

SIMPLY SQL

BY RUDY LIMEBACK



THE FUN AND EASY WAY TO LEARN BEST-PRACTICE SQL

Simply SQL

(Chapters 1, 2, and 3)

Thank-you for downloading the sample chapters of *Simply SQL*.

This excerpt includes the Summary of Contents, Information about the Author and SitePoint, Table of Contents, Preface, three chapters of the book, and the Index.

We hope you find this information useful in evaluating the book.

[For more information or to order, visit sitepoint.com](http://sitepoint.com)

Summary of Contents of this Excerpt

Preface.....xvii

1. An Introduction to SQL.....1

2. An Overview of the SELECT Statement.....23

3. The FROM Clause.....35

Index.....287

Summary of Additional Book Contents

4. The WHERE Clause.....73

5. The GROUP BY Clause.....103

6. The HAVING Clause.....125

7. The SELECT Clause.....133

8. The ORDER BY Clause.....161

9. SQL Data Types.....183

10. Relational Integrity.....211

11. Special Structures.....237

A. Testing Environment.....253

B. Sample Applications.....259

C. Sample Scripts.....265

D. SQL Keywords.....285



SIMPLY SQL

BY RUDY LIMEBACK

Simply SQL

by Rudy Limeback

Copyright © 2008 SitePoint Pty. Ltd.

Expert Reviewer: Joe Celko

Technical Editor: Andrew Tetlaw

Technical Editor: Dan Maharry

Managing Editor: Chris Wyness

Technical Director: Kevin Yank

Printing History:

First Edition: December 2008

Editor: Kelly Steele

Index Editor: Russell Brooks

Cover Design: Alex Walker

Notice of Rights

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

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors, will be held liable for any damages caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

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

ISBN 978-0-9804552-5-0

Printed and bound in the United States of America

About the Author

Rudy Limeback is a semi-retired SQL Consultant living in Toronto, Canada. His broad SQL experience spans more than 20 years, and includes working with DB2, SQL Server, Access, Oracle, and MySQL. He's an avid participant in discussion forums, primarily at SitePoint (<http://sitepoint.com/forums/>). His two web sites are <http://r937.com/> for SQL and <http://rudy.ca/> for personal stuff. When not glued to his computer, Rudy enjoys playing touch football and frisbee golf.

About the Expert Reviewer

Joe Celko served ten years on ANSI/ISO SQL Standards Committee and contributed to the SQL-89 and SQL-92 standards. He is the author of seven books on SQL for Morgan Kaufmann, and has written over 800 columns in the computer trade and academic press, mostly dealing with data and databases.

About the Technical Editors

Andrew Tetlaw has been tinkering with web sites as a web developer since 1997. Before that, he worked as a high school English teacher, an English teacher in Japan, a window cleaner, a car washer, a kitchen hand, and a furniture salesman. Andrew is dedicated to making the world a better place through the technical editing of SitePoint books and kits. He's also a busy father of five, enjoys coffee, and often neglects his blog at <http://tetlaw.id.au/>.

Dan Maharry is a senior developer for Co-operative Web, a software development workers co-op based in the UK. He specializes in working with new technologies and has been working with .NET since its first beta. Dan lives with his lovely wife Jane and a rose bush that's trying to engulf his house.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our books, newsletters, articles, and community forums.

*To my children, Dieter, Damon,
Anna, and Eric, my
granddaughters Claire and Norah,
and to my Mom.*

Table of Contents

Preface	xvii
Who Should Read This Book?	xvii
The Challenges to Learning SQL	xviii
What's in This Book?	xix
How to Gain Help	xxi
The SitePoint Forums	xxii
The Book's Web Site	xxii
The SitePoint Newsletters	xxiii
Your Feedback	xxiii
Conventions Used in This Book	xxiii
Code Samples	xxiii
Tips, Notes, and Warnings	xxiv
Acknowledgments	xxv
 Chapter 1 An Introduction to SQL	1
SQL Statement Overview	1
Keywords, Identifiers, and Constants	2
Clauses	4
Syntax	5
Data Definition Language	6
CREATE, ALTER, and DROP	7
Starting Over	10
Data Manipulation Language	11
INSERT, UPDATE, and DELETE	12
The SELECT Statement	18
Standard SQL	20
Read The Fine Manual	21

Wrapping Up: an Introduction to SQL	22
---	----

Chapter 2 **An Overview of the SELECT**

Statement

The SELECT Statement	23
The SELECT and FROM Clauses	24
Content Management System	25
The WHERE Clause	28
The GROUP BY and HAVING Clauses	29
The ORDER BY Clause	32
Wrapping Up: the SELECT Statement	33

Chapter 3 **The FROM Clause**

Why Start with the FROM Clause?	36
Parsing an SQL Statement	36
FROM One Table	37
FROM More than One Table Using JOINS	37
Types of Join	38
The Inner Join	39
Outer Joins	40
The Cross Join	44
Real World Joins	46
Inner Join: Categories and Entries	48
Left Outer Join: Categories and Entries	56
Right Outer Join: Categories and Entries	61
Full Outer Join: Categories and Entries	63
Views	67
Views in Web Development	69
Subqueries and Derived Tables	70

Wrapping Up: the FROM Clause	71
Chapter 4 The WHERE Clause	73
Conditions	74
Conditions that are True	74
When "Not True" is Preferable	75
Shopping Carts	76
Conditions that Evaluate as UNKNOWN	79
Operators	79
Comparison Operators	79
The LIKE Operator	82
The BETWEEN Operator	83
Compound Conditions with AND and OR	86
Truth Tables	87
Combining AND and OR	89
IN Conditions	91
IN with Subqueries	92
Correlated Subqueries	93
EXISTS Conditions	96
NOT IN or NOT EXISTS?	98
WHERE Clause Performance	99
Indexes	100
Wrapping Up: the WHERE Clause	102
 Chapter 5 The GROUP BY Clause	103
Grouping is More than Sequencing	104
Out of Many, One	109
Drill-down SQL	113
GROUP BY in Context	114
How GROUP BY Works	115

Group Rows	115
Rules for GROUP BY	117
Columns with Certain Large Data Types	117
Wrapping Up: the GROUP BY	122
Chapter 6 The HAVING Clause	125
HAVING Filters Group Rows	125
HAVING without a GROUP BY Clause	128
Wrapping Up: the HAVING Clause	132
Chapter 7 The SELECT Clause	133
SELECT in the Sequence of Execution	134
Which Columns Can Be Selected?	135
Detail Rows	135
Group Rows	136
The Discussion Forum Application	138
The forums Table	139
The members Table	139
The threads Table	139
The posts Table	140
Functions	140
Aggregate Functions	141
Scalar Functions	149
Operators	154
Numeric Operators	154
The Concatenation Operator	154
Temporal Operators	155
The Dreaded, Evil Select Star	156
SELECT DISTINCT	157
Wrapping Up: the SELECT Clause	159

Chapter 8	The ORDER BY Clause	161
	ORDER BY Syntax	162
	How ORDER BY Works	162
	ASC and DESC	164
	ORDER BY Clause Performance	167
	The Sequence of Values	168
	The Scope of ORDER BY	173
	Using ORDER BY with GROUP BY	175
	ORDER BY Expressions	176
	Special Sequencing	176
	ORDER BY with UNION Queries	178
	Wrapping Up: the ORDER BY Clause	182
 Chapter 9	 SQL Data Types	 183
	An Overview of Data Types	184
	Numeric Data Types	185
	Integers	185
	Decimals	187
	Floating-point Numbers	191
	Conversions in Numeric Calculations	193
	Numeric Functions	193
	Character Data Types	194
	CHAR	194
	VARCHAR	195
	Numeric or Character?	196
	NCHAR and NVARCHAR	198
	CLOB and BLOB	198
	String Functions	199
	Temporal Data Types	200
	DATE	200

TIME	203
TIMESTAMP	205
Intervals	206
Date Functions	207
Column Constraints	208
NULL or NOT NULL	208
DEFAULT	209
CHECK Constraints	209
Wrapping Up: SQL Data Types	210
 Chapter 10 Relational Integrity	211
Identity	212
Data Modelling	212
Entities and Attributes	213
Entities and Relationships	214
Primary Keys	219
UNIQUE Constraints	221
Foreign Keys	224
How Foreign Keys Work	225
Using Foreign Keys	226
Natural versus Surrogate Keys	232
Autonumbers	234
Wrapping Up: Relational Integrity	235
 Chapter 11 Special Structures	237
Joining to a Table Twice	237
Joining a Table to Itself	240
Implementing a Many-to-many Relationship: Keywords	246
Wrapping Up: Special Structures	251

Appendix A Testing Environment	253
Download Your Database System Software	253
Bookmark or Download the SQL Reference	254
Connect to the Database System	255
Command Line	255
Front-end Applications	255
SQL Script Library	256
Performance Problems and Obtaining Help	257
Obtaining the Execution Plan	257
Seeking Help	258
Indexing	258
 Appendix B Sample Applications	 259
Data Model Diagrams	259
Teams and Games	260
Content Management System	261
Discussion Forums	262
Shopping Carts	263
 Appendix C Sample Scripts	 265
Teams and Games	266
The teams Table	266
The games Table	267
Content Management System	268
The entries Table	268
The categories Table	270
The entries_with_category View	272
The contents Table	272
The comments Table	273

The entrykeywords Table	274
Discussion Forums	275
The forums Table	275
The members Table	275
The threads Table	276
The posts Table	276
Shopping Carts	279
The items Table	279
The customers Table	281
The carts Table	282
The cartitems Table	283
The vendors Table	284
Appendix D SQL Keywords	285
Index	287

Preface

This book is about SQL, the Structured Query Language.

SQL is the language used by all major database systems today. SQL has been around for about 30 years, but is enjoying a real renaissance in the 21st century, thanks to the tremendous success of database-driven web sites.

Whether your web site is written in PHP, ASP, Perl, ColdFusion, or any other programming language, and no matter which database system you want to use—MySQL, PostgreSQL, SQL Server, DB2, Oracle, or any of the others—one fact is almost certain: if you want to have database-driven content, you'll need to use SQL.

SQL is a simple, high-level language with tremendous power. You can perform tasks with a few lines of SQL that would take pages and pages of intricate coding to accomplish in a programming language.

Who Should Read This Book?

If you're a web designer or developer looking for guidance in learning SQL for your web projects, this book is for you.

In the early days of the Web, everyone was a web developer. Nowadays, the field has matured to the point where many different disciplines exist. Two broad categories emerged:

- Web designers are responsible for what web site visitors see. This includes the design, graphics, and layout of the site. It also includes designing the functionality of the site, how it *works*, with considerations for the usability of site features.
- Web developers are responsible for the code behind the site. This includes the HTML, CSS, and JavaScript that make the site functional. In addition, web developers handle scripting languages such as PHP, which are used to automate the production of HTML and other code. Scripting languages enable dynamic web site interaction, and are used to communicate with the database.

If you're a web designer, you can benefit from learning SQL—at least at a rudimentary level—because it will help you design better user interactions. Understanding how SQL works means that you can make life simpler for the developers who will im-

plement your designs: by ensuring that the web site is organized in a way that not only serves the web site visitor, but also allows for simple SQL and good database design. We'll cover both SQL and database design in this book.

Web developers are the primary audience for the book. Using several simple web application examples, we'll explore all aspects of SQL and database design that are required by web developers to develop efficient and effective web pages. The sample applications in this book really are quite simple, and you may already be familiar with one or more of them, just by using them on the Web.

Of course, database use goes beyond dynamic web sites. For example, databases are also used in desktop and network applications. So even if you're working with a non-web-related application, the chances are good that you're still working with a database that uses SQL. The SQL you learn in this book can be applied in all situations where a database is used.

The Challenges to Learning SQL

We are confronted with insurmountable opportunities.

—Pogo

I first learned SQL in the late 1980s. At that time, there were no books on SQL, nor web site SQL tutorials because the Web was yet to arrive. I learned by practicing, and by reading the manual. In the 1990s, I solidified my own understanding of SQL by helping others learn, as well as by participating in email discussion lists with other SQL practitioners. Today, I'm hopelessly addicted to web discussion forums like SitePoint, and have interacted with literally thousands of people as they learn SQL too.

Some of the complaints about learning SQL that I've heard over the years include:

- Basic SQL tutorials just cover the syntax, but they use trivial examples and fail to explain anything in depth.
- Some SQL tutorials use a secret language (for example, DRI, canonical synthesis, non-trivial FD) that you'd need a PhD to understand.

