1 row in set (0.01 sec)
```

When converting a string to a number, the `cast()` function will attempt to convert the entire string from left to right; if any non-numeric characters are found in the string, the conversion halts without an error. Consider the following example:

```
mysql> SELECT CAST('999ABC111' AS UNSIGNED INTEGER);
+---------------------------------------+
| CAST('999ABC111' AS UNSIGNED INTEGER) |
+---------------------------------------+
|                                   999 |
+---------------------------------------+
1 row in set, 1 warning (0.08 sec)
```

```
mysql> show warnings;
+---------+------+-----------------------------------------------+
| Level   | Code | Message                                       |
+---------+------+-----------------------------------------------+
| Warning | 1292 | Truncated incorrect INTEGER value: '999ABC111' |
+---------+------+-----------------------------------------------+
1 row in set (0.07 sec)
```

In this case, the first three digits of the string are converted, whereas the rest of the string is discarded, resulting in a value of `999`. The server did, however, issue a warning to let you know that not all the string was converted.

If you are converting a string to a `date`, `time`, or `datetime` value, then you will need to stick with the default formats for each type, since you can't provide the `cast()` function with a format string. If your date string is not in the default format (i.e., YYYY-MM-DD HH:MI:SS for `datetime` types), then you will need to resort to using another function, such as MySQL's `str_to_date()` function described earlier in the chapter.

Test Your Knowledge

These exercises are designed to test your understanding of some of the built-in functions shown in this chapter. See Appendix C for the answers.

Exercise 7-1

Write a query that returns the 17th through 25th characters of the string `'Please find the substring in this string'`.

Exercise 7-2

Write a query that returns the absolute value and sign (`-1`, `0`, or `1`) of the number `-25.76823`. Also return the number rounded to the nearest hundredth.

Exercise 7-3

Write a query to return just the month portion of the current date.

CHAPTER 8

Grouping and Aggregates

Data is generally stored at the lowest level of granularity needed by any of a database's users; if Chuck in accounting needs to look at individual customer transactions, then there needs to be a table in the database that stores individual transactions. That doesn't mean, however, that all users must deal with the data as it is stored in the database. The focus of this chapter is on how data can be grouped and aggregated to allow users to interact with it at some higher level of granularity than what is stored in the database.

Grouping Concepts

Sometimes you will want to find trends in your data that will require the database server to cook the data a bit before you can generate the results you are looking for. For example, let's say that you are in charge of operations at the bank, and you would like to find out how many accounts are being opened by each bank teller. You could issue a simple query to look at the raw data:

```
mysql> SELECT open_emp_id
    -> FROM account;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          13 |
```

```
|          13 |
|          13 |
|          16 |
|          16 |
|          16 |
|          16 |
|          16 |
|          16 |
+-------------+
24 rows in set (0.01 sec)
```

With only 24 rows in the `account` table, it is relatively easy to see that four different employees opened accounts and that employee ID 16 has opened six accounts; however, if the bank has dozens of employees and thousands of accounts, this approach would prove tedious and error-prone.

Instead, you can ask the database server to group the data for you by using the `group by` clause. Here's the same query but employing a `group by` clause to group the account data by employee ID:

```
mysql> SELECT open_emp_id
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|          10 |
|          13 |
|          16 |
+-------------+
4 rows in set (0.00 sec)
```

The result set contains one row for each distinct value in the `open_emp_id` column, resulting in four rows instead of the full 24 rows. The reason for the smaller result set is that each of the four employees opened more than one account. To see how many accounts each teller opened, you can use an *aggregate function* in the `select` clause to count the number of rows in each group:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          13 |        3 |
|          16 |        6 |
+-------------+----------+
4 rows in set (0.00 sec)
```

The aggregate function `count()` counts the number of rows in each group, and the asterisk tells the server to count everything in the group. Using the combination of a

`group by` clause and the `count()` aggregate function, you are able to generate exactly the data needed to answer the business question without having to look at the raw data.

When grouping data, you may need to filter out undesired data from your result set based on groups of data rather than based on the raw data. Since the `group by` clause runs *after* the `where` clause has been evaluated, you cannot add filter conditions to your `where` clause for this purpose. For example, here's an attempt to filter out any cases where an employee has opened fewer than five accounts:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> WHERE COUNT(*) > 4
    -> GROUP BY open_emp_id;
ERROR 1111 (HY000): Invalid use of group function
```

You cannot refer to the aggregate function `count(*)` in your `where` clause, because the groups have not yet been generated at the time the `where` clause is evaluated. Instead, you must put your group filter conditions in the `having` clause. Here's what the query would look like using `having`:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> HAVING COUNT(*) > 4;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          16 |        6 |
+-------------+----------+
3 rows in set (0.00 sec)
```

Because those groups containing fewer than five members have been filtered out via the `having` clause, the result set now contains only those employees who have opened five or more accounts, thus eliminating employee ID 13 from the results.

Aggregate Functions

Aggregate functions perform a specific operation over all rows in a group. Although every database server has its own set of specialty aggregate functions, the common aggregate functions implemented by all major servers include:

`Max()`
: Returns the maximum value within a set

`Min()`
: Returns the minimum value within a set

`Avg()`
: Returns the average value across a set

`Sum()`
: Returns the sum of the values across a set

`Count()`
: Returns the number of values in a set

Here's a query that uses all of the common aggregate functions to analyze the available balances for all checking accounts:

```
mysql> SELECT MAX(avail_balance) max_balance,
    ->   MIN(avail_balance) min_balance,
    ->   AVG(avail_balance) avg_balance,
    ->   SUM(avail_balance) tot_balance,
    ->   COUNT(*) num_accounts
    -> FROM account
    -> WHERE product_cd = 'CHK';
+-------------+-------------+-------------+-------------+--------------+
| max_balance | min_balance | avg_balance | tot_balance | num_accounts |
+-------------+-------------+-------------+-------------+--------------+
|    38552.05 |      122.37 | 7300.800985 |    73008.01 |           10 |
+-------------+-------------+-------------+-------------+--------------+
1 row in set (0.09 sec)
```

The results from this query tell you that, across the 10 checking accounts in the `account` table, there is a maximum balance of $38,552.05, a minimum balance of $122.37, an average balance of $7,300.80, and a total balance across all 10 accounts of $73,008.01. Hopefully, this gives you an appreciation for the role of these aggregate functions; the next subsections further clarify how you can utilize these functions.

Implicit Versus Explicit Groups

In the previous example, every value returned by the query is generated by an aggregate function, and the aggregate functions are applied across the group of rows specified by the filter condition `product_cd = 'CHK'`. Since there is no `group by` clause, there is a single, *implicit* group (all rows returned by the query).

In most cases, however, you will want to retrieve additional columns along with columns generated by aggregate functions. What if, for example, you wanted to extend the previous query to execute the same five aggregate functions for *each* product type, instead of just for checking accounts? For this query, you would want to retrieve the `product_cd` column along with the five aggregate functions, as in:

```
SELECT product_cd,
  MAX(avail_balance) max_balance,
  MIN(avail_balance) min_balance,
  AVG(avail_balance) avg_balance,
  SUM(avail_balance) tot_balance,
  COUNT(*) num_accounts
FROM account;
```

However, if you try to execute the query, you will receive the following error:

```
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(),...) with no GROUP
columns is illegal if there is no GROUP BY clause
```

While it may be obvious to you that you want the aggregate functions applied to each set of products found in the `account` table, this query fails because you have not *explicitly* specified how the data should be grouped. Therefore, you will need to add a `group by` clause to specify over which group of rows the aggregate functions should be applied:

```
mysql> SELECT product_cd,
    ->   MAX(avail_balance) max_balance,
    ->   MIN(avail_balance) min_balance,
    ->   AVG(avail_balance) avg_balance,
    ->   SUM(avail_balance) tot_balance,
    ->   COUNT(*) num_accts
    -> FROM account
    -> GROUP BY product_cd;
+------------+-------------+-------------+--------------+-------------+-----------+
| product_cd | max_balance | min_balance | avg_balance  | tot_balance | num_accts |
+------------+-------------+-------------+--------------+-------------+-----------+
| BUS        |     9345.55 |        0.00 |  4672.774902 |     9345.55 |         2 |
| CD         |    10000.00 |     1500.00 |  4875.000000 |    19500.00 |         4 |
| CHK        |    38552.05 |      122.37 |  7300.800985 |    73008.01 |        10 |
| MM         |     9345.55 |     2212.50 |  5681.713216 |    17045.14 |         3 |
| SAV        |      767.77 |      200.00 |   463.940002 |     1855.76 |         4 |
| SBL        |    50000.00 |    50000.00 | 50000.000000 |    50000.00 |         1 |
+------------+-------------+-------------+--------------+-------------+-----------+
6 rows in set (0.00 sec)
```

With the inclusion of the `group by` clause, the server knows to group together rows having the same value in the `product_cd` column first and then to apply the five aggregate functions to each of the six groups.