- Some SQL tutorials are tantalizingly close to what you're looking for, but fail to *close the sale*. That's because they're unable to relate their examples to your own real-world situation.

By using common web-related sample applications, this book will cover not only simple examples of SQL that are relevant to you, but also more complex concepts using terminology I hope you won't find too obscure.

What's in This Book?

This book comprises the following chapters. In the first eight chapters, we'll learn about SQL, the language, its various statements and clauses, and how to use SQL to store and retrieve database data. These chapters are organized to provide first an introduction to the SQL language, then an overview of the `SELECT` statement, followed by an examination of each of the `SELECT` statement's clauses. In the last three chapters, we'll learn how to design databases effectively, taking into consideration column data types, table relationships, primary and foreign keys, and so on.

Why this separation? Why do we postpone learning about designing tables until well after the `SELECT` statement has been thoroughly dissected? Because effective database design requires an understanding of how SQL works. You must walk before you can run. If you're new to SQL, you'll want to focus on learning SQL first, rather than be prematurely sidetracked on the whys and wherefores of database design issues.

The SQL Language

Chapter 1: An Introduction to SQL

This introductory chapter will put the SQL language into a perspective relevant to a typical web developer. You'll learn the difference between a statement and a clause, as well as data definition language (DDL) and data manipulation language (DML). You'll also go on a whirlwind tour through all the common SQL statements: `CREATE`, `ALTER`, `DROP`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT`.

Chapter 2: An Overview of the `SELECT` Statement

If SQL is all about database queries, then the `SELECT` statement is where all the action is. This overview will dissect the SQL `SELECT` statement into its compon-

ent clauses and give you a taste of what's to come. The next six chapters will look at each clause in detail.

Chapter 3: The FROM Clause

Few SQL books begin with the FROM clause, but this is where it all begins; the FROM clause is executed first and all other clauses use the tabular results it produces. That's why it's a great place to start our in-depth examination of all the clauses of the SELECT statement. This chapter will ease you through the tricky subject of database table joins, where you'll easily master the concepts of the inner join, left outer join, right outer join, full outer join, and cross join. We'll also touch on the topics of database views and subqueries (or derived tables).

Chapter 4: The WHERE Clause

The WHERE clause filters the results of the FROM clause. This chapter will teach you all about how to express conditions using the SQL keywords: LIKE, BETWEEN, AND, OR, IN, EXISTS, and NOT, as well as correlated subqueries. We'll also discuss performance issues and indexing.

Chapter 5: The GROUP BY Clause

The GROUP BY clause aggregates the result of the FROM and WHERE clauses into groups. Chapter 5 will teach you how grouping works and the rules for its use.

Chapter 6: The HAVING Clause

The HAVING clause filters the group rows produced by the GROUP BY clause. Chapter 6 will show you how to write HAVING clause conditions, and how to use the HAVING clause without a GROUP BY clause.

Chapter 7: The SELECT Clause

The SELECT clause defines the columns in the final result set. Chapter 7 will show you how to use the SELECT clause effectively, and how its scope changes in the presence or absence of the GROUP BY clause. This chapter also does a survey of the common SQL aggregate functions (like SUM and COUNT) and scalar functions (like CASE and SUBSTRING) available in most database systems.

Chapter 8: The ORDER BY Clause

The ORDER BY clause determines the order in which the result set is returned. Chapter 8 is all about sequencing the various data types, and the difference between sequencing and grouping. It'll teach you how to write effective ORDER

BY expressions, including how to use the CASE functions to implement special sequencing.

Database Design

Chapter 9: SQL Data Types

This chapter provides a detailed look at the various numeric, character, and temporal data types available for columns, and the rationales for their use. We tour the common numeric, string, and date functions available in most database systems. This chapter also contains a section on the column constraints: NULL, NOT NULL, DEFAULT, and CHECK.

Chapter 10: Relational Integrity

Chapter 10 introduces some topics that are the source of much befuddlement for many people new to databases. It's about relational integrity, the real heart and soul of effective database design. This chapter discusses the concept of identity, primary keys, and uniqueness, and also extensively covers the concept of the relationships between database entities: how they're expressed with database modelling and how they're implemented using foreign keys.

Chapter 11: Special Structures

Chapter 11 examines some of the common database structures that can be used to implement complex relationships between entities. This chapter takes a look at how to join to a table twice, join a table to itself, and how to implement the concept of keywords (or tagging)—a many-to-many relationship.

The sample applications used throughout the book are described in the appendices, and all the examples of SQL in the book are taken from these applications. Instructions are given on creating the applications and loading them with data.

How to Gain Help

SQL has been around a few decades and hardly ever changes. Despite major database systems like MySQL and SQL Server constantly upgrading themselves, the underlying SQL language is stable, so everything you learn in this book will be applicable for years to come.

However, SQL is an immensely broad topic. Of necessity, much needs to be left out of any single book that tries to cover all of SQL. Therefore, if you have any questions

[Order the print version of this book to get all 200+ pages!](#)

about SQL or database design, you might wish to seek further help. Two sources you can try are the SitePoint Forums and the web site for this book.

The SitePoint Forums

The SitePoint Forums¹ are discussion forums where you can ask questions about anything related to web development. You may, of course, answer questions, too. That's how a discussion forum site works—some people ask, some people answer—and most people do a bit of both. Sharing your knowledge benefits others and strengthens the community. A lot of fun and experienced web designers and developers hang out there. It's a good way to learn new stuff, get questions answered in a hurry, and just have fun.

The SitePoint Forums include a main forum for Databases, and a subforum for MySQL specifically (because of the immense popularity of this database system).

- Databases: <http://www.sitepoint.com/forums/forumdisplay.php?f=88>
- MySQL: <http://www.sitepoint.com/forums/forumdisplay.php?f=182>

The Book's Web Site

Located at <http://www.sitepoint.com/books/sql1/>, the web site that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains each and every line of example source code that's printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the archive.²

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page on the book's web site will always have the latest information about known typographical and code errors.

¹ <http://www.sitepoint.com/forums/>

² <http://www.sitepoint.com/books/sql1/code.php>

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters, such as *SitePoint Design View*, *SitePoint Market Watch*, and *SitePoint Tech Times*, to name a few. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of web development. Sign up to one or more SitePoint newsletters at <http://www.sitepoint.com/newsletter/>.

Your Feedback

If you can't find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is books@sitepoint.com. We have a well-staffed email support system set up to track your inquiries, and if our support team members are unable to answer your question, they'll send it straight to us. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

Conventions Used in This Book

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

Code Samples

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

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

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css (*excerpt*)

```
border-top: 1px solid #333;
```

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

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

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

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

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

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➡ets-come-of-age/");
```

Tips, Notes, and Warnings



Hey, You!

Tips will give you helpful little pointers.



Ahem, Excuse Me ...

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



Make Sure You Always ...

... pay attention to these important points.



Watch Out!

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

Acknowledgments

I'd like to thank Andrew Tetlaw, SitePoint's Technical Editor, and Joe Celko, the main reviewer. Both gave valuable advice that made the book better.

Joe, in particular, has been a tremendous influence on me as I learned SQL over the years. I met Joe at a database conference in the early 1990s and have been a big fan ever since, so I was very pleased that he agreed to review the book. As you may know, Joe has been an outspoken SQL advocate for decades. He helped establish the SQL standard, and has written numerous SQL columns, as well as many books on SQL and databases. In particular, his book *SQL for Smarties*³ is a *must have* for anyone venturing into advanced SQL.

I also owe Joe an apology for willfully disregarding ISO-11179, an international standard for assigning data element names, primarily in my choice of “id” as the name of several of my table columns. Joe was also dissatisfied with my coding style, where I place the comma that separates items in a list—on a new line, at the front. However, it's *my* coding style and I trust that you'll not find it too disruptive; it's served me well for many years, and I write a lot of SQL.

³ *SQL for Smarties* 3rd edition (San Francisco, Morgan Kaufmann, 2005)

Chapter 1

An Introduction to SQL

Almost every web site nowadays, it seems, uses a database. Your objective, as a web developer, is to be able to design or build web sites that use a database. To do this, you must acquire an understanding of and the ability to use **Structured Query Language** (SQL), the language used to communicate with databases.

We'll start by introducing the SQL language and the major SQL statements that you'll encounter as a web developer.

SQL Statement Overview

In the past, SQL has been criticized for having an inappropriate name. Structured Query Language lacks a proper structure, does more than just queries, and only barely qualifies as a programming language. You might think it fair criticism then, but let me make three comments:

- **Structure** refers to the fact that SQL is about tables of data or, more specifically, tabular structures. A table of data has columns and rows. There are many instances where we'll encounter an alternative that isn't, strictly speaking, a table, but looks and acts like one. This tabular structure will be explained in Chapter 3.

2 Simply SQL

- While SQL includes many different types of statements, the main one is the **SELECT** statement, which performs a *query* against the database, to retrieve data. Querying data effectively is *where the action is*, the primary focus of the first eight chapters. Designing the database effectively is covered in the last three chapters.
- The SQL language has been standardized. This is immensely important, because when you learn effective SQL, you can apply your skills in many different database environments. You can develop sites for your client or boss using any of today's common database systems—whether proprietary or open source—because they all support SQL.

Those three concepts—tabular structures, effective querying, and SQL standards—are the secret to mastering SQL. We'll see these concepts throughout the book.



SQL or Sequel?

Before the real fun begins, let's put to rest a question often asked by newcomers: how do you pronounce SQL?

Some people say the letters, “S-Q-L.” Some people pronounce it as a word, “sequel”. Either is correct. For example, the database system SQL Server (by Microsoft, originally by Sybase) is often pronounced “sequel server”. However, SQL, by itself—either the language in general or a given statement in that language—is usually pronounced as S-Q-L.

Throughout this book, therefore, SQL is pronounced as *S-Q-L*. Thus, you will read about *an* SQL statement and not *a* SQL statement.

We'll begin our overview of SQL statements by looking at their components: keywords, identifiers, and constants.

Keywords, Identifiers, and Constants

Just as sentences are made up of words that can be nouns, verbs, and so on, an SQL statement is made up of words that are keywords, identifiers, and constants. Every word in an SQL statement is always one of these:

- Keywords** These are words defined in the SQL standard that we use to construct an SQL statement. Many keywords are mandatory, but most of them are optional.
- Identifiers** These are names that we give to database objects such as tables and columns.
- Constants** These are literals that represent fixed values.

Let's look at an example:

```
SELECT name FROM teams WHERE id = 9
```

Here is a perfectly respectable SQL statement. Let's examine its keywords, identifiers, and constants:

- **SELECT**, **FROM**, and **WHERE** are keywords. **SELECT** and **FROM** are mandatory, but **WHERE** is optional. We'll cover only the important keywords in SQL in this book. However, they're all listed in Appendix D for your reference.
- **name**, **teams**, and **id** are identifiers that refer to objects in the database. **name** and **id** are column names, while **teams** is a table name.

We'll define both columns and tables later on in this chapter but, yes, they are exactly what you think they are.

- The *equals* sign (=) is an **operator**, a special type of keyword.
- 9 is a *numeric constant*. Again, we'll look at constants later in the chapter.

So there you have it. Our sample SQL statement is made up of keywords, identifiers, and constants. Not so mysterious.

Clauses

In addition, we often speak of the clauses of an SQL statement. This book has entire chapters devoted to individual clauses. A **clause** is a portion of an SQL statement. The name of the clause corresponds to the SQL keyword that begins the clause. For example, let's look at that simple SQL statement again:

```
SELECT
  name
FROM
  teams
WHERE
  id = 9
```

The SELECT clause is:

```
SELECT
  name
```

The FROM clause is:

```
FROM
  teams
```

The WHERE clause is:

```
WHERE
  id = 9
```



Coding Style

You'll have noticed that, this time, the query is written with line breaks and indentation. Even though line breaks and extra white space are ignored in SQL—just as they are in HTML—readability is *very* important. Neatness counts, and becomes more pertinent with longer queries: the tidier your queries the more likely you are to spot errors. I'll say more on coding style later.

Syntax

Each clause in an SQL statement has syntax rules for how it may be written. **Syntax** simply means how the clause is put together—what keywords, identifiers, and constants it consists of, and, more importantly, whether they are in the correct order, according to SQL's *grammar*. For example, the `SELECT` clause must start with the keyword `SELECT`.



Syntax and Semantics

In addition to syntax, **semantics** is another term sometimes used in discussing SQL statements. These terms simply mean the difference between what the SQL statement actually says versus what you intended it to say; *syntax* is what you said, *semantics* is what you meant.

The database system won't run any SQL statement with a syntax error. To add insult to injury, the system can only tell you if your SQL statement has a syntax error; it doesn't know what you actually meant.

To demonstrate the difference between syntax and semantics, suppose we were to rewrite the example from the previous section like so:

```
FROM teams WHERE id = 9 SELECT name
```

This seems to make some sense. The *semantics* are clear. However, the *syntax* is wrong. It's an invalid SQL statement. More often, you'll get syntactically correct queries that are semantically incorrect. Indeed, we'll come across some of these as we go through the book and discuss how to correct them.

Up to this point, I've alluded to a couple of database object types: tables and columns. To reference database objects in SQL statements we use their identifiers, which are names that are assigned when the objects are first created. This leads naturally to the question of *how* those objects are created.

Before we answer that, let's take a moment to introduce some new terminology. SQL statements can be divided into two types: DDL and DML.

Data Definition Language (DDL)

DDL is used to manage *database objects* like tables and columns.

Data Manipulation Language (DML)

DML is used to manage the *data* that resides in our tables and columns.

The terms DDL and DML are in common use, so if you run into them, remember that they're just different types of SQL statements. The difference is merely a convenient way to distinguish between the types of SQL statements and their effect on the database. DDL changes the database structure, while DML changes only the data. Depending on the project, and your role as a developer, you may not have the authority or permission to write DDL statements. Often, the database already exists, so rather than change it, you can only manipulate the data in it using DML statements.

The next section looks at DDL SQL statements, and how database objects are created and managed.

Data Definition Language

In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move.

—Douglas Adams

Where do database objects like tables and columns come from? They are created, modified, and finally removed from the database using DDL, the Data Definition Language part of SQL. Indeed those three tasks are accomplished using the `CREATE`, `ALTER`, and `DROP` SQL statements.



Trying Out Your SQL

It's one thing to see an example of SQL syntax, and another to adapt it to your particular circumstance or project. Trying out your SQL statements is a great way to learn. If you have some previous SQL experience, you already know this (and might want to skip ahead to Chapter 2). If you are new to SQL, and want to experiment with the following DDL statements, keep in mind that you can always start over. What you **CREATE**, you can **ALTER**, or **DROP**, if necessary.

Appendix A explains how to set up a testing environment for five popular database systems—MySQL, PostgreSQL, SQL Server, DB2, and Oracle—and Appendix C contains a number of DDL scripts you can try running if you wish.

CREATE, ALTER, and DROP

Of the many DDL statements, **CREATE**, **ALTER**, and **DROP** are the three main ones that web developers need to be aware of. (The others are more advanced and beyond the scope of this book.) Even if you haven't been granted the authority or permission to execute DDL statements on a given project, it helps to know the DDL to see how tables are structured and interconnected.

The **CREATE** Statement

Earlier on, I suggested that a tabular structure is one of the main concepts you need to understand when learning SQL. It's actually quite simple, and a table of data looks exactly like you would intuitively expect it to—it has columns and rows. Each table contains information concerning a set of items. Each **row** in a table represents a single item. Each **column** represents one piece of information that can be stored about each item. Figure 1.1 provides a visualization of a table called **teams** with three columns named **id**, **name**, and **conference**. The table pictured also contains some data; each row in the table represents a single team and can store three pieces of information about that individual team: its id number, its name, and its conference (its division or league).

Table Name

	id	name	conference
	9	Riff Raff	F
	37	Havoc	F
	63	Brewers	C

Each row represents an item

Each column in a row represents a property of the item

Figure 1.1. Tables have rows, and rows have columns

Here’s an example of a DDL statement; this is the statement that creates the database table pictured in Figure 1.1:

```
CREATE TABLE teams
(
  id          INTEGER      NOT NULL PRIMARY KEY,
  name        VARCHAR(37)  NOT NULL,
  conference  VARCHAR(2)   NULL
)
```

The `CREATE TABLE` statement creates the `teams` table—but not the data—with three columns named `id`, `name`, and `conference`. This is a table used in the *Teams and Games* sample application, one of several sample applications used in the book. All the applications are described in Appendix B.



The Order of Columns

Note that while tables are represented graphically with the columns always in the same order, this is for our ease of reference only. The database itself doesn’t care about the order of the columns.

It would be optimistic to expect you to understand everything in the `CREATE TABLE` statement above at this stage. (I’m sure some of you, new to SQL, might be wondering “What’s an `id`?” or “What does `PRIMARY KEY` do?” and so on.) We simply want to see an example of the `CREATE TABLE` statement, and not be sidetracked by design issues for the Teams and Games application.

Note that the keywords of the `CREATE TABLE` statement are all in capital letters, while the identifiers are all in lower case letters. This choice is part of my coding style.



Upper Case or Lower Case?

Although it's of no consequence to SQL whether a font appears in caps or lower case, identifiers may indeed be case-sensitive. However, I'd strongly advise you to create your database objects with lower case letters to avoid syntax problems with unknown names.

Notice also the formatting and white space. Imagine having to read this SQL statement all on one long line:

```
CREATE TABLE teams ( id INTEGER NOT NULL PRIMARY KEY,
➡  name VARCHAR(37), conference VARCHAR(2) NULL )
```

Neatness helps us to spot parts of the statement we have omitted or misspelled, like the `NOT NULL` that was accidentally left off the name column in the one line version of the statement above. Did you spot the omission before you read this?

Looking at the sample `CREATE TABLE` statement, we see that each of the three columns is given a data type (e.g., `INTEGER`, `VARCHAR`), and is also designated `NULL` or `NOT NULL`. Again, please don't worry if these terms are new to you. We will discuss how they work and what they're for in Chapter 9. This introductory chapter is not supposed to teach you the SQL statements in detail, merely introduce them to you and briefly describe their general purpose.

Are there other database objects that we can create besides tables? Yes. There are schemas, databases, views, triggers, procedures and several more but we're getting ahead of ourselves again. Many `CREATE` statements are for administrative use only and hence solely used by designated database administrators (DBAs). Learning to be a DBA is such a large subject, it requires a book of its own just to cover its scope! Needless to say, our coverage of Database Administration topics will be kept to a minimum.

The ALTER Statements

As its name suggests, **ALTER** changes an object in a database. Here's an example **ALTER** statement:

```
ALTER TABLE teams DROP COLUMN conference
```

The keyword **DROP** identifies what's being dropped, or removed, from the table. In this example, the **teams** table is being altered by removing the **conference** column. Once the column is dropped, it's no longer part of the table.

Note that if we tried to run the same **ALTER** statement for a second time, a syntax error would occur because the database cannot remove a column that does not exist from a table. Syntax errors can arise from more than just the improper construction of the SQL statement using keywords, identifiers, and constants. Many syntax errors arise from attempting to alter what are perceived (wrongly) to be the current structure or current contents of the table.

The DROP Statement

The **DROP** statement—to round out our trio of basic DDL statements—drops, removes, deletes, obliterates, cancels, blows away, and/or destroys the object it is dropping. After the **DROP** statement has been run, the object is gone.

The syntax is as simple as it can be:

```
DROP TABLE teams
```

To summarize, the Data Definition Language statements **CREATE**, **ALTER**, and **DROP** allow us to manage database objects like tables and columns. In fact, they can be very effective when used together, such as when you need to start over.

Starting Over

Database development is usually iterative. Or rather, when building and testing your table (or tables—there is seldom only one) you will often find yourself repeating one of the following patterns:

- **CREATE**, then test

First, you create a table. Then you test it, perhaps by running some `SELECT` queries, to confirm that it works. The table is so satisfactory that it can be used exactly as it is, indefinitely. If only life were like this more often ...

- `CREATE`, then test ... `ALTER`, then test ... `ALTER`, then test ...

You create and test a table, and it's good enough to be used regularly, such as when your web site goes live. You alter it occasionally, to make small changes. Small changes are easier than larger changes, especially if much code in the application depends on a particular table structure.

- `CREATE`, then test ... `DROP`, `CREATE`, then test ...

After creating and testing a table the first time, you realize it's wrong. Or perhaps the table has been in use for some time, but is no longer adequate. You need to drop it, change your DDL, create it again (except that it's different now), and then test again.

Dropping and recreating, or starting over, becomes much easier using an SQL **script**. A script is a text file of SQL statements that can be run as a batch to create and populate database objects. Maintaining a script allows you to start over easily. Improvements in the design—new tables, different columns, and so on—are incorporated into the SQL statements, and when the script is run, these SQL statements create the objects using the new design. Appendix C contains SQL scripts used for the sample applications in this book. These scripts and more are available to download from the web site for this book, at: <http://www.sitepoint.com/books/sql1/>.

Data Manipulation Language

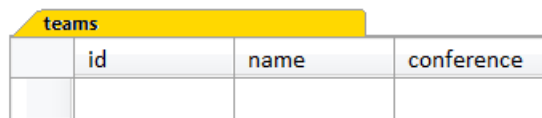
In the last section, we covered the three main SQL statements used in Data Definition Language. These were `CREATE`, `ALTER`, and `DROP`, and they are used to manage database objects like tables and columns.

Data Manipulation Language has three similar statements: `INSERT`, `UPDATE`, and `DELETE`. These statements are used to manage the data within our tables and columns.

Remember the earlier CREATE statement example:

```
CREATE TABLE teams
(
  id            INTEGER      NOT NULL PRIMARY KEY,
  name          VARCHAR(37)  NOT NULL,
  conference    VARCHAR(2)   NULL
)
```

This statement creates a table called `teams` that has three columns, pictured in Figure 1.2.



teams			
	id	name	conference

Figure 1.2. The teams table

Once the table has been created, we say it exists. And once a table exists we may place our data in it, and we need a way to manage that data. We want to use the table the way it's currently structured, so DDL is irrelevant for our purposes here (that is, changes aren't required).

Instead, we need the three DML statements, `INSERT`, `UPDATE`, and `DELETE`.

INSERT, UPDATE, and DELETE

Until we put data into it, the table is empty. Managing our data may be accomplished in several ways: adding data to the table, updating some of the data, inserting some more data, or deleting some or all of it. Throughout this process, the table structure stays the same. Just the table contents change.

Let's start by adding some data.

The INSERT Statement

The `INSERT` DML statement is similar to the `CREATE` DDL statement, in that it creates a new object in the database. The difference is that while `CREATE` creates a new table and defines its structure, `INSERT` creates a new row, inserting it and the data it contains into an existing table.

The INSERT statement inserts one or more rows. Here is our first opportunity to see rows in action. Here is how to insert a row of data into the teams table:

```
INSERT INTO teams
  ( id , name , conference )
VALUES
  ( 9 , 'Riff Raff' , 'F' )
```

The important part to remember, with our tabular structure in mind, is that the INSERT statement inserts *entire rows*. An INSERT statement should contain two comma-separated lists surrounded by parentheses. The first list identifies the columns in the new row into which the constants in the second list will be inserted. The first column named in the first list will receive the first constant in the second list, the second column has the second constant, and so on. *There must be the same number of columns specified in the first list as constants given in the second, or an error will occur.*

In the above example, three constants, 9, 'Riff Raff', and 'F' are specified in the VALUES clause. They are inserted, into the id, name, and conference columns respectively of a single new row of data in the teams table. Strings, such as 'Riff Raff', and 'F', are surrounded by single quotes to denote their beginning and end. We'll look at strings in more detail in Chapter 9.

You are allowed (but it would be unusual) to write this INSERT statement as:

```
INSERT INTO teams
  ( conference , id , name )
VALUES
  ( 'F' , 9 , 'Riff Raff' )
```

We noted earlier that the database itself doesn't care about the order of the columns within a table; however, it's common practice to order the columns in an INSERT statement in the order in which they were created for our own ease of reference. As long as we make sure that we list columns and their intended values in the correct corresponding order, this version of the INSERT statement has exactly the same effect as the one preceding it.

Sometimes you may see an `INSERT` statement like this:

```
INSERT INTO teams
VALUES
  ( 9 , 'Riff Raff' , 'F' )
```

This is perhaps more convenient, because it saves typing. The list of columns is *assumed*. The columns in the new row being inserted are populated according to their perceived position within the table, based on the order in which they were originally added when the table was created. However, we must supply a value for every column in this variation of `INSERT`; if we aren't supplying a value for each and every column, which happens often, we can't use it. If you do, the perceived list of columns will be longer than the list of values, and we'll receive a syntax error.

My advice is to always specify the list of column names in an `INSERT` statement, as in the first example. It makes things much easier to follow.

Finally, to insert more than one row, we could use the following variant of the `INSERT` statement:

```
INSERT INTO teams
  ( conference , id , name )
VALUES
  ( 'F' , 9 , 'Riff Raff' ),
  ( 'F' , 37 , 'Havoc' ),
  ( 'C' , 63 , 'Brewers' )
```

This example shows an `INSERT` statement that inserts three rows of data, and the result can be seen in Figure 1.3. Each row's worth of data is specified within a set of parentheses, known as a **row constructor**, and each row constructor is separated by a comma.

teams			
	id	name	conference
	9	Riff Raff	F
	37	Havoc	F
	63	Brewers	C

Figure 1.3. The result of the `INSERT` statement: three rows of data

Next up, we want to change some of our data. For this, we use the `UPDATE` statement.



A Note on Multiple Row Constructors

While the syntax in the above example, where one `INSERT` statement inserts multiple rows of data, is valid SQL, not every database system allows the `INSERT` statement to use multiple row constructors; those that do allow it include DB2, PostgreSQL, and MySQL. If your database system's `INSERT` statement allows only one row to be inserted at a time, as is the case with SQL Server, simply run three `INSERT` statements, like so:

```
INSERT INTO teams
  ( id , conference , name )
VALUES
  ( 9 , 'F' , 'Riff Raff' )
;
INSERT INTO teams
  ( id , conference , name )
VALUES
  ( 37 , 'F' , 'Havoc' )
;
INSERT INTO teams
  ( id , conference , name )
VALUES
  ( 63 , 'C' , 'Brewers' )
;
```

Notice that a semicolon (;) is used to separate SQL statements when we're running multiple statements like this, not unlike its function in everyday language. Syntactically, the semicolon counts as a keyword in our scheme of keywords, identifiers, and constants. The comma, used to separate items in a list, does too.

The `UPDATE` Statement

The `UPDATE` DML statement is similar to the `ALTER` DDL statement, in that it produces a change. The difference is that, whereas `ALTER` changes the structure of a table, `UPDATE` changes the data contained within a table, while the table's structure remains the same.

Let's pretend that the team Riff Raff is changing conferences so we need to update the value in the `conference` column from F to E; we'll write the following `UPDATE` statement:

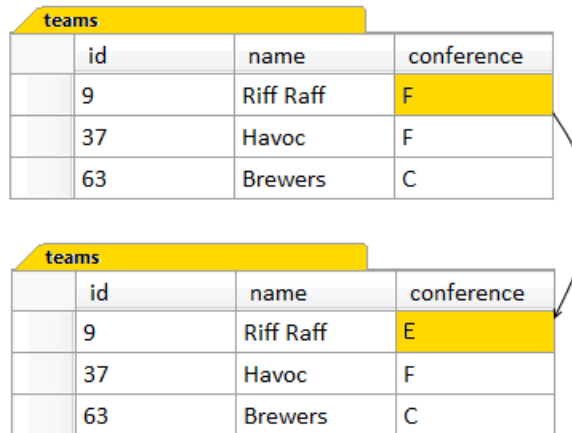
```
UPDATE
  teams
SET
  conference = 'E'
```

The above statement would change the value of the `conference` column in every row to E. This is not really what we wanted to do; we only wanted to change the value for one team. So we add a `WHERE` clause to limit the rows that will be updated:

Teams_04_UPDATE.sql (excerpt)

```
UPDATE
  teams
SET
  conference = 'E'
WHERE
  id = 9
```

As shown in Figure 1.4, the above example will update only one value. The `UPDATE` clause alone would change the value of the `conference` column in every row, but the `WHERE` clause limits the change to just the one row: where the `id` column has the value 9. Whatever value the `conference` column had before, it now has E after the update.



teams			
	id	name	conference
	9	Riff Raff	F
	37	Havoc	F
	63	Brewers	C

teams			
	id	name	conference
	9	Riff Raff	E
	37	Havoc	F
	63	Brewers	C

Figure 1.4. Updating a row in a table

Sometimes, we'll want to update values in multiple rows. The `UPDATE` statement will set column values for every row specified by the `WHERE` clause. The classic example, included in every textbook (so I simply had to include it too, although it isn't part of any of our sample applications), is:

```
UPDATE
  personnel
SET
  salary = salary * 1.07
WHERE
  jobgrade <= 4
```

Here, everyone is scoring a 7% raise, but only if their `jobgrade` is 4 or less. The `UPDATE` statement operates on multiple rows simultaneously, but only on those rows specified by the `WHERE` clause.

Notice that the existing value of the `salary` column is used to determine the new value of the `salary` column. `UPDATE` operates on each row independently of all others, which is exactly what we want, as it's likely that the salary values are different for most rows.

Finally, there is the `DELETE` statement.

The DELETE Statement

The DELETE DML statement is similar to the DROP DDL statement, in that it removes objects from the database. The difference is that DROP removes a table from the database, while DELETE removes entire rows of data from a table, but the table continues to exist:

Teams_05_DELETE.sql (excerpt)

```
DELETE
FROM
  teams
WHERE
  id = 63
```

Once again, like the UPDATE statement, the scope of the DELETE statement is every row which satisfies the WHERE clause. If there is no WHERE clause, all the rows are deleted and the table is left empty; it has a structure, but no rows.

Finally, we are ready to meet the SELECT statement.

The SELECT Statement

The SELECT statement is usually called a **query**. Informally, all SQL statements are sometimes called queries (as in “I ran the DELETE query and received an error”), but the SELECT statement is truly a query because all it does is retrieve information from the database.

When we run a SELECT query against the database, it can retrieve data from one or more tables. Exactly how the data in multiple tables is combined, collated, compared, summarized, sorted, and presented—*by a single query*—is what makes SQL so wonderful.

The power is outstanding. The simplicity is amazing. SQL allows us to produce complex, customized information with a minimum of fuss, in a declarative, non-procedural way, using a small number of keywords.

SELECT is our fourth DML statement, although the operation it performs on the data is simply to retrieve it. Nothing is changed in the database. This is one reason why I prefer to discuss SELECT separately from the other three DML statements. Another is that it breaks up the pleasant symmetry between the DDL and DML statements:

- DDL: CREATE, ALTER, DROP
- DML: INSERT, UPDATE, DELETE ... and SELECT

The SELECT Retrieves Data

A simple SELECT statement has two parts, or **clauses**. Both are mandatory:

```
SELECT expression(s) involving keywords, identifiers, and constants
FROM tabular structure(s)
```

The purpose of the SELECT statement is to retrieve data from the database:

- the SELECT clause specifies what you want to retrieve, and
- the FROM clause specifies where to retrieve it from.

The SELECT clause consists of one or more expressions involving keywords, identifiers, and constants. For example, this SELECT clause contains one expression, consisting of a single identifier:

```
SELECT
  name
FROM
  teams
```

In this case the expression in the SELECT clause is `name`, which is a column name. However, the SELECT clause can contain many expressions, simply by listing them one after another, using commas as separators. For example, we may want to return the contents of several columns from rows in the `teams` table:

```
SELECT
  id, name, conference
FROM
  teams
```

In addition, each expression can be more complex, consisting of formulas, calculations, and so on. Ultimately then, the SELECT clause can be fairly complex, but it gives us the ability to include everything we need to return from the database. We examine the SELECT clause in Chapter 7.

Similarly, the **FROM** clause can be multifaceted. The **FROM** clause specifies the tabular structure(s) that contain the data that we want to retrieve. In this chapter we've seen sample queries in which the **FROM** clause specified a single table; complexity in the **FROM** clause occurs when more than one table is specified. I suggested earlier that tabular structures are one of the secrets to mastering SQL. We'll cover them in detail in Chapter 3.

We'll have an overview of the **SELECT** statement and its optional clauses, **WHERE**, **GROUP BY**, **HAVING**, and **ORDER BY**, in Chapter 2, and then look in detail at each of its clauses in chapters of their own.

The **SELECT** Statement Produces a Tabular Result Set

One important fact to remember about **SELECT** is that the result of running a **SELECT** query is *a table*.

When your web application (whether it is written in PHP, Java, or any other application language) runs a **SELECT** query, the database system returns a tabular structure, and the application handles it accordingly, as rows and columns of data. The query might return a list of selected items in a shopping cart, or posts in active forums threads, or whatever your web application needs to retrieve from your database.

We say that the **SELECT** statement produces a tabular **result set**. A result set is not actually a table in the database; it is *derived from* one or more tables in the database, and delivered, as tabular data, to your application.

One final introduction will complete our introduction to SQL: a comment about standard SQL.

Standard SQL

The nice thing about standards is that there are so many to choose from.

—Andrew S. Tanenbaum

In the beginning, I mentioned that SQL is a standard language. SQL has been standardized both nationally and internationally. If you are relatively new to SQL, *do not* look for any information on the standard just yet; you're only going to confuse

yourself. The SQL standard is exceedingly complex. The standard is also not freely available; it must be purchased from the relevant standards organisations.

The fact that the standard exists, though, is very important, and not just because it makes the skill of knowing SQL *highly* portable. The SQL standard is being adopted, in increasing measures, by all relational database systems. New database software releases always seem to mention some specific features of the standard that are now supported.

So as well as your knowledge of using simple SQL to produce comprehensive results being portable, there is a better chance that your next project's database system will actually support the techniques that you already know. The industry and its practitioners are involved in a positive feedback loop.

And yet, there are differences between standard SQL and the SQL supported by various database systems you'll encounter—sometimes maddeningly so. These variations in the language are called **dialects** of SQL. Numerous proprietary extensions to standard SQL have been invented for the various different database systems. These extensions can be considerable, occasionally pointless, often counterintuitive, and sometimes obscure.

There is only one sane way to cope.

Read The Fine Manual

Never memorize what you can look up in books.

—Albert Einstein

There will be occasions throughout this book where I'll suggest referring to the manual. This will be the manual for your particular database system, whatever it may be. All database systems have manuals—often a great many—but fortunately, most are of interest only to DBAs. The one you want is typically called *SQL Reference*.

After more than 20 years of writing SQL, I still need to look up certain parts of SQL. Granted, I have committed most of standard SQL to memory, but there are always nuances that trip me up, and new features to learn, and all those proprietary extensions ...



Finding the Manual

Appendix A gives links for downloading and installing five popular database systems: MySQL, PostgreSQL, SQL Server, DB2, and Oracle. In addition, links are given to the online SQL reference manuals for each of these systems.

Make it easy on yourself. Bookmark the SQL reference manual on your computer, on your company server, or on the Web. Be prepared for syntax errors. But be reassured that they're easy to fix, if you know where to look in the manual.

Wrapping Up: an Introduction to SQL

In this chapter, we covered *lots* of ground. I hope you're not feeling completely overwhelmed. The purpose of the whirlwind tour was simply to put the SQL language into the perspective that a typical web developer needs; there are SQL statements for everything you need to do, and, all things considered, these statements are quite simple.

The two basic types of SQL statement are:

- Data Definition Language (DDL) statements: `CREATE`, `ALTER`, `DROP`
- Data Manipulation Language (DML) statements: `INSERT`, `UPDATE`, `DELETE ...` and `SELECT`

If you're building your own database, you'll need to know DDL, but you should have some experience using DML first. Designing databases is the subject of the last three chapters in this book.

As mentioned earlier, SQL is mainly about queries, and the `SELECT` statement is *where it's at*. Chapter 2 begins our in-depth look at the `SELECT` statement, providing an overview of its six clauses.

Chapter 2

An Overview of the **SELECT** Statement

In Chapter 1, our quick introduction to SQL, we briefly met the main SQL statements that we will encounter as web developers:

- CREATE, ALTER, DROP
- INSERT, UPDATE, DELETE ... and
- *SELECT*

In this chapter, we'll begin our more detailed look into SQL with an overview of the **SELECT** statement.

The **SELECT** Statement

The **SELECT** statement's single purpose is to retrieve data from our database and return it to us in a tabular result set. In Chapter 1, we saw some of its syntax:

```
SELECT expression(s) involving keywords, identifiers, and constants  
FROM tabular structure(s)
```

The **SELECT** and **FROM** clauses are mandatory; what we didn't reveal in Chapter 1 is that the **SELECT** statement can include optional clauses used to filter, group, and

sort the returned results. Let's expand our **SELECT** statement syntax to include the optional clauses:

```
SELECT expression(s) involving keywords, identifiers, and constants
FROM tabular structure(s)
[WHERE clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause]
```

We've placed each optional clause in square brackets, [...], to denote their optional status. Don't worry—we'll introduce the **WHERE**, **GROUP BY**, **HAVING**, and **ORDER** clauses in short order.

The examples we are about to discuss are fairly simple, so this chapter will be like a quick review and set the stage for the chapters that follow. We'll look at each of its clauses in turn, with some very simple examples.



Trying Out Your SQL

If you are new to SQL, you may wish to try out the sample **SELECT** statements on your own computer or testing system. You don't have to—you can simply read along—but trying out the sample SQL for yourself is a good way to begin. Appendix C contains the SQL that creates and populates the databases used for the sample applications in this book. These scripts and more are available to download from the web site for this book, located at <http://www.sitepoint.com/books/sql1/>.

The **SELECT** and **FROM** Clauses

Only the first two clauses of the **SELECT** statement—the **SELECT** and **FROM** clauses—are mandatory, so our first **SELECT** statement will use only these.

To illustrate our **SELECT** statements, we'll use another sample application. In Chapter 1, the application was Teams and Games (although we used only the **teams** table). In this chapter, the application is a content management system, or CMS for short. Remember, the sample applications are described in Appendix B.

Content management system is a generic term that simply means a system to store, manage, and retrieve content. In most cases this means the content of a web site.

(One example is Wordpress, a popular blogging tool.¹ Our sample content management system isn’t anywhere near as sophisticated.) So before we can examine and discuss our sample `SELECT` statements, we need to have some sample tables to select from.

Content Management System

The database portion of our CMS will be designed to store articles. (The application portion of the CMS will handle the processes of adding and updating these articles, and displaying them on web pages. We’ll focus on the database portion, and how it interacts with the application.) We’ll want our content—the articles—to be stored in the database, *and* also displayed on web pages after being retrieved with SQL. One way to display them would be as individual articles shown in their entirety on their own web page. Another way might be a list of articles, such as on a site map. We might also expect to have a search page to search for articles, and so on.

Actually, they don’t have to be articles. The database tables in our CMS will work just as easily for stories, news updates, blog posts, journal/diary entries ...

Yes, that’s it—let’s call them *entries*.

The entries Table

Our CMS database, and our first sample `SELECT` statement, begins with a single table, the `entries` table. This consists of columns that hold data about each entry (article or story), such as its title, the date it was created, and so on. Figure 2.1 shows a simplified version of the table. If you look at the `CREATE` statement for this table in the section called “Content Management System” in Appendix C, you’ll see it has extra columns: an `id` column to store an ID number, an `updated` column for the date each entry was last updated, and a `content` column to store the text content of each entry. But for now we’ll just keep it simple.

¹ Others are listed here: http://en.wikipedia.org/wiki/List_of_content_management_systems

entries		
category	title	created
angst	What If I Get Sick and Die?	2008-12-30
humor	Uncle Karl and the Gasoline	2009-02-28
advice	Be Nice to Everybody	2009-03-02
humor	Hello Statue	2009-03-17
science	The Size of Our Galaxy	2009-04-03

Figure 2.1. The simplified CMS entries table

Now it's finally time to look at our first sample `SELECT` statement. This one returns two columns of data from the `entries` table:

```
SELECT
  title, category
FROM
  entries
```

Figure 2.2 shows the tabular result set produced by the above query.

Results		
title	category	
What If I Get Sick and Die?	angst	
Uncle Karl and the Gasoline	humor	
Be Nice to Everybody	advice	
Hello Statue	humor	
The Size of Our Galaxy	science	

Figure 2.2. First results—two columns of data

So our first simple `SELECT` statement produced a list of all entries, showing the title and category of each. To put it a slightly different way, the query *selected* two columns *from* a table, producing a tabular result set.



Displaying Query Results on a Web Page

The result set produced by our first sample query could be displayed on a web page using the following HTML:

```
<h2>List of Articles</h2>
<ul>
  <li>What If I Get Sick and Die? (category: angst)</li>
  <li>Uncle Karl and the Gasoline (category: humor)</li>
  <li>Be Nice to Everybody (category: advice)</li>
  <li>Hello Statue (category: humor)</li>
  <li>The Size of Our Galaxy (category: science)</li>
</ul>
```

It could also be displayed using `<table>`, `<tr>`, `<th>`, and `<td>` tags. After all, this is tabular data.

The general strategy for displaying query results on a web page is to use the application programming language to loop over the rows in the result set, generating one or more lines of output HTML for each row.

The specific mechanics of how this is achieved depends, of course, on which application programming language you're using. The point of the example is to show a simple result set—several rows of data, in two columns—being used to create a web page.

Our first SELECT statement was a very simple case of the more general syntax:

```
SELECT expression(s) involving keywords, identifiers, and constants
FROM tabular structure(s)
```

In our first SELECT statement, the SELECT clause's *expressions* were two simple columns, title and category, and the FROM clause's *tabular structure* was the entries table.

Both the SELECT clause and the FROM clause are mandatory in the SELECT statement. The next clause is optional, but it lets us be more selective about which rows to return.

The WHERE Clause

The WHERE clause is optional, but when it's used, it acts as a *filter*.

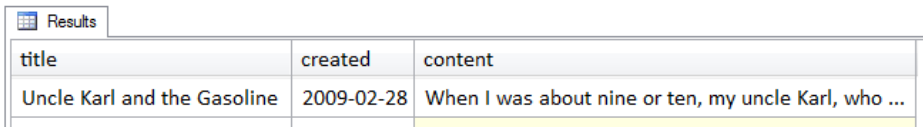
To demonstrate, let's take the simple query from the previous section and change it slightly. We'll select some different columns, and add a WHERE clause:

CMS_02_Display_an_Entry.sql (excerpt)

```
SELECT
    title, created, content
FROM
    entries
WHERE
    id = 524
```

Notice that the WHERE clause specifies a **condition**, an expression that can be evaluated by the database system as either true, false, or in some cases, indeterminable. The condition—in this case, whether the value of the `id` column is 524—is evaluated against each row of the table. If, the condition is true for any given row, that row is kept in the query result set and returned when the SELECT statement finishes executing. If the condition is false for a row, it is not included as part of the results.

Figure 2.3 shows the result set produced by this query.



Results		
title	created	content
Uncle Karl and the Gasoline	2009-02-28	When I was about nine or ten, my uncle Karl, who ...

Figure 2.3. Long field values may appear abridged (but never are)

Note that the result set consists of only one row. There are three columns in the result set, corresponding to the three columns in the SELECT clause, but just one row.

The third column, the `content` column, is a TEXT column, containing text that includes line breaks. However, it's still just one value, even though it's quite a longish value to be revealed in its entirety—as shown in Figure 2.3. (TEXT columns can actually hold values up to several megabytes in size, or even larger. We'll learn how to choose data types when designing tables in Chapter 9.)

What's the significance of the result set consisting of only one row? It means that the `WHERE` clause has *filtered out* all the rows of the `entries` table *except* for the row that has a value of 524 in the `id` column. This is the `id` value that was assigned to the entry `Uncle Karl` and the `Gasoline`. Notice that it was not necessary for the `SELECT` clause to include the `id` column, even though the `id` column was specified in the `WHERE` clause.

To recap, the `WHERE` clause specifies conditions which are evaluated against the rows of the table; these conditions act as a filter that determines which rows are returned in the query result set.

We have now covered half of the clauses of the `SELECT` statement:

```
SELECT expression(s) involving keywords, identifiers, and constants
FROM tabular structure(s)
[WHERE clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause]
```

We'll look at the next two clauses together, because that's how they are used.

The `GROUP BY` and `HAVING` Clauses

The main purpose of the `GROUP BY` clause is to have the database examine every row in the set generated by the `FROM` clause, and then filtered by the `WHERE` clause if there is one; then it groups them together by the values of one or more of their columns to produce a single new row (per group). This process is known as **aggregation**. The `GROUP BY` clause identifies the column(s) that are used to determine the groups. Each group consists of one or more rows, while all the rows in each group have the *same values* in the grouping column(s). The difficulty with this concept is understanding that, after the grouping operation has completed, the original rows are no longer available. Instead, **group rows** are produced. A group row is a new row created to represent each group of rows found during the aggregation process.

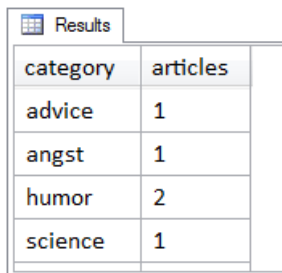
Here's an example of a `GROUP BY` query:

`CMS_03_Count_Entries_by_Category.sql` (excerpt)

```
SELECT
    category, COUNT(*) AS articles
FROM
    entries
GROUP BY
    category
```

The grouping column in this example is `category`, as specified by the `GROUP BY` clause. Each row in the `entries` table is assigned to a specific group based on the value of its `category` column. Then the magic happens. The grouping, or aggregation, of the rows in each group produces one row per group. Another way to think about this is that grouping collapses each group's rows, producing a single group row for every group.

Figure 2.4 shows the result set produced by the query above.



category	articles
advice	1
angst	1
humor	2
science	1

Figure 2.4. Results for a `GROUP BY` query

The column called `articles` contains a count of the number of table rows in each group. The name `articles` was assigned as a *column alias* for the expression `COUNT(*)`. This expression is an example of an *aggregate function*; one that counts numbers of rows. We'll meet this function later on in the section called "Aggregate Functions" in Chapter 7.

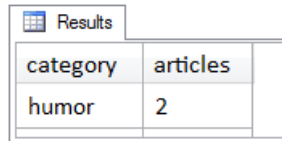
This sample `GROUP BY` query produces a count of articles in each category. This seems innocuous enough, because it's so simple. The `GROUP BY` clause proves difficult for some people only when more complex queries are attempted. Grouping is a concept that takes a bit of effort to understand, so Chapter 5 focuses entirely on that.

The **HAVING** clause works in conjunction with the **GROUP BY** clause, by specifying conditions which filter the group rows. As a simple example, let's add a **HAVING** clause to the previous example:

CMS_03_Count_Entries_by_Category.sql (excerpt)

```
SELECT
    category, COUNT(*) AS articles
FROM
    entries
GROUP BY
    category
HAVING
    COUNT(*) > 1
```

The filtering effect of the **HAVING** clause is apparent when you see the result, as shown in Figure 2.5.



category	articles
humor	2

Figure 2.5. Grouped results filtered by a **HAVING** clause

The **HAVING** clause operates only on group rows, and acts as a filter on them in exactly the same way that the **WHERE** clause acts as a filter on table rows. In this case, it filters out rows in which the number of articles is one or fewer.

We'll conclude our quick review of the **SELECT** statement with the **ORDER BY** clause.

The ORDER BY Clause

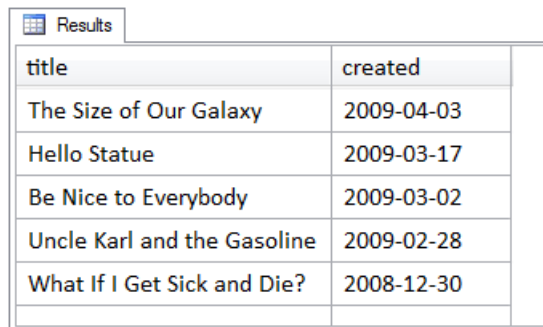
The purpose of the `ORDER BY` clause is to return the tabular result set in a specific sequence. It works just as you would expect it to; it *sorts the rows* in the result set.

Here's an example:

CMS_04_Entries_Sorted_Latest_First.sql (excerpt)