Counting Distinct Values

When using the `count()` function to determine the number of members in each group, you have your choice of counting *all* members in the group, or counting only the *distinct* values for a column across all members of the group. For example, consider the following data, which shows the employee responsible for opening each account:

```
mysql> SELECT account_id, open_emp_id
    -> FROM account
    -> ORDER BY open_emp_id;
+------------+-------------+
| account_id | open_emp_id |
+------------+-------------+
|          8 |           1 |
|          9 |           1 |
|         10 |           1 |
|         12 |           1 |
|         13 |           1 |
|         17 |           1 |
|         18 |           1 |
```

```
|         19 |           1 |
|          1 |          10 |
|          2 |          10 |
|          3 |          10 |
|          4 |          10 |
|          5 |          10 |
|         14 |          10 |
|         22 |          10 |
|          6 |          13 |
|          7 |          13 |
|         24 |          13 |
|         11 |          16 |
|         15 |          16 |
|         16 |          16 |
|         20 |          16 |
|         21 |          16 |
|         23 |          16 |
+------------+-------------+
24 rows in set (0.00 sec)
```

As you can see, multiple accounts were opened by four different employees (employee IDs 1, 10, 13, and 16). Let's say that, instead of performing a manual count, you want to create a query that counts the number of employees who have opened accounts. If you apply the `count()` function to the `open_emp_id` column, you will see the following results:

```
mysql> SELECT COUNT(open_emp_id)
    -> FROM account;
+--------------------+
| COUNT(open_emp_id) |
+--------------------+
|                 24 |
+--------------------+
1 row in set (0.00 sec)
```

In this case, specifying the `open_emp_id` column as the column to be counted generates the same results as specifying `count(*)`. If you want to count *distinct* values in the group rather than just counting the number of rows in the group, you need to specify the `distinct` keyword, as in:

```
mysql> SELECT COUNT(DISTINCT open_emp_id)
    -> FROM account;
+-----------------------------+
| COUNT(DISTINCT open_emp_id) |
+-----------------------------+
|                           4 |
+-----------------------------+
1 row in set (0.00 sec)
```

By specifying `distinct`, therefore, the `count()` function examines the values of a column for each member of the group in order to find and remove duplicates, rather than simply counting the number of values in the group.

Using Expressions

Along with using columns as arguments to aggregate functions, you can build expressions to use as arguments. For example, you may want to find the maximum value of pending deposits across all accounts, which is calculated by subtracting the available balance from the pending balance. You can achieve this via the following query:

```
mysql> SELECT MAX(pending_balance - avail_balance) max_uncleared
    -> FROM account;
+---------------+
| max_uncleared |
+---------------+
|        660.00 |
+---------------+
1 row in set (0.00 sec)
```

While this example uses a fairly simple expression, expressions used as arguments to aggregate functions can be as complex as needed, as long as they return a number, string, or date. In Chapter 11, I show you how you can use `case` expressions with aggregate functions to determine whether a particular row should or should not be included in an aggregation.

How Nulls Are Handled

When performing aggregations, or, indeed, any type of numeric calculation, you should always consider how `null` values might affect the outcome of your calculation. To illustrate, I will build a simple table to hold numeric data and populate it with the set {1, 3, 5}:

```
mysql> CREATE TABLE number_tbl
    ->  (val SMALLINT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO number_tbl VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (5);
Query OK, 1 row affected (0.00 sec)
```

Consider the following query, which performs five aggregate functions on the set of numbers:

```
mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
```

```
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        3 |        3 |     9 |       5 |  3.0000 |
+----------+----------+-------+---------+---------+
1 row in set (0.08 sec)
```

The results are as you would expect: both `count(*)` and `count(val)` return the value `3`, `sum(val)` returns the value `9`, `max(val)` returns `5`, and `avg(val)` returns `3`. Next, I will add a `null` value to the `number_tbl` table and run the query again:

```
mysql> INSERT INTO number_tbl VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        4 |        3 |     9 |       5 |  3.0000 |
+----------+----------+-------+---------+---------+
1 row in set (0.00 sec)
```

Even with the addition of the `null` value to the table, the `sum()`, `max()`, and `avg()` functions all return the same values, indicating that they ignore any `null` values encountered. The `count(*)` function now returns the value `4`, which is valid since the `number_tbl` table contains four rows, while the `count(val)` function still returns the value `3`. The difference is that `count(*)` counts the number of rows, whereas `count(val)` counts the number of *values* contained in the `val` column and ignores any `null` values encountered.

Generating Groups

People are rarely interested in looking at raw data; instead, people engaging in data analysis will want to manipulate the raw data to better suit their needs. Examples of common data manipulations include:

- Generating totals for a geographic region, such as total European sales
- Finding outliers, such as the top salesperson for 2005
- Determining frequencies, such as the number of new accounts opened for each branch

To answer these types of queries, you will need to ask the database server to group rows together by one or more columns or expressions. As you have seen already in several examples, the `group by` clause is the mechanism for grouping data within a query. In this section, you will see how to group data by one or more columns, how to group data using expressions, and how to generate rollups within groups.

Single-Column Grouping

Single-column groups are the simplest and most-often-used type of grouping. If you want to find the total balances for each product, for example, you need only group on the `account.product_cd` column, as in:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> GROUP BY product_cd;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| BUS        |      9345.55 |
| CD         |     19500.00 |
| CHK        |     73008.01 |
| MM         |     17045.14 |
| SAV        |      1855.76 |
| SBL        |     50000.00 |
+------------+--------------+
6 rows in set (0.00 sec)
```

This query generates six groups, one for each product, and then sums the available balances for each member of the group.

Multicolumn Grouping

In some cases, you may want to generate groups that span *more* than one column. Expanding on the previous example, imagine that you want to find the total balances not just for each product, but for both products and branches (e.g., what's the total balance for all checking accounts opened at the Woburn branch?). The following example shows how you can accomplish this:

```
mysql> SELECT product_cd, open_branch_id,
    ->   SUM(avail_balance) tot_balance
    -> FROM account
    -> GROUP BY product_cd, open_branch_id;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SBL        |              3 |    50000.00 |
```

```
+------------+---------------+-------------+
14 rows in set (0.00 sec)
```

This version of the query generates 14 groups, one for each combination of product and branch found in the `account` table. Along with adding the `open_branch_id` column to the `select` clause, I also added it to the `group by` clause, since `open_branch_id` is retrieved from a table and is not generated via an aggregate function.

Grouping via Expressions

Along with using columns to group data, you can build groups based on the values generated by expressions. Consider the following query, which groups employees by the year they began working for the bank:

```
mysql> SELECT EXTRACT(YEAR FROM start_date) year,
    ->   COUNT(*) how_many
    -> FROM employee
    -> GROUP BY EXTRACT(YEAR FROM start_date);
+------+----------+
| year | how_many |
+------+----------+
| 2004 |        2 |
| 2005 |        3 |
| 2006 |        8 |
| 2007 |        3 |
| 2008 |        2 |
+------+----------+
5 rows in set (0.15 sec)
```

This query employs a fairly simple expression, which uses the `extract()` function to return only the year portion of a date, to group the rows in the `employee` table.

Generating Rollups

In "Multicolumn Grouping" on page 151, I showed an example that generates total account balances for each product and branch. Let's say, however, that along with the total balances for each product/branch combination, you also want total balances for each distinct product. You could run an additional query and merge the results, you could load the results of the query into a spreadsheet, or you could build a Perl script, Java program, or some other mechanism to take that data and perform the additional calculations. Better yet, you could use the `with rollup` option to have the database server do the work for you. Here's the revised query using `with rollup` in the `group by` clause:

```
mysql> SELECT product_cd, open_branch_id,
    ->   SUM(avail_balance) tot_balance
    -> FROM account
    -> GROUP BY product_cd, open_branch_id WITH ROLLUP;

+------------+---------------+-------------+
| product_cd | open_branch_id | tot_balance |
```

```
+------------+----------------+-------------+
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| BUS        |           NULL |     9345.55 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CD         |           NULL |    19500.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| CHK        |           NULL |    73008.01 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| MM         |           NULL |    17045.14 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SAV        |           NULL |     1855.76 |
| SBL        |              3 |    50000.00 |
| SBL        |           NULL |    50000.00 |
| NULL       |           NULL |   170754.46 |
+------------+----------------+-------------+
21 rows in set (0.02 sec)
```

There are now seven additional rows in the result set, one for each of the six distinct products and one for the grand total (all products combined). For the six product rollups, a `null` value is provided for the `open_branch_id` column, since the rollup is being performed across all branches. Looking at the third line of the output, for example, you will see that a total of $9,345.55 was deposited in BUS accounts across all branches. For the grand total row, a `null` value is provided for both the `product_cd` and `open_branch_id` columns; the last line of output shows a total of $170,754.46 across all products and branches.

If you are using Oracle Database, you need to use a slightly different syntax to indicate that you want a rollup performed. The `group by` clause for the previous query would look as follows when using Oracle:

```
GROUP BY ROLLUP(product_cd, open_branch_id)
```

The advantage of this syntax is that it allows you to perform rollups on a subset of the columns in the `group by` clause. If you are grouping by columns a, b, and c, for example, you could indicate that the server should perform rollups on only b and c via the following:

```
GROUP BY a, ROLLUP(b, c)
```

If, along with totals by product, you also want to calculate totals per branch, then you can use the `with cube` option, which generates summary rows for *all* possible combinations of the grouping columns. Unfortunately, `with cube` is not available in version 6.0 of MySQL, but it is available with SQL Server and Oracle Database. Here's an

example using `with cube`, but I have removed the `mysql>` prompt to show that the query cannot yet be performed with MySQL:

```
SELECT product_cd, open_branch_id,
  SUM(avail_balance) tot_balance
FROM account
GROUP BY product_cd, open_branch_id WITH CUBE;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| NULL       |           NULL |   170754.46 |
| NULL       |              1 |    27882.57 |
| NULL       |              2 |    21361.32 |
| NULL       |              3 |    53270.25 |
| NULL       |              4 |    68240.32 |
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| BUS        |           NULL |     9345.55 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CD         |           NULL |    19500.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| CHK        |           NULL |    73008.01 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| MM         |           NULL |    17045.14 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SAV        |           NULL |     1855.76 |
| SBL        |              3 |    50000.00 |
| SBL        |           NULL |    50000.00 |
+------------+----------------+-------------+
25 rows in set (0.02 sec)
```

Using `with cube` generates four more rows than the `with rollup` version of the query, one for each of the four branch IDs. Similar to `with rollup`, `null` values are placed in the `product_cd` column to indicate that a branch summary is being performed.

Once again, if you are using Oracle Database, you need to use a slightly different syntax to indicate that you want a cube operation performed. The `group by` clause for the previous query would look as follows when using Oracle:

```
GROUP BY CUBE(product_cd, open_branch_id)
```

Group Filter Conditions

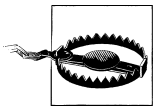
In Chapter 4, I introduced you to various types of filter conditions and showed how you can use them in the `where` clause. When grouping data, you also can apply filter conditions to the data *after* the groups have been generated. The `having` clause is where you should place these types of filter conditions. Consider the following example:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING SUM(avail_balance) >= 10000;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| CD         |     19500.00 |
| CHK        |     73008.01 |
| MM         |     17045.14 |
| SBL        |     50000.00 |
+------------+--------------+
4 rows in set (0.00 sec)
```

This query has two filter conditions: one in the `where` clause, which filters out inactive accounts, and the other in the `having` clause, which filters out any product whose total available balance is less than $10,000. Thus, one of the filters acts on data *before* it is grouped, and the other filter acts on data *after* the groups have been created. If you mistakenly put both filters in the `where` clause, you will see the following error:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    ->   AND SUM(avail_balance) > 10000
    -> GROUP BY product_cd;
ERROR 1111 (HY000): Invalid use of group function
```

This query fails because you cannot include an aggregate function in a query's `where` clause. This is because the filters in the `where` clause are evaluated *before* the grouping occurs, so the server can't yet perform any functions on groups.



When adding filters to a query that includes a `group by` clause, think carefully about whether the filter acts on raw data, in which case it belongs in the `where` clause, or on grouped data, in which case it belongs in the `having` clause.

You may, however, include aggregate functions in the `having` clause, that do *not* appear in the `select` clause, as demonstrated by the following:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING MIN(avail_balance) >= 1000
```

```
    ->   AND MAX(avail_balance) <= 10000;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| CD         |     19500.00 |
| MM         |     17045.14 |
+------------+--------------+
2 rows in set (0.00 sec)
```

This query generates total balances for each active product, but then the filter condition in the `having` clause excludes all products for which the minimum balance is less than $1,000 or the maximum balance is greater than $10,000.

Test Your Knowledge

Work through the following exercises to test your grasp of SQL's grouping and aggregating features. Check your work with the answers in Appendix C.

Exercise 8-1

Construct a query that counts the number of rows in the `account` table.

Exercise 8-2

Modify your query from Exercise 8-1 to count the number of accounts held by each customer. Show the customer ID and the number of accounts for each customer.

Exercise 8-3

Modify your query from Exercise 8-2 to include only those customers having at least two accounts.

Exercise 8-4 (Extra Credit)

Find the total available balance by product and branch where there is more than one account per product and branch. Order the results by total balance (highest to lowest).

CHAPTER 9
Subqueries

Subqueries are a powerful tool that you can use in all four SQL data statements. This chapter explores in great detail the many uses of the subquery.

What Is a Subquery?

A *subquery* is a query contained within another SQL statement (which I refer to as the *containing statement* for the rest of this discussion). A subquery is always enclosed within parentheses, and it is usually executed prior to the containing statement. Like any query, a subquery returns a result set that may consist of:

- A single row with a single column
- Multiple rows with a single column
- Multiple rows and columns

The type of result set the subquery returns determines how it may be used and which operators the containing statement may use to interact with the data the subquery returns. When the containing statement has finished executing, the data returned by any subqueries is discarded, making a subquery act like a temporary table with *statement scope* (meaning that the server frees up any memory allocated to the subquery results after the SQL statement has finished execution).

You already saw several examples of subqueries in earlier chapters, but here's a simple example to get started:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE account_id = (SELECT MAX(account_id) FROM account);
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
1 row in set (0.65 sec)
```

In this example, the subquery returns the maximum value found in the `account_id` column in the `account` table, and the containing statement then returns data about that account. If you are ever confused about what a subquery is doing, you can run the subquery by itself (without the parentheses) to see what it returns. Here's the subquery from the previous example:

```
mysql> SELECT MAX(account_id) FROM account;
+-----------------+
| MAX(account_id) |
+-----------------+
|              29 |
+-----------------+
1 row in set (0.00 sec)
```

So, the subquery returns a single row with a single column, which allows it to be used as one of the expressions in an equality condition (if the subquery returned two or more rows, it could be *compared* to something but could not be *equal* to anything, but more on this later). In this case, you can take the value the subquery returned and substitute it into the righthand expression of the filter condition in the containing query, as in:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE account_id = 29;
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
1 row in set (0.02 sec)
```

The subquery is useful in this case because it allows you to retrieve information about the highest numbered account in a single query, rather than retrieving the maximum `account_id` using one query and then writing a second query to retrieve the desired data from the `account` table. As you will see, subqueries are useful in many other situations as well, and may become one of the most powerful tools in your SQL toolkit.

Subquery Types

Along with the differences noted previously regarding the type of result set a subquery returns (single row/column, single row/multicolumn, or multiple columns), you can use another factor to differentiate subqueries; some subqueries are completely self-contained (called *noncorrelated subqueries*), while others reference columns from the containing statement (called *correlated subqueries*). The next several sections explore these two subquery types and show the different operators that you can employ to interact with them.

Noncorrelated Subqueries

The example from earlier in the chapter is a noncorrelated subquery; it may be executed alone and does not reference anything from the containing statement. Most subqueries that you encounter will be of this type unless you are writing `update` or `delete` statements, which frequently make use of correlated subqueries (more on this later). Along with being noncorrelated, the example from earlier in the chapter also returns a table comprising a single row and column. This type of subquery is known as a *scalar subquery* and can appear on either side of a condition using the usual operators (`=`, `<>`, `<`, `>`, `<=`, `>=`). The next example shows how you can use a scalar subquery in an inequality condition:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE open_emp_id <> (SELECT e.emp_id
    ->   FROM employee e INNER JOIN branch b
    ->     ON e.assigned_branch_id = b.branch_id
    ->   WHERE e.title = 'Head Teller' AND b.city = 'Woburn');
+------------+------------+---------+--------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+--------------+
|          7 | CHK        |       3 |       1057.75 |
|          8 | MM         |       3 |       2212.50 |
|         10 | CHK        |       4 |        534.12 |
|         11 | SAV        |       4 |        767.77 |
|         12 | MM         |       4 |       5487.09 |
|         13 | CHK        |       5 |       2237.97 |
|         14 | CHK        |       6 |        122.37 |
|         15 | CD         |       6 |      10000.00 |
|         18 | CHK        |       8 |       3487.19 |
|         19 | SAV        |       8 |        387.99 |
|         21 | CHK        |       9 |        125.67 |
|         22 | MM         |       9 |       9345.55 |
|         23 | CD         |       9 |       1500.00 |
|         24 | CHK        |      10 |      23575.12 |
|         25 | BUS        |      10 |          0.00 |
|         28 | CHK        |      12 |      38552.05 |
|         29 | SBL        |      13 |      50000.00 |
+------------+------------+---------+--------------+
17 rows in set (0.86 sec)
```

This query returns data concerning all accounts that were *not* opened by the head teller at the Woburn branch (the subquery is written using the assumption that there is only a single head teller at each branch). The subquery in this example is a bit more complex than in the previous example, in that it joins two tables and includes two filter conditions. Subqueries may be as simple or as complex as you need them to be, and they may utilize any and all the available query clauses (`select`, `from`, `where`, `group by`, `having`, and `order by`).

If you use a subquery in an equality condition, but the subquery returns more than one row, you will receive an error. For example, if you modify the previous query such that

the subquery returns *all* tellers at the Woburn branch instead of the single head teller, you will receive the following error:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE open_emp_id <> (SELECT e.emp_id
    ->   FROM employee e INNER JOIN branch b
    ->     ON e.assigned_branch_id = b.branch_id
    ->   WHERE e.title = 'Teller' AND b.city = 'Woburn');
ERROR 1242 (21000): Subquery returns more than 1 row
```

If you run the subquery by itself, you will see the following results:

```
mysql> SELECT e.emp_id
    -> FROM employee e INNER JOIN branch b
    ->   ON e.assigned_branch_id = b.branch_id
    -> WHERE e.title = 'Teller' AND b.city = 'Woburn';
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
2 rows in set (0.02 sec)
```

The containing query fails because an expression (`open_emp_id`) cannot be equated to a set of expressions (`emp_id`s 11 and 12). In other words, a single thing cannot be equated to a set of things. In the next section, you will see how to fix the problem by using a different operator.

Multiple-Row, Single-Column Subqueries

If your subquery returns more than one row, you will not be able to use it on one side of an equality condition, as the previous example demonstrated. However, there are four additional operators that you can use to build conditions with these types of subqueries.

The in and not in operators

While you can't *equate* a single value to a set of values, you can check to see whether a single value can be found *within* a set of values. The next example, while it doesn't use a subquery, demonstrates how to build a condition that uses the `in` operator to search for a value within a set of values:

```
mysql> SELECT branch_id, name, city
    -> FROM branch
    -> WHERE name IN ('Headquarters', 'Quincy Branch');
+-----------+--------------+---------+
| branch_id | name         | city    |
+-----------+--------------+---------+
|         1 | Headquarters | Waltham |
|         3 | Quincy Branch | Quincy  |
```

```
+----------+--------------+---------+
2 rows in set (0.03 sec)
```

The expression on the lefthand side of the condition is the `name` column, while the righthand side of the condition is a set of strings. The `in` operator checks to see whether either of the strings can be found in the `name` column; if so, the condition is met and the row is added to the result set. You could achieve the same results using two equality conditions, as in:

```
mysql> SELECT branch_id, name, city
    -> FROM branch
    -> WHERE name = 'Headquarters' OR name = 'Quincy Branch';
+----------+--------------+---------+
| branch_id | name          | city    |
+----------+--------------+---------+
|         1 | Headquarters  | Waltham |
|         3 | Quincy Branch | Quincy  |
+----------+--------------+---------+
2 rows in set (0.01 sec)
```

While this approach seems reasonable when the set contains only two expressions, it is easy to see why a single condition using the `in` operator would be preferable if the set contained dozens (or hundreds, thousands, etc.) of values.

Although you will occasionally create a set of strings, dates, or numbers to use on one side of a condition, you are more likely to generate the set at query execution via a subquery that returns one or more rows. The following query uses the `in` operator with a subquery on the righthand side of the filter condition to see which employees supervise other employees:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id IN (SELECT superior_emp_id
    ->   FROM employee);
+--------+---------+-----------+--------------------+
| emp_id | fname   | lname     | title              |
+--------+---------+-----------+--------------------+
|      1 | Michael | Smith     | President          |
|      3 | Robert  | Tyler     | Treasurer          |
|      4 | Susan   | Hawthorne | Operations Manager |
|      6 | Helen   | Fleming   | Head Teller        |
|     10 | Paula   | Roberts   | Head Teller        |
|     13 | John    | Blake     | Head Teller        |
|     16 | Theresa | Markham   | Head Teller        |
+--------+---------+-----------+--------------------+
7 rows in set (0.01 sec)
```

The subquery returns the IDs of all employees who supervise other employees, and the containing query retrieves four columns from the `employee` table for these employees. Here are the results of the subquery:

```
mysql> SELECT superior_emp_id
    -> FROM employee;
+-----------------+
```

```
| superior_emp_id |
+-----------------+
|            NULL |
|               1 |
|               1 |
|               3 |
|               4 |
|               4 |
|               4 |
|               4 |
|               4 |
|               6 |
|               6 |
|               6 |
|              10 |
|              10 |
|              13 |
|              13 |
|              16 |
|              16 |
+-----------------+
18 rows in set (0.00 sec)
```

As you can see, some employee IDs are listed more than once, since some employees supervise multiple people. This doesn't adversely affect the results of the containing query, since it doesn't matter whether an employee ID can be found in the result set of the subquery once or more than once. Of course, you could add the `distinct` keyword to the subquery's `select` clause if it bothers you to have duplicates in the table returned by the subquery, but it won't change the containing query's result set.

Along with seeing whether a value exists within a set of values, you can check the converse using the `not in` operator. Here's another version of the previous query using `not in` instead of `in`:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id NOT IN (SELECT superior_emp_id
    ->   FROM employee
    ->   WHERE superior_emp_id IS NOT NULL);
+--------+----------+----------+----------------+
| emp_id | fname    | lname    | title          |
+--------+----------+----------+----------------+
|      2 | Susan    | Barker   | Vice President |
|      5 | John     | Gooding  | Loan Manager   |
|      7 | Chris    | Tucker   | Teller         |
|      8 | Sarah    | Parker   | Teller         |
|      9 | Jane     | Grossman | Teller         |
|     11 | Thomas   | Ziegler  | Teller         |
|     12 | Samantha | Jameson  | Teller         |
|     14 | Cindy    | Mason    | Teller         |
|     15 | Frank    | Portman  | Teller         |
|     17 | Beth     | Fowler   | Teller         |
|     18 | Rick     | Tulman   | Teller         |
```

```
+--------+----------+----------+----------------+
11 rows in set (0.00 sec)
```

This query finds all employees who do *not* supervise other people. For this query, I needed to add a filter condition to the subquery to ensure that `null` values do not appear in the table returned by the subquery; see the next section for an explanation of why this filter is needed in this case.

The all operator

While the `in` operator is used to see whether an expression can be found within a set of expressions, the `all` operator allows you to make comparisons between a single value and every value in a set. To build such a condition, you will need to use one of the comparison operators (`=`, `<>`, `<`, `>`, etc.) in conjunction with the `all` operator. For example, the next query finds all employees whose employee IDs are not equal to any of the supervisor employee IDs:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id <> ALL (SELECT superior_emp_id
    ->   FROM employee
    ->   WHERE superior_emp_id IS NOT NULL);
+--------+----------+----------+----------------+
| emp_id | fname    | lname    | title          |
+--------+----------+----------+----------------+
|      2 | Susan    | Barker   | Vice President |
|      5 | John     | Gooding  | Loan Manager   |
|      7 | Chris    | Tucker   | Teller         |
|      8 | Sarah    | Parker   | Teller         |
|      9 | Jane     | Grossman | Teller         |
|     11 | Thomas   | Ziegler  | Teller         |
|     12 | Samantha | Jameson  | Teller         |
|     14 | Cindy    | Mason    | Teller         |
|     15 | Frank    | Portman  | Teller         |
|     17 | Beth     | Fowler   | Teller         |
|     18 | Rick     | Tulman   | Teller         |
+--------+----------+----------+----------------+
11 rows in set (0.05 sec)
```

Once again, the subquery returns the set of IDs for those employees who supervise other people, and the containing query returns data for each employee whose ID is not equal to all of the IDs returned by the subquery. In other words, the query finds all employees who are not supervisors. If this approach seems a bit clumsy to you, you are in good company; most people would prefer to phrase the query differently and avoid using the `all` operator. For example, this query generates the same results as the last example in the previous section, which used the `not in` operator. It's a matter of preference, but I think that most people would find the version that uses `not in` to be easier to understand.

When using `not in` or `<> all` to compare a value to a set of values, you must be careful to ensure that the set of values does not contain a `null` value, because the server equates the value on the lefthand side of the expression to each member of the set, and any attempt to equate a value to `null` yields `unknown`. Thus, the following query returns an empty set:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id NOT IN (1, 2, NULL);
Empty set (0.00 sec)
```

In some cases, the `all` operator is a bit more natural. The next example uses `all` to find accounts having an available balance smaller than all of Frank Tucker's accounts:

```
mysql> SELECT account_id, cust_id, product_cd, avail_balance
    -> FROM account
    -> WHERE avail_balance < ALL (SELECT a.avail_balance
    ->   FROM account a INNER JOIN individual i
    ->     ON a.cust_id = i.cust_id
    ->   WHERE i.fname = 'Frank' AND i.lname = 'Tucker');
+------------+---------+------------+---------------+
| account_id | cust_id | product_cd | avail_balance |
+------------+---------+------------+---------------+
|          2 |       1 | SAV        |        500.00 |
|          5 |       2 | SAV        |        200.00 |
|         10 |       4 | CHK        |        534.12 |
|         11 |       4 | SAV        |        767.77 |
|         14 |       6 | CHK        |        122.37 |
|         19 |       8 | SAV        |        387.99 |
|         21 |       9 | CHK        |        125.67 |
|         25 |      10 | BUS        |          0.00 |
+------------+---------+------------+---------------+
8 rows in set (0.17 sec)
```

Here's the data returned by the subquery, which consists of the available balance from each of Frank's accounts:

```
mysql> SELECT a.avail_balance
    -> FROM account a INNER JOIN individual i
    ->   ON a.cust_id = i.cust_id
    -> WHERE i.fname = 'Frank' AND i.lname = 'Tucker';
+---------------+
| avail_balance |
+---------------+
|       1057.75 |
|       2212.50 |
+---------------+
2 rows in set (0.01 sec)
```

Frank has two accounts, with the lowest balance being $1,057.75. The containing query finds all accounts having a balance smaller than any of Frank's accounts, so the result set includes all accounts having a balance less than $1,057.75.

The any operator

Like the `all` operator, the `any` operator allows a value to be compared to the members of a set of values; unlike `all`, however, a condition using the `any` operator evaluates to `true` as soon as a single comparison is favorable. This is different from the previous example using the `all` operator, which evaluates to `true` only if comparisons against *all* members of the set are favorable. For example, you might want to find all accounts having an available balance greater than *any* of Frank Tucker's accounts:

```
mysql> SELECT account_id, cust_id, product_cd, avail_balance
    -> FROM account
    -> WHERE avail_balance > ANY (SELECT a.avail_balance
    ->   FROM account a INNER JOIN individual i
    ->     ON a.cust_id = i.cust_id
    ->   WHERE i.fname = 'Frank' AND i.lname = 'Tucker');
+------------+---------+------------+---------------+
| account_id | cust_id | product_cd | avail_balance |
+------------+---------+------------+---------------+
|          3 |       1 | CD         |       3000.00 |
|          4 |       2 | CHK        |       2258.02 |
|          8 |       3 | MM         |       2212.50 |
|         12 |       4 | MM         |       5487.09 |
|         13 |       5 | CHK        |       2237.97 |
|         15 |       6 | CD         |      10000.00 |
|         17 |       7 | CD         |       5000.00 |
|         18 |       8 | CHK        |       3487.19 |
|         22 |       9 | MM         |       9345.55 |
|         23 |       9 | CD         |       1500.00 |
|         24 |      10 | CHK        |      23575.12 |
|         27 |      11 | BUS        |       9345.55 |
|         28 |      12 | CHK        |      38552.05 |
|         29 |      13 | SBL        |      50000.00 |
+------------+---------+------------+---------------+
14 rows in set (0.00 sec)
```

Frank has two accounts with balances of $1,057.75 and $2,212.50; to have a balance greater than *any* of these two accounts, an account must have a balance of at least $1,057.75.

Although most people prefer to use `in`, using `= any` is equivalent to using the `in` operator.

Multicolumn Subqueries

So far, all of the subquery examples in this chapter have returned a single column and one or more rows. In certain situations, however, you can use subqueries that return two or more columns. To show the utility of multiple-column subqueries, it might help to look first at an example that uses multiple, single-column subqueries:

```
mysql> SELECT account_id, product_cd, cust_id
    -> FROM account
    -> WHERE open_branch_id = (SELECT branch_id
    ->   FROM branch
    ->   WHERE name = 'Woburn Branch')
    ->   AND open_emp_id IN (SELECT emp_id
    ->   FROM employee
    ->   WHERE title = 'Teller' OR title = 'Head Teller');
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|         17 | CD         |       7 |
|         27 | BUS        |      11 |
+------------+------------+---------+
7 rows in set (0.09 sec)
```

This query uses two subqueries to identify the ID of the Woburn branch and the IDs of all bank tellers, and the containing query then uses this information to retrieve all checking accounts opened by a teller at the Woburn branch. However, since the `employee` table includes information about which branch each employee is assigned to, you can achieve the same results by comparing both the `account.open_branch_id` and `account.open_emp_id` columns to a single subquery against the `employee` and `branch` tables. To do so, your filter condition must name both columns from the `account` table surrounded by parentheses and in the same order as returned by the subquery, as in:

```
mysql> SELECT account_id, product_cd, cust_id
    -> FROM account
    -> WHERE (open_branch_id, open_emp_id) IN
    ->  (SELECT b.branch_id, e.emp_id
    ->   FROM branch b INNER JOIN employee e
    ->     ON b.branch_id = e.assigned_branch_id
    ->   WHERE b.name = 'Woburn Branch'
    ->     AND (e.title = 'Teller' OR e.title = 'Head Teller'));
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|         17 | CD         |       7 |
|         27 | BUS        |      11 |
+------------+------------+---------+
7 rows in set (0.00 sec)
```

This version of the query performs the same function as the previous example, but with a single subquery that returns two columns instead of two subqueries that each return a single column.

Of course, you could rewrite the previous example simply to join the three tables instead of using a subquery, but it's helpful when learning SQL to see multiple ways of achieving the same results. Here's another example, however, that requires a subquery. Let's say that there have been some customer complaints regarding incorrect values in the available/pending balance columns in the `account` table. Your job is to find all accounts whose balances don't match the sum of the transaction amounts for that account. Here's a partial solution to the problem:

```
SELECT 'ALERT! : Account #1 Has Incorrect Balance!'
FROM account
WHERE (avail_balance, pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
  FROM transaction
  WHERE account_id = 1)
  AND account_id = 1;
```

As you can see, I have neglected to fill in the expressions used to sum the transaction amounts for the available and pending balance calculations, but I promise to finish the job in Chapter 11 after you learn how to build `case` expressions. Even so, the query is complete enough to see that the subquery is generating two sums from the `transaction` table that are then compared to the `avail_balance` and `pending_balance` columns in the `account` table. Both the subquery and the containing query include the filter condition `account_id = 1`, so the query in its present form will check only a single account at a time. In the next section, you will learn how to write a more general form of the query that will check *all* accounts with a single execution.

Correlated Subqueries

All of the subqueries shown thus far have been independent of their containing statements, meaning that you can execute them by themselves and inspect the results. A *correlated subquery*, on the other hand, is *dependent* on its containing statement from which it references one or more columns. Unlike a noncorrelated subquery, a correlated subquery is not executed once prior to execution of the containing statement; instead, the correlated subquery is executed once for each candidate row (rows that might be included in the final results). For example, the following query uses a correlated subquery to count the number of accounts for each customer, and the containing query then retrieves those customers having exactly two accounts:

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer c
    -> WHERE 2 = (SELECT COUNT(*)
    ->   FROM account a
    ->   WHERE a.cust_id = c.cust_id);
+---------+--------------+---------+
| cust_id | cust_type_cd | city    |
+---------+--------------+---------+
|       2 | I            | Woburn  |
|       3 | I            | Quincy  |
```

```
|       6 | I            | Waltham |
|       8 | I            | Salem   |
|      10 | B            | Salem   |
+---------+--------------+---------+
5 rows in set (0.01 sec)
```

The reference to `c.cust_id` at the very end of the subquery is what makes the subquery correlated; the containing query must supply values for `c.cust_id` for the subquery to execute. In this case, the containing query retrieves all 13 rows from the `customer` table and executes the subquery once for each customer, passing in the appropriate customer ID for each execution. If the subquery returns the value `2`, then the filter condition is met and the row is added to the result set.

Along with equality conditions, you can use correlated subqueries in other types of conditions, such as the range condition illustrated here:

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer c
    -> WHERE (SELECT SUM(a.avail_balance)
    ->     FROM account a
    ->     WHERE a.cust_id = c.cust_id)
    ->   BETWEEN 5000 AND 10000;
+---------+--------------+------------+
| cust_id | cust_type_cd | city       |
+---------+--------------+------------+
|       4 | I            | Waltham    |
|       7 | I            | Wilmington |
|      11 | B            | Wilmington |
+---------+--------------+------------+
3 rows in set (0.02 sec)
```

This variation on the previous query finds all customers whose total available balance across all accounts lies between $5,000 and $10,000. Once again, the correlated subquery is executed 13 times (once for each customer row), and each execution of the subquery returns the total account balance for the given customer.

Another subtle difference in the previous query is that the subquery is on the lefthand side of the condition, which may look a bit odd but is perfectly valid.

At the end of the previous section, I demonstrated how to check the available and pending balances of an account against the transactions logged against the account, and I promised to show you how to modify the example to run all accounts in a single execution. Here's the example again:

```
SELECT 'ALERT! : Account #1 Has Incorrect Balance!'
FROM account
WHERE (avail_balance, pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
  FROM transaction
```

```
  WHERE account_id = 1)
  AND account_id = 1;
```

Using a correlated subquery instead of a noncorrelated subquery, you can execute the containing query once, and the subquery will be run for each account. Here's the updated version:

```
SELECT CONCAT('ALERT! : Account #', a.account_id,
  ' Has Incorrect Balance!')
FROM account a
WHERE (a.avail_balance, a.pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

The subquery now includes a filter condition linking the transaction's account ID to the account ID from the containing query. The `select` clause has also been modified to concatenate an alert message that includes the account ID rather than the hardcoded value `1`.

The exists Operator

While you will often see correlated subqueries used in equality and range conditions, the most common operator used to build conditions that utilize correlated subqueries is the `exists` operator. You use the `exists` operator when you want to identify that a relationship exists without regard for the quantity; for example, the following query finds all the accounts for which a transaction was posted on a particular day, without regard for how many transactions were posted:

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
FROM account a
WHERE EXISTS (SELECT 1
  FROM transaction t
  WHERE t.account_id = a.account_id
    AND t.txn_date = '2008-09-22');
```

Using the `exists` operator, your subquery can return zero, one, or many rows, and the condition simply checks whether the subquery returned any rows. If you look at the `select` clause of the subquery, you will see that it consists of a single literal (`1`); since the condition in the containing query only needs to know how many rows have been returned, the actual data the subquery returned is irrelevant. Your subquery can return whatever strikes your fancy, as demonstrated next:

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
FROM account a
WHERE EXISTS (SELECT t.txn_id, 'hello', 3.1415927
  FROM transaction t
  WHERE t.account_id = a.account_id
    AND t.txn_date = '2008-09-22');
```

However, the convention is to specify either `select 1` or `select *` when using `exists`.

You may also use `not exists` to check for subqueries that return no rows, as demonstrated by the following:

```
mysql> SELECT a.account_id, a.product_cd, a.cust_id
    -> FROM account a
    -> WHERE NOT EXISTS (SELECT 1
    ->   FROM business b
    ->   WHERE b.cust_id = a.cust_id);
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|          7 | CHK        |       3 |
|          8 | MM         |       3 |
|         10 | CHK        |       4 |
|         11 | SAV        |       4 |
|         12 | MM         |       4 |
|         13 | CHK        |       5 |
|         14 | CHK        |       6 |
|         15 | CD         |       6 |
|         17 | CD         |       7 |
|         18 | CHK        |       8 |
|         19 | SAV        |       8 |
|         21 | CHK        |       9 |
|         22 | MM         |       9 |
|         23 | CD         |       9 |
+------------+------------+---------+
19 rows in set (0.99 sec)
```

This query finds all customers whose customer ID does not appear in the `business` table, which is a roundabout way of finding all nonbusiness customers.

Data Manipulation Using Correlated Subqueries

All of the examples thus far in the chapter have been `select` statements, but don't think that means that subqueries aren't useful in other SQL statements. Subqueries are used heavily in `update`, `delete`, and `insert` statements as well, with correlated subqueries appearing frequently in `update` and `delete` statements. Here's an example of a correlated subquery used to modify the `last_activity_date` column in the `account` table:

```
UPDATE account a
SET a.last_activity_date =
 (SELECT MAX(t.txn_date)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

This statement modifies every row in the `account` table (since there is no `where` clause) by finding the latest transaction date for each account. While it seems reasonable to expect that every account will have at least one transaction linked to it, it would be best

to check whether an account has any transactions before attempting to update the `last_activity_date` column; otherwise, the column will be set to `null`, since the subquery would return no rows. Here's another version of the `update` statement, this time employing a `where` clause with a second correlated subquery:

```
UPDATE account a
SET a.last_activity_date =
 (SELECT MAX(t.txn_date)
  FROM transaction t
  WHERE t.account_id = a.account_id)
WHERE EXISTS (SELECT 1
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

The two correlated subqueries are identical except for the `select` clauses. The subquery in the `set` clause, however, executes only if the condition in the `update` statement's `where` clause evaluates to `true` (meaning that at least one transaction was found for the account), thus protecting the data in the `last_activity_date` column from being overwritten with a `null`.

Correlated subqueries are also common in `delete` statements. For example, you may run a data maintenance script at the end of each month that removes unnecessary data. The script might include the following statement, which removes data from the `department` table that has no child rows in the `employee` table:

```
DELETE FROM department
WHERE NOT EXISTS (SELECT 1
  FROM employee
  WHERE employee.dept_id = department.dept_id);
```

When using correlated subqueries with `delete` statements in MySQL, keep in mind that, for whatever reason, table aliases are not allowed when using `delete`, which is why I had to use the entire table name in the subquery. With most other database servers, you could provide aliases for the `department` and `employee` tables, such as:

```
DELETE FROM department d
WHERE NOT EXISTS (SELECT 1
  FROM employee e
  WHERE e.dept_id = d.dept_id);
```

When to Use Subqueries

Now that you have learned about the different types of subqueries and the different operators that you can employ to interact with the data returned by subqueries, it's time to explore the many ways in which you can use subqueries to build powerful SQL statements. The next three sections demonstrate how you may use subqueries to construct custom tables, to build conditions, and to generate column values in result sets.

Subqueries As Data Sources

Back in Chapter 3, I stated that the `from` clause of a `select` statement names the *tables* to be used by the query. Since a subquery generates a result set containing rows and columns of data, it is perfectly valid to include subqueries in your `from` clause along with tables. Although it might, at first glance, seem like an interesting feature without much practical merit, using subqueries alongside tables is one of the most powerful tools available when writing queries. Here's a simple example:

```
mysql> SELECT d.dept_id, d.name, e_cnt.how_many num_employees
    -> FROM department d INNER JOIN
    ->  (SELECT dept_id, COUNT(*) how_many
    ->   FROM employee
    ->   GROUP BY dept_id) e_cnt
    ->   ON d.dept_id = e_cnt.dept_id;
+---------+----------------+---------------+
| dept_id | name           | num_employees |
+---------+----------------+---------------+
|       1 | Operations     |            14 |
|       2 | Loans          |             1 |
|       3 | Administration |             3 |
+---------+----------------+---------------+
3 rows in set (0.04 sec)
```

In this example, a subquery generates a list of department IDs along with the number of employees assigned to each department. Here's the result set generated by the subquery:

```
mysql> SELECT dept_id, COUNT(*) how_many
    -> FROM employee
    -> GROUP BY dept_id;
+---------+----------+
| dept_id | how_many |
+---------+----------+
|       1 |       14 |
|       2 |        1 |
|       3 |        3 |
+---------+----------+
3 rows in set (0.00 sec)
```

The subquery is given the name `e_cnt` and is joined to the `department` table via the `dept_id` column. The containing query then retrieves the department ID and name from the `department` table, along with the employee count from the `e_cnt` subquery.

Subqueries used in the `from` clause must be noncorrelated; they are executed first, and the data is held in memory until the containing query finishes execution. Subqueries offer immense flexibility when writing queries, because you can go far beyond the set of available tables to create virtually any view of the data that you desire, and then join the results to other tables or subqueries. If you are writing reports or generating data feeds to external systems, you may be able to do things with a single query that used to demand multiple queries or a procedural language to accomplish.

Data fabrication

Along with using subqueries to summarize existing data, you can use subqueries to generate data that doesn't exist in any form within your database. For example, you may wish to group your customers by the amount of money held in deposit accounts, but you want to use group definitions that are not stored in your database. For example, let's say you want to sort your customers into the groups shown in Table 9-1.

Table 9-1. Customer balance groups

Group name	Lower limit	Upper limit
Small Fry	0	$4,999.99
Average Joes	$5,000	$9,999.99
Heavy Hitters	$10,000	$9,999,999.99

To generate these groups within a single query, you will need a way to define these three groups. The first step is to define a query that generates the group definitions:

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    -> UNION ALL
    -> SELECT 'Average Joes' name, 5000 low_limit, 9999.99 high_limit
    -> UNION ALL
    -> SELECT 'Heavy Hitters' name, 10000 low_limit, 9999999.99 high_limit;
+---------------+-----------+------------+
| name          | low_limit | high_limit |
+---------------+-----------+------------+
| Small Fry     |         0 |    4999.99 |
| Average Joes  |      5000 |    9999.99 |
| Heavy Hitters |     10000 | 9999999.99 |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

I have used the set operator `union all` to merge the results from three separate queries into a single result set. Each query retrieves three literals, and the results from the three queries are put together to generate a result set with three rows and three columns. You now have a query to generate the desired groups, and you can place it into the `from` clause of another query to generate your customer groups:

```
mysql> SELECT groups.name, COUNT(*) num_customers
    -> FROM
    ->  (SELECT SUM(a.avail_balance) cust_balance
    ->   FROM account a INNER JOIN product p
    ->     ON a.product_cd = p.product_cd
    ->   WHERE p.product_type_cd = 'ACCOUNT'
    ->   GROUP BY a.cust_id) cust_rollup
    ->   INNER JOIN
    ->  (SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    ->   UNION ALL
    ->   SELECT 'Average Joes' name, 5000 low_limit,
    ->     9999.99 high_limit
    ->   UNION ALL
    ->   SELECT 'Heavy Hitters' name, 10000 low_limit,
```

```
    ->     9999999.99 high_limit) groups
    ->   ON cust_rollup.cust_balance
    ->     BETWEEN groups.low_limit AND groups.high_limit
    -> GROUP BY groups.name;
+---------------+---------------+
| name          | num_customers |
+---------------+---------------+
| Average Joes  |             2 |
| Heavy Hitters |             4 |
| Small Fry     |             5 |
+---------------+---------------+
3 rows in set (0.01 sec)
```

The `from` clause contains two subqueries; the first subquery, named `cust_rollup`, returns the total deposit balances for each customer, while the second subquery, named `groups`, generates the three customer groupings. Here's the data generated by `cust_rollup`:

```
mysql> SELECT SUM(a.avail_balance) cust_balance
    -> FROM account a INNER JOIN product p
    ->   ON a.product_cd = p.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT'
    -> GROUP BY a.cust_id;
+--------------+
| cust_balance |
+--------------+
|      4557.75 |
|      2458.02 |
|      3270.25 |
|      6788.98 |
|      2237.97 |
|     10122.37 |
|      5000.00 |
|      3875.18 |
|     10971.22 |
|     23575.12 |
|     38552.05 |
+--------------+
11 rows in set (0.05 sec)
```

The data generated by `cust_rollup` is then joined to the `groups` table via a range condition (`cust_rollup.cust_balance BETWEEN groups.low_limit AND groups.high_limit`). Finally, the joined data is grouped and the number of customers in each group is counted to generate the final result set.

Of course, you could simply decide to build a permanent table to hold the group definitions instead of using a subquery. Using that approach, you would find your database to be littered with small special-purpose tables after awhile, and you wouldn't remember the reason for which most of them were created. I've worked in environments where the database users were allowed to create their own tables for special purposes, and the results were disastrous (tables not included in backups, tables lost during server upgrades, server downtime due to space allocation issues, etc.). Armed with subqueries,

however, you will be able to adhere to a policy where tables are added to a database only when there is a clear business need to store new data.

Task-oriented subqueries

In systems used for reporting or data-feed generation, you will often come across queries such as the following:

```
mysql> SELECT p.name product, b.name branch,
    ->   CONCAT(e.fname, ' ', e.lname) name,
    ->   SUM(a.avail_balance) tot_deposits
    -> FROM account a INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    ->   INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    ->   INNER JOIN product p
    ->   ON a.product_cd = p.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT'
    -> GROUP BY p.name, b.name, e.fname, e.lname
    -> ORDER BY 1,2;
+-----------------------+--------------+-----------------+--------------+
| product               | branch       | name            | tot_deposits |
+-----------------------+--------------+-----------------+--------------+
| certificate of deposit | Headquarters | Michael Smith   |     11500.00 |
| certificate of deposit | Woburn Branch | Paula Roberts  |      8000.00 |
| checking account      | Headquarters | Michael Smith   |       782.16 |
| checking account      | Quincy Branch | John Blake     |      1057.75 |
| checking account      | So. NH Branch | Theresa Markham |     67852.33 |
| checking account      | Woburn Branch | Paula Roberts  |      3315.77 |
| money market account  | Headquarters | Michael Smith   |     14832.64 |
| money market account  | Quincy Branch | John Blake     |      2212.50 |
| savings account       | Headquarters | Michael Smith   |       767.77 |
| savings account       | So. NH Branch | Theresa Markham |       387.99 |
| savings account       | Woburn Branch | Paula Roberts  |       700.00 |
+-----------------------+--------------+-----------------+--------------+
11 rows in set (0.00 sec)
```

This query sums all deposit account balances by account type, the employee that opened the accounts, and the branches at which the accounts were opened. If you look at the query closely, you will see that the `product`, `branch`, and `employee` tables are needed only for display purposes, and that the `account` table has everything needed to generate the groupings (`product_cd`, `open_branch_id`, `open_emp_id`, and `avail_balance`). Therefore, you could separate out the task of generating the groups into a subquery, and then join the other three tables to the table generated by the subquery to achieve the desired end result. Here's the grouping subquery:

```
mysql> SELECT product_cd, open_branch_id branch_id, open_emp_id emp_id,
    ->   SUM(avail_balance) tot_deposits
    -> FROM account
    -> GROUP BY product_cd, open_branch_id, open_emp_id;
+------------+-----------+--------+--------------+
| product_cd | branch_id | emp_id | tot_deposits |
+------------+-----------+--------+--------------+
```

```
| BUS        |          2 |     10 |      9345.55 |
| BUS        |          4 |     16 |         0.00 |
| CD         |          1 |      1 |     11500.00 |
| CD         |          2 |     10 |      8000.00 |
| CHK        |          1 |      1 |       782.16 |
| CHK        |          2 |     10 |      3315.77 |
| CHK        |          3 |     13 |      1057.75 |
| CHK        |          4 |     16 |     67852.33 |
| MM         |          1 |      1 |     14832.64 |
| MM         |          3 |     13 |      2212.50 |
| SAV        |          1 |      1 |       767.77 |
| SAV        |          2 |     10 |       700.00 |
| SAV        |          4 |     16 |       387.99 |
| SBL        |          3 |     13 |     50000.00 |
+------------+------------+--------+--------------+
14 rows in set (0.02 sec)
```

This is the heart of the query; the other tables are needed only to provide meaningful strings in place of the `product_cd`, `open_branch_id`, and `open_emp_id` foreign key columns. The next query wraps the query against the `account` table in a subquery and joins the table that results to the other three tables:

```
mysql> SELECT p.name product, b.name branch,
    ->   CONCAT(e.fname, ' ', e.lname) name,
    ->   account_groups.tot_deposits
    -> FROM
    ->  (SELECT product_cd, open_branch_id branch_id,
    ->     open_emp_id emp_id,
    ->     SUM(avail_balance) tot_deposits
    ->   FROM account
    ->   GROUP BY product_cd, open_branch_id, open_emp_id) account_groups
    ->   INNER JOIN employee e ON e.emp_id = account_groups.emp_id
    ->   INNER JOIN branch b ON b.branch_id = account_groups.branch_id
    ->   INNER JOIN product p ON p.product_cd = account_groups.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT';
+-----------------------+--------------+-----------------+--------------+
| product               | branch       | name            | tot_deposits |
+-----------------------+--------------+-----------------+--------------+
| certificate of deposit | Headquarters | Michael Smith   |     11500.00 |
| certificate of deposit | Woburn Branch | Paula Roberts  |      8000.00 |
| checking account      | Headquarters | Michael Smith   |       782.16 |
| checking account      | Quincy Branch | John Blake     |      1057.75 |
| checking account      | So. NH Branch | Theresa Markham |     67852.33 |
| checking account      | Woburn Branch | Paula Roberts  |      3315.77 |
| money market account  | Headquarters | Michael Smith   |     14832.64 |
| money market account  | Quincy Branch | John Blake     |      2212.50 |
| savings account       | Headquarters | Michael Smith   |       767.77 |
| savings account       | So. NH Branch | Theresa Markham |       387.99 |
| savings account       | Woburn Branch | Paula Roberts  |       700.00 |
+-----------------------+--------------+-----------------+--------------+
11 rows in set (0.01 sec)
```

I realize that beauty is in the eye of the beholder, but I find this version of the query to be far more satisfying than the big, flat version. This version may execute faster, as well, because the grouping is being done on small, numeric foreign key columns (`product_cd`,

open_branch_id, open_emp_id) instead of potentially lengthy string columns (branch.name, product.name, employee.fname, employee.lname).

Subqueries in Filter Conditions

Many of the examples in this chapter used subqueries as expressions in filter conditions, so it should not surprise you that this is one of the main uses for subqueries. However, filter conditions using subqueries are not found only in the where clause. For example, the next query uses a subquery in the having clause to find the employee responsible for opening the most accounts:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> HAVING COUNT(*) = (SELECT MAX(emp_cnt.how_many)
    ->   FROM (SELECT COUNT(*) how_many
    ->     FROM account
    ->     GROUP BY open_emp_id) emp_cnt);
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
+-------------+----------+
1 row in set (0.01 sec)
```

The subquery in the having clause finds the maximum number of accounts opened by any employee, and the containing query finds the employee that has opened that number of accounts. If multiple employees tie for the highest number of opened accounts, then the query would return multiple rows.

Subqueries As Expression Generators

For this last section of the chapter, I finish where I began: with single-column, single-row scalar subqueries. Along with being used in filter conditions, scalar subqueries may be used wherever an expression can appear, including the select and order by clauses of a query and the values clause of an insert statement.

In "Task-oriented subqueries" on page 175, I showed you how to use a subquery to separate out the grouping mechanism from the rest of the query. Here's another version of the same query that uses subqueries for the same purpose, but in a different way:

```
mysql> SELECT
    ->  (SELECT p.name FROM product p
    ->   WHERE p.product_cd = a.product_cd
    ->     AND p.product_type_cd = 'ACCOUNT') product,
    ->  (SELECT b.name FROM branch b
    ->   WHERE b.branch_id = a.open_branch_id) branch,
    ->  (SELECT CONCAT(e.fname, ' ', e.lname) FROM employee e
    ->   WHERE e.emp_id = a.open_emp_id) name,
    ->   SUM(a.avail_balance) tot_deposits
    -> FROM account a
```

```
    -> GROUP BY a.product_cd, a.open_branch_id, a.open_emp_id
    -> ORDER BY 1,2;
+-----------------------+---------------+-----------------+--------------+
| product               | branch        | name            | tot_deposits |
+-----------------------+---------------+-----------------+--------------+
| NULL                  | Quincy Branch | John Blake      |     50000.00 |
| NULL                  | So. NH Branch | Theresa Markham |         0.00 |
| NULL                  | Woburn Branch | Paula Roberts   |      9345.55 |
| certificate of deposit | Headquarters  | Michael Smith   |     11500.00 |
| certificate of deposit | Woburn Branch | Paula Roberts   |      8000.00 |
| checking account      | Headquarters  | Michael Smith   |       782.16 |
| checking account      | Quincy Branch | John Blake      |      1057.75 |
| checking account      | So. NH Branch | Theresa Markham |     67852.33 |
| checking account      | Woburn Branch | Paula Roberts   |      3315.77 |
| money market account  | Headquarters  | Michael Smith   |     14832.64 |
| money market account  | Quincy Branch | John Blake      |      2212.50 |
| savings account       | Headquarters  | Michael Smith   |       767.77 |
| savings account       | So. NH Branch | Theresa Markham |       387.99 |
| savings account       | Woburn Branch | Paula Roberts   |       700.00 |
+-----------------------+---------------+-----------------+--------------+
14 rows in set (0.01 sec)
```

There are two main differences between this query and the earlier version using a subquery in the `from` clause:

- Instead of joining the `product`, `branch`, and `employee` tables to the account data, correlated scalar subqueries are used in the `select` clause to look up the product, branch, and employee names.
- The result set has 14 rows instead of 11 rows, and three of the product names are `null`.

The reason for the extra three rows in the result set is that the previous version of the query included the filter condition `p.product_type_cd = 'ACCOUNT'`. That filter eliminated rows with product types of `INSURANCE` and `LOAN`, such as small business loans. Since this version of the query doesn't include a join to the `product` table, there is no way to include the filter condition in the main query. The correlated subquery against the `product` table does include this filter, but the only effect is to leave the product name `null`. If you want to get rid of the extra three rows, you could join the `product` table to the `account` table and include the filter condition, or you could simply do the following:

```
mysql> SELECT all_prods.product, all_prods.branch,
    ->   all_prods.name, all_prods.tot_deposits
    -> FROM
    ->  (SELECT
    ->    (SELECT p.name FROM product p
    ->     WHERE p.product_cd = a.product_cd
    ->       AND p.product_type_cd = 'ACCOUNT') product,
    ->    (SELECT b.name FROM branch b
    ->     WHERE b.branch_id = a.open_branch_id) branch,
    ->    (SELECT CONCAT(e.fname, ' ', e.lname) FROM employee e
    ->     WHERE e.emp_id = a.open_emp_id) name,
    ->     SUM(a.avail_balance) tot_deposits
    ->   FROM account a
```

```
    ->   GROUP BY a.product_cd, a.open_branch_id, a.open_emp_id
    ->  ) all_prods
    -> WHERE all_prods.product IS NOT NULL
    -> ORDER BY 1,2;
+-----------------------+--------------+-----------------+--------------+
| product               | branch       | name            | tot_deposits |
+-----------------------+--------------+-----------------+--------------+
| certificate of deposit | Headquarters  | Michael Smith   |     11500.00 |
| certificate of deposit | Woburn Branch | Paula Roberts   |      8000.00 |
| checking account       | Headquarters  | Michael Smith   |       782.16 |
| checking account       | Quincy Branch | John Blake      |      1057.75 |
| checking account       | So. NH Branch | Theresa Markham |     67852.33 |
| checking account       | Woburn Branch | Paula Roberts   |      3315.77 |
| money market account   | Headquarters  | Michael Smith   |     14832.64 |
| money market account   | Quincy Branch | John Blake      |      2212.50 |
| savings account        | Headquarters  | Michael Smith   |       767.77 |
| savings account        | So. NH Branch | Theresa Markham |       387.99 |
| savings account        | Woburn Branch | Paula Roberts   |       700.00 |
+-----------------------+--------------+-----------------+--------------+
11 rows in set (0.01 sec)
```

Simply by wrapping the previous query in a subquery (called `all_prods`) and adding a filter condition to exclude `null` values of the `product` column, the query now returns the desired 11 rows. The end result is a query that performs all grouping against raw data in the `account` table, and then embellishes the output using data in three other tables, and *without doing any joins*.

As previously noted, scalar subqueries can also appear in the `order by` clause. The following query retrieves employee data sorted by the last name of each employee's boss, and then by the employee's last name:

```
mysql> SELECT emp.emp_id, CONCAT(emp.fname, ' ', emp.lname) emp_name,
    ->  (SELECT CONCAT(boss.fname, ' ', boss.lname)
    ->   FROM employee boss
    ->   WHERE boss.emp_id = emp.superior_emp_id) boss_name
    -> FROM employee emp
    -> WHERE emp.superior_emp_id IS NOT NULL
    -> ORDER BY (SELECT boss.lname FROM employee boss
    ->   WHERE boss.emp_id = emp.superior_emp_id), emp.lname;
+--------+-----------------+-----------------+
| emp_id | emp_name        | boss_name       |
+--------+-----------------+-----------------+
|     14 | Cindy Mason     | John Blake      |
|     15 | Frank Portman   | John Blake      |
|      9 | Jane Grossman   | Helen Fleming   |
|      8 | Sarah Parker    | Helen Fleming   |
|      7 | Chris Tucker    | Helen Fleming   |
|     13 | John Blake      | Susan Hawthorne |
|      6 | Helen Fleming   | Susan Hawthorne |
|      5 | John Gooding    | Susan Hawthorne |
|     16 | Theresa Markham | Susan Hawthorne |
|     10 | Paula Roberts   | Susan Hawthorne |
|     17 | Beth Fowler     | Theresa Markham |
|     18 | Rick Tulman     | Theresa Markham |
```

```
|     12 | Samantha Jameson | Paula Roberts   |
|     11 | Thomas Ziegler   | Paula Roberts   |
|      2 | Susan Barker     | Michael Smith   |
|      3 | Robert Tyler     | Michael Smith   |
|      4 | Susan Hawthorne  | Robert Tyler    |
+--------+------------------+-----------------+
17 rows in set (0.01 sec)
```

The query uses two correlated scalar subqueries: one in the `select` clause to retrieve the full name of each employee's boss, and another in the `order by` clause to return just the last name of each employee's boss for sorting purposes.

Along with using correlated scalar subqueries in `select` statements, you can use non-correlated scalar subqueries to generate values for an `insert` statement. For example, let's say you are going to generate a new account row, and you've been given the following data:

- The product name ("savings account")
- The customer's federal ID ("555-55-5555")
- The name of the branch where the account was opened ("Quincy Branch")
- The first and last names of the teller who opened the account ("Frank Portman")

Before you can create a row in the `account` table, you will need to look up the key values for all of these pieces of data so that you can populate the foreign key columns in the `account` table. You have two choices for how to go about it: execute four queries to retrieve the primary key values and place those values into an `insert` statement, or use subqueries to retrieve the four key values from within an `insert` statement. Here's an example of the latter approach:

```
INSERT INTO account
 (account_id, product_cd, cust_id, open_date, last_activity_date,
  status, open_branch_id, open_emp_id, avail_balance, pending_balance)
VALUES (NULL,
 (SELECT product_cd FROM product WHERE name = 'savings account'),
 (SELECT cust_id FROM customer WHERE fed_id = '555-55-5555'),
  '2008-09-25', '2008-09-25', 'ACTIVE',
 (SELECT branch_id FROM branch WHERE name = 'Quincy Branch'),
 (SELECT emp_id FROM employee WHERE lname = 'Portman' AND fname = 'Frank'),
  0, 0);
```

Using a single SQL statement, you can create a row in the `account` table and look up four foreign key column values at the same time. There is one downside to this approach, however. When you use subqueries to generate data for columns that allow `null` values, your `insert` statement will succeed even if one of your subqueries fails to return a value. For example, if you mistyped Frank Portman's name in the fourth subquery, a row will still be created in `account`, but the `open_emp_id` would be set to `null`.

Subquery Wrap-up

I covered a lot of ground in this chapter, so it might be a good idea to review it. The examples I used in this chapter demonstrated subqueries that:

- Return a single column and row, a single column with multiple rows, and multiple columns and rows
- Are independent of the containing statement (noncorrelated subqueries)
- Reference one or more columns from the containing statement (correlated subqueries)
- Are used in conditions that utilize comparison operators as well as the special-purpose operators `in`, `not in`, `exists`, and `not exists`
- Can be found in `select`, `update`, `delete`, and `insert` statements
- Generate result sets that can be joined to other tables (or subqueries) in a query
- Can be used to generate values to populate a table or to populate columns in a query's result set
- Are used in the `select`, `from`, `where`, `having`, and `order by` clauses of queries

Obviously, subqueries are a very versatile tool, so don't feel bad if all these concepts haven't sunk in after reading this chapter for the first time. Keep experimenting with the various uses for subqueries, and you will soon find yourself thinking about how you might utilize a subquery every time you write a nontrivial SQL statement.

Test Your Knowledge

These exercises are designed to test your understanding of subqueries. Please see Appendix C for the solutions.

Exercise 9-1

Construct a query against the `account` table that uses a filter condition with a noncorrelated subquery against the `product` table to find all loan accounts (`product.product_type_cd = 'LOAN'`). Retrieve the account ID, product code, customer ID, and available balance.

Exercise 9-2

Rework the query from Exercise 9-1 using a *correlated* subquery against the `product` table to achieve the same results.

Exercise 9-3

Join the following query to the `employee` table to show the experience level of each employee:

```
SELECT 'trainee' name, '2004-01-01' start_dt, '2005-12-31' end_dt
UNION ALL
SELECT 'worker' name, '2002-01-01' start_dt, '2003-12-31' end_dt
UNION ALL
SELECT 'mentor' name, '2000-01-01' start_dt, '2001-12-31' end_dt
```

Give the subquery the alias `levels`, and include the employee ID, first name, last name, and experience level (`levels.name`). (Hint: build a join condition using an inequality condition to determine into which level the `employee.start_date` column falls.)

Exercise 9-4

Construct a query against the `employee` table that retrieves the employee ID, first name, and last name, along with the names of the department and branch to which the employee is assigned. Do not join any tables.

CHAPTER 10
Joins Revisited

By now, you should be comfortable with the concept of the inner join, which I introduced in Chapter 5. This chapter focuses on other ways in which you can join tables, including the outer join and the cross join.

Outer Joins

In all the examples thus far that have included multiple tables, we haven't been concerned that the join conditions might fail to find matches for all the rows in the tables. For example, when joining the `account` table to the `customer` table, I did not mention the possibility that a value in the `cust_id` column of the `account` table might not match a value in the `cust_id` column of the `customer` table. If that were the case, then some of the rows in one table or the other would be left out of the result set.

Just to be sure, let's check the data in the tables. Here are the `account_id` and `cust_id` columns from the `account` table:

```
mysql> SELECT account_id, cust_id
    -> FROM account;
+------------+---------+
| account_id | cust_id |
+------------+---------+
|          1 |       1 |
|          2 |       1 |
|          3 |       1 |
|          4 |       2 |
|          5 |       2 |
|          7 |       3 |
|          8 |       3 |
|         10 |       4 |
|         11 |       4 |
|         12 |       4 |
|         13 |       5 |
|         14 |       6 |
|         15 |       6 |
|         17 |       7 |
|         18 |       8 |
```

```
|         19 |       8 |
|         21 |       9 |
|         22 |       9 |
|         23 |       9 |
|         24 |      10 |
|         25 |      10 |
|         27 |      11 |
|         28 |      12 |
|         29 |      13 |
+------------+---------+
24 rows in set (1.50 sec)
```

There are 24 accounts spanning 13 different customers, with customer IDs 1 through 13 having at least one account. Here's the set of customer IDs from the `customer` table:

```
mysql> SELECT cust_id
    -> FROM customer;

+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       5 |
|       6 |
|       7 |
|       8 |
|       9 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
13 rows in set (0.02 sec)
```

There are 13 rows in the `customer` table with IDs 1 through 13, so every customer ID is included at least once in the `account` table. When the two tables are joined on the `cust_id` column, therefore, you would expect all 24 rows to be included in the result set (barring any other filter conditions):

```
mysql> SELECT a.account_id, c.cust_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id;
+------------+---------+
| account_id | cust_id |
+------------+---------+
|          1 |       1 |
|          2 |       1 |
|          3 |       1 |
|          4 |       2 |
|          5 |       2 |
|          7 |       3 |
|          8 |       3 |
```