```
SELECT
    title, created
FROM
    entries
ORDER BY
    created DESC
```

The `ORDER BY` clause in the example above specifies that the results should be sorted into a descending sequence based on the `created` column. This is a typical requirement in a CMS context—to show latest articles first. The special keyword `DESC` determines that it's a descending sequence. (There's a corresponding `ASC` keyword, but `ASC` is the default and is optional.) Figure 2.6 shows the result set produced by the above query.



title	created
The Size of Our Galaxy	2009-04-03
Hello Statue	2009-03-17
Be Nice to Everybody	2009-03-02
Uncle Karl and the Gasoline	2009-02-28
What If I Get Sick and Die?	2008-12-30

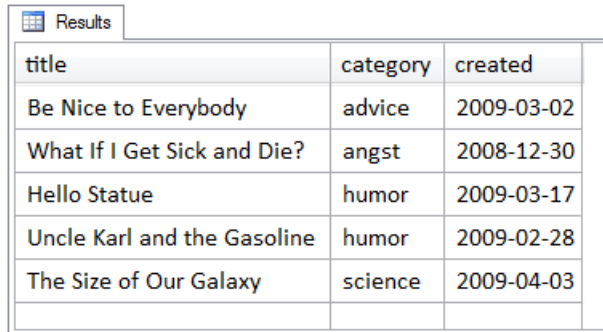
Figure 2.6. Results ordered on the `created` column

You may specify multiple columns in the `ORDER BY` clause, to give multiple levels of sequencing. Another way to describe this is to say that the `ORDER BY` clause allows any number of major and minor sort keys.

For instance, consider the following `ORDER BY` clause:

```
ORDER BY
  category, created DESC
```

In the above example, the result rows are sorted first on the `category` column from A to Z, and then the entries within each category are sorted on the `created` column, from most recent to least recent, as shown in Figure 2.7.



title	category	created
Be Nice to Everybody	advice	2009-03-02
What If I Get Sick and Die?	angst	2008-12-30
Hello Statue	humor	2009-03-17
Uncle Karl and the Gasoline	humor	2009-02-28
The Size of Our Galaxy	science	2009-04-03

Figure 2.7. Results ordered on multiple columns

Wrapping Up: the SELECT Statement

To summarize, in this chapter we conducted a quick review of the clauses of the `SELECT` statement. The syntax for the `SELECT` statement is:

```
SELECT expression(s) involving keywords, identifiers, and constants
FROM tabular structure(s)
[WHERE clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause]
```

- The `SELECT` and `FROM` clauses are mandatory, and the other clauses are optional. `SELECT` determines the columns in the result set, and `FROM` specifies where the data comes from.
- The `WHERE` clause, when present, acts as a filter on the rows retrieved from the table.

- The `GROUP BY` clause, when present, performs grouping or aggregation, in effect collapsing all the rows retrieved from the table to produce one group row per group. `HAVING` filters group rows in the same way that `WHERE` filters table rows.
- The `ORDER BY` clause is used if the rows are to be sorted and returned in a specific sequence.

Now we are ready to begin our detailed, in-depth analysis of the `SELECT` statement, starting with the `FROM` clause in Chapter 3.

Chapter 3

The FROM Clause

In Chapter 2, we broke the `SELECT` statement down into its various clauses, but looked at each clause only briefly. In this chapter, we'll begin our more detailed look at the `SELECT` statement, starting with the `FROM` clause.

The `FROM` clause can be simple, and it can also be quite complex. In all cases, though, the important point about the `FROM` clause is that it *produces a tabular structure*. This tabular structure is referred to as the *result set* of the `FROM` clause. You may also see it referred to as an intermediate result set, an intermediate tabular result set, or an intermediate table. But, no matter whether the `SELECT` query retrieves data from one table, from many tables, or from other, similar tabular structures, the result is always the same—the `FROM` clause produces a tabular structure.

In this chapter we'll review the common types of `FROM` clause that we might encounter in web development.

Why Start with the FROM Clause?

To begin writing a `SELECT` statement, my strategy is to skip over the `SELECT` clause for the time being, and write the `FROM` clause first. Eventually, we'll need to input some expressions into the `SELECT` clause and we might also need to use `WHERE`, `GROUP BY`, and the other clauses too. But there are good reasons why we should always start with the `FROM` clause:

- If we get the `FROM` clause wrong, the SQL statement will always return the wrong results. It's the `FROM` clause that produces the tabular structure, the starting set of data on which all other operations in a `SELECT` statement are performed.
- The `FROM` clause is the first clause that the database system looks at when it parses the SQL statement.

Parsing an SQL Statement

Whenever we send an SQL statement to the database system to be executed, the first action that the system performs is called **parsing**. This is how the database system examines the SQL statement to see if it has any syntax errors. First it divides the statement into its component clauses; then it examines each clause according to the syntax rules for that clause. Contrary to what we might expect, the database system parses the `FROM` clause first, rather than the `SELECT` clause.

For example, suppose we were to attempt to run the following SQL statement, in which we have misspelled `teams` as `teans`:

Teams_06_FROM_Teans.sql (excerpt)

```
SELECT
    id, name
FROM
    teans
WHERE
    conference = 'F'
```

In this case, the `FROM` clause refers to a non-existing table, so there is an immediate syntax error. If the database system *were* to parse the `SELECT` clause first, it would need to examine the table definitions of all the tables in the database, looking for one that might contain two columns called `name` and `id`. In fact, it's quite common

for a database to have several tables with two columns called `name` and `id`. Confusion could ensue and the database would require more information from us to know which table to retrieve `name` and `id` from. Hence why the database system parses the `FROM` clause first, and this is the first clause we think about as well.

FROM One Table

We've already seen the `FROM` clause with a single table. In Chapter 1, we saw the `FROM` clause specify the `teams` table:

```
SELECT
  id, name
FROM
  teams
```

In Chapter 2, we saw the `FROM` clause specify the `entries` table:

```
SELECT
  title, category
FROM
  entries
```

This form of the `FROM` clause is as simple as it gets. There must be at least one tabular structure specified, and a single table fits that requirement. When we want to retrieve data from more than one table at the same time however, we need to start using *joins*.

FROM More than One Table Using JOINS

A **join** relates, associates, or combines two tables together. A join starts with two tables, then combines—or joins—them together in one of several different ways, producing a single tabular structure (as the result of the join). Actually, the verb *to join* is very descriptive of what happens, as we'll see in a moment.

The way that the tables are joined—the *type* of join—is specified in the `FROM` clause using special keywords *as well as* the keyword `JOIN`. There are several different types of join, which I'll describe briefly, so that you can see how they differ. Then we'll look at specific join examples, using our sample applications.

Types of Join

A join combines the rows of two tables, based on a rule called a **join condition**; this compares values from the rows of both tables to determine which rows should be joined.

There are three basic types of join:

- *inner join*, created with the `INNER JOIN` keywords
- *outer join*, which comes in three varieties:
 - `LEFT OUTER JOIN`
 - `RIGHT OUTER JOIN`
 - `FULL OUTER JOIN`
- *cross join*, created with the `CROSS JOIN` keywords

To visualize how joins work, we're going to use two tables named A and B, as shown in Figure 3.1.

A	B
102	101
104	102
106	104
107	106
	108

Figure 3.1. Tables A and B



On Tables A and B

These tables are actually oversimplified, because they blur the distinction between table and column names. The join condition actually specifies the columns that must match. Further, it's unusual for tables to have just one column.

Don't worry about what A and B might actually represent. They could be anything. The idea in the following illustrations is for you to focus your attention on the *values* in the rows being joined. Table A has one column called `a` and rows with values

102, 104, 106, and 107. Table B has one column called *b* and rows with values 101, 102, 104, 106, and 108.



To Create Tables A and B

The SQL script to create tables A and B is available in the download for the book. The file is called **test_01_illustrated.sql**.

The Inner Join

For an **inner join**, *only rows satisfying the condition in the ON clause are returned*. Inner joins are the most common type of join. In most cases, such as the example below, the **ON** clause specifies that two columns must have matching values. In this case, if the value (of column *a*) in a row from one table (A) is equal to the value (of column *b*) in a row from the other table (B), the join condition is satisfied, and those rows are joined:

test_01_illustrated.sql (excerpt)

```
SELECT
  a, b
FROM
  A INNER JOIN B
    ON a=b
```

Figure 3.2 illustrates how this works.

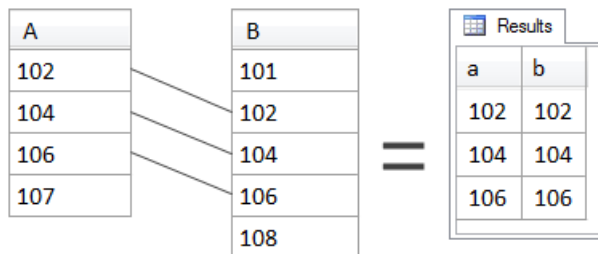


Figure 3.2. A INNER JOIN B

As you can see, a row from A is joined to a row from B when their values are equal. Thus values 102, 104, and 106 are returned in the result set. Value 107 in A has no

match in **B**, and therefore is not included in the result set. Similarly, the values 101 and 108 in **B** have no match in **A**, so they're not included in the result set either. If it's easier to do so, you can think of it as though the matching rows are actually concatenated into a single longer row on which the rest of the **SELECT** statement then operates.

Outer Joins

Next, we'll look at **outer joins**. Outer joins differ from inner joins in that *unmatched* rows can also be returned. As a result, most people say that an outer join includes rows that don't match the join condition. This is correct, but might be a bit misleading, because outer joins do include all rows that match. Typical outer joins have many rows that match, and only a few that don't.

There are three different types of outer join: left, right, and full. We'll start with the left outer join.

The Left Outer Join

For a **left outer join**, *all rows from the left table are returned, regardless of whether they have a matching row in the right table*. Which one's the left table, and which one's the right table? These are simply the tables mentioned to the left and to the right of the **OUTER JOIN** keywords. For example, in the following statement, **A** is the left table and **B** is the right table and a left outer join is specified in the **FROM** clause:

```
SELECT
  a, b
FROM
  A LEFT OUTER JOIN B
    ON a=b
```

test_01_illustrated.sql (excerpt)

Figure 3.3 shows the results of this join. Remember—left outer joins return all rows from the left table, together with matching rows of the right table, *if any*.

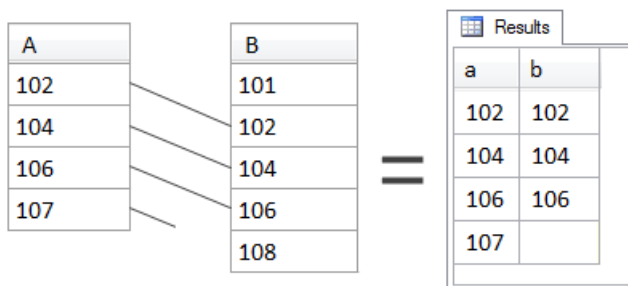


Figure 3.3. A LEFT OUTER JOIN B

Notice that all values from A are returned. This is because A is the left table. In the case of 107, which did not have a match in B, we see that it is indeed included in the results, but there is *no value* in that particular result row from B. For the time being, it's okay just to think of the value from B as missing—which, of course, for 107 it is.

The Right Outer Join

For a **right outer join**, *all rows from the right table are returned, regardless of whether they have a match in the left table*. In other words, a right outer join works exactly like a left outer join, except that all the rows of the right table are returned instead:

test_01_illustrated.sql (excerpt)

```
SELECT
  a, b
FROM
  A RIGHT OUTER JOIN B
  ON a=b
```

In the example above, A is still the left table and B is still the right table, because that's where they are mentioned in relation to the OUTER JOIN keywords. Consequently, the result of the join contains all the rows from table B, together with matching rows of table A, *if any*, as shown in Figure 3.4.

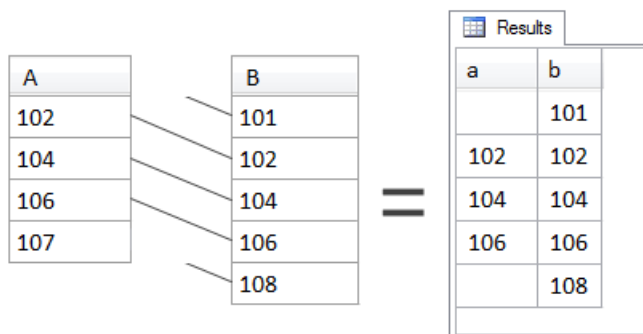


Figure 3.4. A RIGHT OUTER JOIN B

The right outer join is the reverse of the left outer join. With the same tables in the same positions—A as the left table and B as the right table—the results of the right outer join are very different from those of a left outer join. This time, all values from B are returned. In the case of 101 and 108, which did not have a match in A, they are indeed included in the results, but there is *no value* in their particular result rows from A. Again, those values from A are missing, but the row is still returned.

The Full Outer Join

For a **full outer join**, *all rows from both tables are returned, regardless of whether they have a match in the other table*. In other words, a full outer join works just like left and right outer joins, except this time all the rows of *both* tables are returned. Consider this example:

```
SELECT
  a, b
FROM
  A FULL OUTER JOIN B
    ON a=b
```

Once again, A is the left table and B is the right table, although this time it doesn't really matter. Full outer joins return all rows from both tables, together with matching rows of the other table, *if any*, as shown in Figure 3.5.

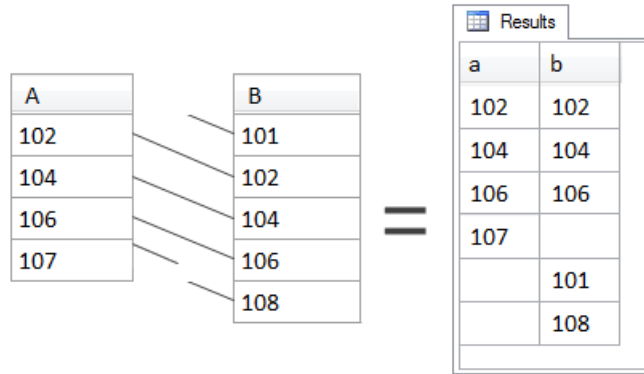


Figure 3.5. A FULL OUTER JOIN B

The full outer join is a *combination* of left and right outer joins. (More technically, if you remember your set theory from mathematics at school, it's the union of the results from the left and right outer joins.) Matching rows are—of course—included, but rows that have no match from either table, are also included.



The Difference between Inner and Outer Joins

The results of an outer join will *always* equal the results of the corresponding inner join between the two tables *plus* some unmatched rows from either the left table, the right table, or both—depending on whether it is a left, right, or full outer join, respectively.

Thus the difference between a left outer join and a right outer join is simply the difference between whether the left table's rows are all returned, with or without matching rows from the right table, or whether the right table's rows are all returned, with or without matching rows from the left table.

A full outer join, meanwhile, will always include the results from both left and right outer joins.

The Cross Join

For a **cross join**, every row from both tables is returned, joined to every row of the other table, regardless of whether they match. The distinctive feature of a cross join is that it has *no* ON clause—as you can see in the following query:

```
SELECT
  a, b
FROM
  A CROSS JOIN B
```

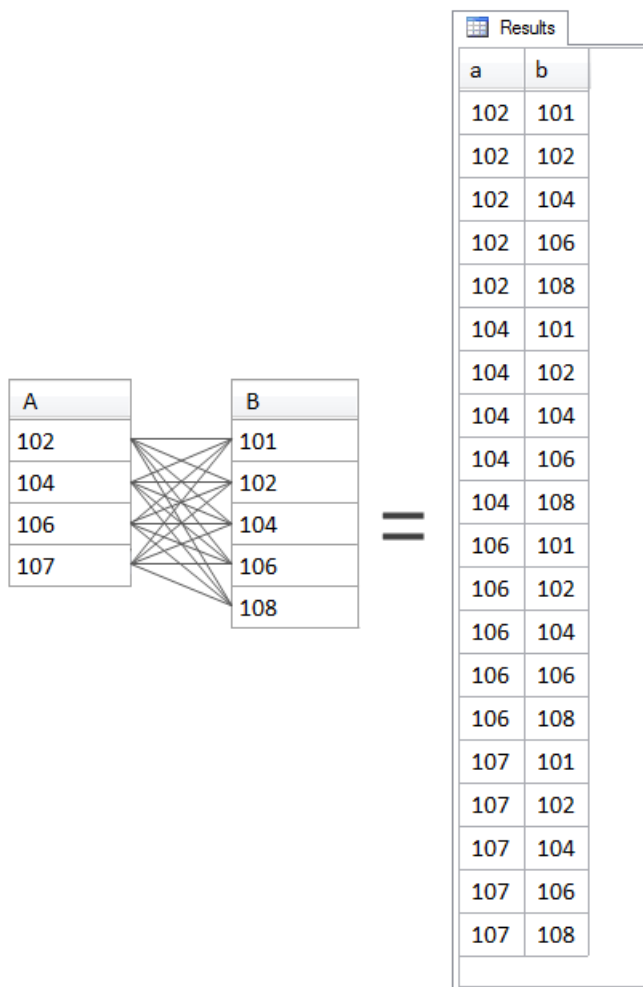


Figure 3.6. A CROSS JOIN B

Cross joins can be very useful but are exceedingly rare. Their purpose is to produce a tabular structure containing rows which rep all possible combinations of two sets of values (in our example, columns from two tables) as shown in Figure 3.6; this can be useful in generating test data or looking for missing values.



Old-Style Joins

There's another type of join, which has a comma-separated list of tables in the FROM clause, with the necessary join conditions in the WHERE clause; this type of join is sometimes called the "old-style" join, or "comma list" join, or "WHERE clause" join. For example, for the A and B tables, it would look like this:

```
SELECT
  a, b
FROM
  A, B
WHERE
  a=b
```

These old-style joins can only ever be inner joins; the other join types are only possible with very proprietary and confusing syntax, which the database system vendors themselves caution is deprecated. Compare this with the recommended syntax for an INNER JOIN:

```
SELECT
  a, b
FROM
  A INNER JOIN B
    ON a=b
```

You may see these old-style joins *in the wild* but I'd caution you against writing them yourself. *Always* use JOIN syntax.

To recap our quick survey of joins, there are three basic types of join and a total of five different variations:

- inner join
- left outer join, right outer join, and full outer join
- cross join

Now for some more realistic examples.

Real World Joins

Chapter 2 introduced the Content Management System `entries` table, which we'll continue to use in the following queries to demonstrate how to write joins. Figure 3.7 shows some—but not all—of its contents. The `content` column, for example, is missing.

entries		
category	title	created
angst	What If I Get Sick and Die?	2008-12-30
humor	Uncle Karl and the Gasoline	2009-02-28
advice	Be Nice to Everybody	2009-03-02
humor	Hello Statue	2009-03-17
science	The Size of Our Galaxy	2009-04-03

Figure 3.7. The `entries` table

Within our CMS web site, the aim is to give each category its own area on the site, linked from the site's main menu and front page. The science area will contain all the entries in the science category, the humor area will contain all the entries in the humor category, and so on, as shown in Figure 3.8. To this end, each entry is given a category, stored in the `category` column of each row.

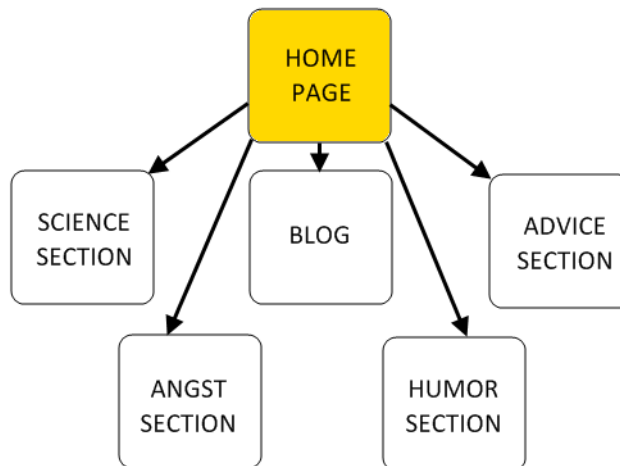


Figure 3.8. A suggested CMS site structure

The main category pages themselves would need more than just the one word category name that we see in the `entries` table. Site visitors will want to understand what each section is about, so we'll need a more descriptive name for each category. But where to store this in the site? We could *hardcode* the longer name directly into each main section page of the web site. A better solution, however, would be to save the names in the database. Another table will do the job nicely, and so we create the `categories` table for this purpose; we'll give it two columns—category and name—as shown in Figure 3.9.

categories	
category	name
advice	Gentle Words of Advice
angst	Stories from the Id
blog	Log On to My Blog
humor	Humorous Anecdotes
science	Our Spectacular Universe

Figure 3.9. The categories table

The category column is the **key** to each row in the `categories` table. It's called a key because the values in this column are unique, and are used to identify each row. This is the column that we'll use to join to the `entries` table. We'll learn more about designing tables with keys in Chapter 10. Right now, let's explore the different ways to join the `categories` and `entries` tables.



Creating the Categories Table

The script to create the `categories` table can be found in Appendix C and in the download for the book in a file called `CMS_05_Categories_INNER_JOIN_Entries.sql`.

Inner Join: Categories and Entries

The first join type we'll look at is an inner join:

CMS_05_Categories_INNER_JOIN_Entries.sql (excerpt)

```
SELECT
    categories.name, entries.title, entries.created
FROM
    categories
    INNER JOIN entries
        ON entries.category = categories.category
```

Figure 3.10 shows the results of this query.



The screenshot shows a window titled 'Results' containing a table with three columns: 'name', 'title', and 'created'. The table contains five rows of data, representing the results of an inner join between the 'categories' and 'entries' tables. The 'name' column lists category names, the 'title' column lists entry titles, and the 'created' column lists the creation dates. The data is as follows:

name	title	created
Gentle Words of Advice	Be Nice to Everybody	2009-03-02
Stories from the Id	What If I Get Sick and Die?	2008-12-30
Humorous Anecdotes	Uncle Karl and the Gasoline	2009-02-28
Humorous Anecdotes	Hello Statue	2009-03-17
Our Spectacular Universe	The Size of Our Galaxy	2009-04-03

Figure 3.10. The results of the inner join

Let's walk through the query clause by clause and examine what it's doing, while comparing the query to the results it produces. The first part of the query to look at, of course, is the **FROM** clause:

```
FROM
    categories
    INNER JOIN entries
        ON entries.category = categories.category
```

The **categories** table is joined to the **entries** table using the keywords **INNER JOIN**. The **ON** clause specifies the join condition, which dictates how the rows of the two tables must match in order to participate in the join. It uses **dot notation** (*table-name.rowname*) to specify that rows of the **categories** table will match rows of

the `entries` table only when the values in their `category` columns are equal. We'll look in more detail at dot notation later in this chapter.

Figure 3.11 shows in detail how the result set of the query is produced by the inner join of the `categories` table to the `entries` table. Because it's an inner join, each of the rows of the `categories` table is joined only to the rows of the `entries` table that have matching values in their respective `category` columns.

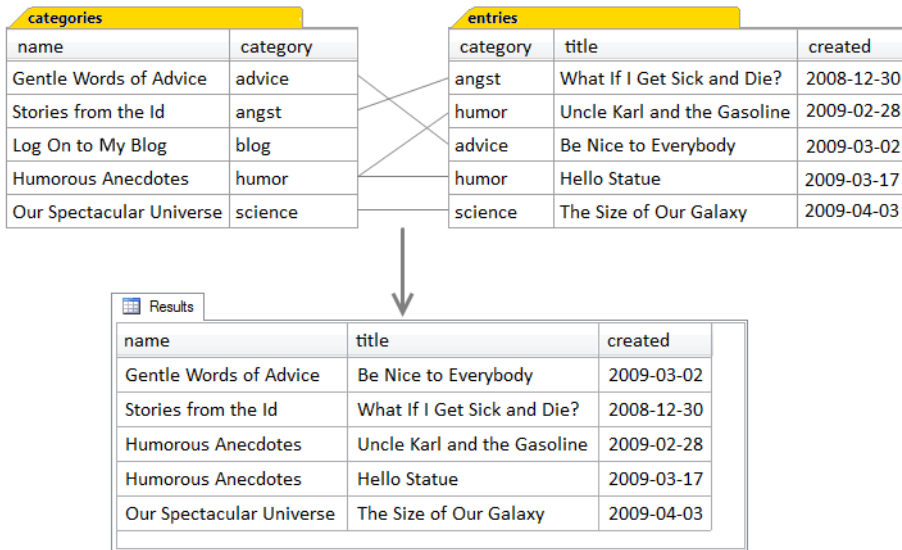


Figure 3.11. The inner join in detail



Some of the Entries Table is Hidden

The `entries` table actually has several additional columns that are not shown: `id`, `updated`, and `content`. These columns are also available, but were omitted to keep the diagram simple. In fact, the diagram would've been quite messy if the `content` column had been included, as it contains multiple lines of text. Since these columns were not mentioned in the query at all, including them in the diagram might have made it confusing. Some readers would surely ask, "Hey, where did these come from?"

Regarding the matching of rows of the `categories` and `entries` tables, notice that:

- The `categories` row for `humor` matched two `entries` rows, and both instances of matched rows are in the results, with the name of the `humor` category appearing twice.
- The `categories` row for `blog` matched no `entries` rows. Consequently, as this is an inner join, this category does not appear in the results.
- The other `categories` rows matched one `entries` row each, and these matched rows are in the result.

Stating these observations in a slightly different way, we can see that a single row in the `categories` table can match no rows, one row, or more than one row in the `entries` table.



One-to-Many Relationships

The *more than one* aspect of the relationship between a row in the `categories` table and matching rows in the `entries` table is the fundamental characteristic of what we call a **one-to-many relationship**. Each (one) category can have multiple (many) entries.

Even though a given category (`blog`) might have no matching entries, and only one of the categories (`humor`) has more than one entry, the relationship between the `categories` and `entries` tables is still a one-to-many relationship in structure. Once the tables are fully populated with live data, it's likely that all categories will have many entries.

Looking at this relationship from the other direction, as it were, we can see that each entry can belong to only one category. This is a direct result of the category column in the `entries` table having only one value, which can match only one category value in the `categories` table. Yet more than one entry can match the same category, as we saw with the `humor` entries. So a one-to-many relationship is also a many-to-one relationship. It just depends on the *direction* of the relationship being discussed.

Now we've examined the `FROM` clause and seen how the `INNER JOIN` and its `ON` condition have specified how the tables are to be joined, we can look at the `SELECT` clause:

```
SELECT
  categories.name
, entries.title
, entries.created
```

As you would expect, the `SELECT` clause simply specifies which columns from the result of the inner join are to be included in the result set.



Leading Commas

Notice that the `SELECT` clause has now been written with one line per column, using a convention called *leading commas*; this places the commas used to separate the second and subsequent items in a list at the *front* of their line. This may look unusual at first, but the syntax is perfectly okay; remember, new lines and white space are ignored by SQL just as they are by HTML. Experienced developers may be more used to having trailing commas at the end of the lines, like this:

```
SELECT
  categories.name,
  entries.title,
  entries.created
```

I use leading commas as a coding style convention to make SQL queries more readable and maintainable. The importance of readability and maintainability can't be overstated. For example, see if you can spot the two coding errors in this hypothetical query:

```
SELECT
  first_name,
  last_name,
  title
  position,
  staff_id,
  group,
  region,
FROM
  staff
```

Now see if you can spot the coding errors here:

```
SELECT
    first_name
  , last_name
  , title
    position
  , pay_scale
  , group
  , region
  ,
FROM
    staff
```

The query is missing a comma in the middle of the column list and has an unneeded, additional comma at the end of the list. In which example were the errors easier to spot?

In addition, leading commas are easier to handle if you edit your SQL in a text editor with the keyboard. Sometimes you need to move or delete a column from the **SELECT** clause, and it's easier to select (highlight) the single line with the keyboard's Shift and Arrow keys. Similarly, removing the last column requires also removing the trailing comma from the previous line, which is easy to forget. A *dangling* comma in front of the **FROM** keyword is a common error that's difficult to make using leading commas.

All Columns Are Available after a Join

In any join, all columns of the tables being joined are available to the **SELECT** query, even if they're not used by the query. Let's look at our inner join again:

```
SELECT
    categories.name
  , entries.title
  , entries.created
FROM
    categories
    INNER JOIN entries
      ON entries.category = categories.category
```

In most join queries, tables being joined usually contain more columns than those mentioned in the **SELECT** clause. This is true here too; the **entries** table has other

columns not mentioned in the query. We haven't included them in Figure 3.11 just to keep the figure simple. Although the figure is correct, it could be construed as slightly misleading, because it shows only the result set of the query, rather than the tabular structure produced by the inner join.

Figure 3.12 expands on the actual processing of the query and shows the tabular structure that's produced by the FROM clause and the inner join; it includes the two category columns—one from each table. This tabular structure, the intermediate table, is produced by the database system as it performs the join, and held temporarily for the SELECT clause.

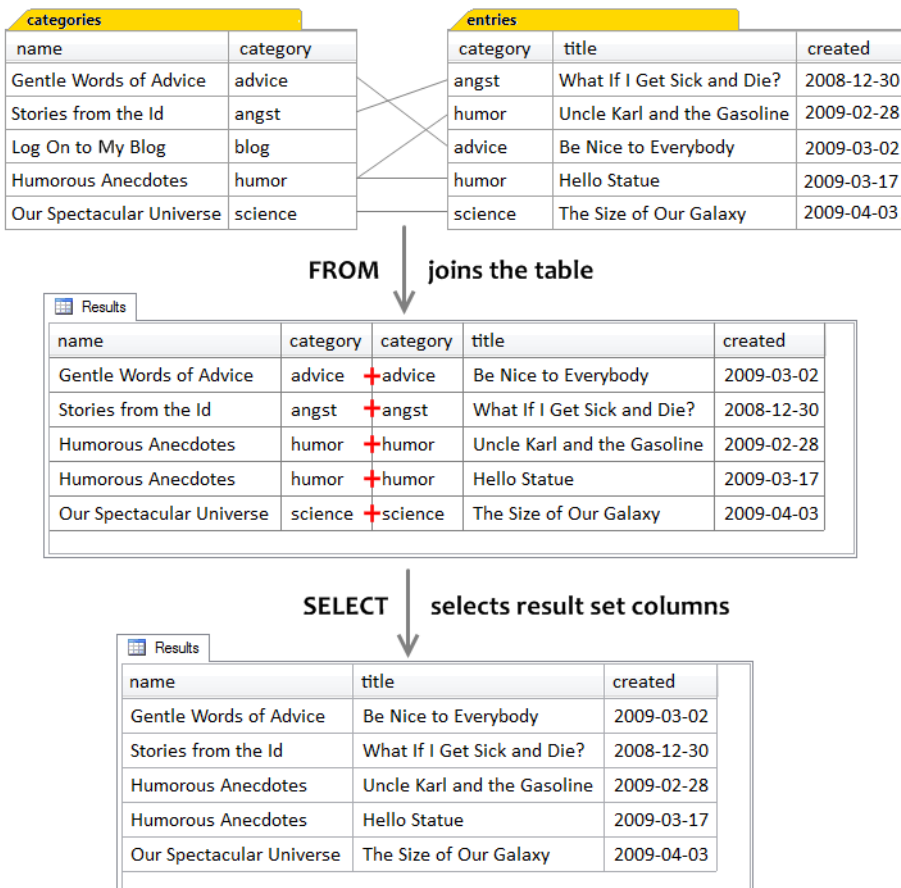


Figure 3.12. How an inner join is actually processed

When a Join is Executed in a Query

Two important points come out of the analysis of our first example join query:

- A join produces an intermediate tabular result set;
- The **SELECT** clause occurs *after* the **FROM** clause and operates on the intermediate result set.

At the beginning of this chapter, I mentioned that the **FROM** clause is the first clause that the database system parses when we submit a query. If there are no syntax errors, the database system goes ahead and executes the query. Well, it turns out that the **FROM** clause is the first clause that the database system executes, too.

You could consider the execution of a join query as working in the following manner. First, the database system produces an intermediate tabular result set based on the join specified in the **FROM** clause. This contains all the columns from both tables. Then the database system uses the **SELECT** clause to select only the specified columns from this intermediate result set, and extracts them into the final tabular structure that is returned as the result of the query.

Qualifying Column Names

Finally, let's take one more look at our inner join query:

CMS_05_Categories_INNER_JOIN_Entries.sql (excerpt)

```
SELECT
  categories.name
, entries.title
, entries.created
FROM
  categories
  INNER JOIN entries
    ON entries.category = categories.category
```

Each of the column names used in this query is **qualified** by its table name, using **dot notation**, where the table name precedes the column name with a dot between them.

Qualifying column names is mandatory when there is more than one instance of the same column name in a query. (These would be from different tables, of course; more than one instance of the same column name in a single table is not possible,

as all columns within a table must each have unique names.) If you don't uniquely identify each of the columns that have the same name but are in different tables, you will receive a syntax error about ambiguous names. This applies whether the query makes reference to both columns or not; every single reference must be qualified.

When there is only one instance of the column name in the query, then qualifying column names becomes optional. Thus, we could have written the following and be returned the same result set:

```
SELECT
    name
  , title
  , created
FROM
    categories
    INNER JOIN entries
      ON entries.category = categories.category
```

However, it's a good idea to qualify all column names in this situation because when you look at the `SELECT` clause, you can't always tell which table each column comes from. This can be especially frustrating if you're only remotely familiar with the tables involved in the query, such as when you're troubleshooting a query written by another person (or even by yourself, a few months ago).



Always Qualify Every Column in a Join Query

Even though some or even all columns may not need to be qualified within a join query, qualifying every column in a multi-table query is part of good SQL coding style, because it makes the query easier for us to understand.

In a way, qualifying column names makes the query **self-documenting**: it makes it obvious what the query is doing so that it's easier to explain in documentation.

Table Aliases

Another way to qualify column names is by using **table aliases**. A table alias is an alternate name assigned to a table in the query. In practice, a table alias is often shorter than the table name. For example, here's the same inner join using table aliases:

```
SELECT
  cat.name
, ent.title
, ent.created
FROM
  categories AS cat
  INNER JOIN entries AS ent
    ON ent.category = cat.category
```

Here, the `categories` table has been assigned the alias `cat`, and the `entries` table has been assigned the alias `ent`. You're free to choose any names you wish; the table aliases are temporary, and are valid only for the duration of the query. Some people like to use single letters as table aliases when possible, because it reduces the number of characters in the query and so makes it easier to read.

The only caveat in using table aliases is that once you have assigned an alias to a table, you can no longer use the table name to qualify its columns *in that query*; you must use the alias name consistently throughout the query. Once the query is complete however, you're free to refer to the original table by its full name again, the same alias, or even a different alias; the point here being that a table alias is defined only for the duration of the query that contains it.

Left Outer Join: Categories and Entries

Continuing our look at join queries, the left outer join query we'll examine is exactly the same as the inner join query we just covered, except that it uses `LEFT OUTER JOIN` as the join keywords:

CMS_06_Categories_LEFT_OUTER_JOIN_Entries.sql (excerpt)

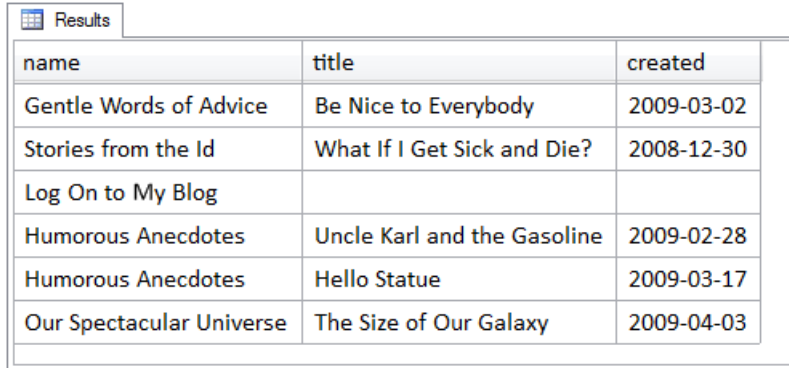
```
SELECT
  categories.name
, entries.title
, entries.created
```

```

FROM
  categories
LEFT OUTER JOIN entries
  ON entries.category = categories.category

```

Figure 3.13 shows the results of the above query.



name	title	created
Gentle Words of Advice	Be Nice to Everybody	2009-03-02
Stories from the Id	What If I Get Sick and Die?	2008-12-30
Log On to My Blog		
Humorous Anecdotes	Uncle Karl and the Gasoline	2009-02-28
Humorous Anecdotes	Hello Statue	2009-03-17
Our Spectacular Universe	The Size of Our Galaxy	2009-04-03

Figure 3.13. The results of the left outer join query

The only difference between this left outer join query and the preceding inner join query is the inclusion of one additional row—for the category with the name `Log On to My Blog`—in the result set. The additional row is included because the query uses an outer join. Specifically, it's a left outer join, and therefore all of the rows of the left table, the `categories` table, must be included in the results. The left table, you may recall, is simply the table that is mentioned to the left of the `LEFT OUTER JOIN` keywords. Figure 3.14 shows the process of the join and selection in more detail.

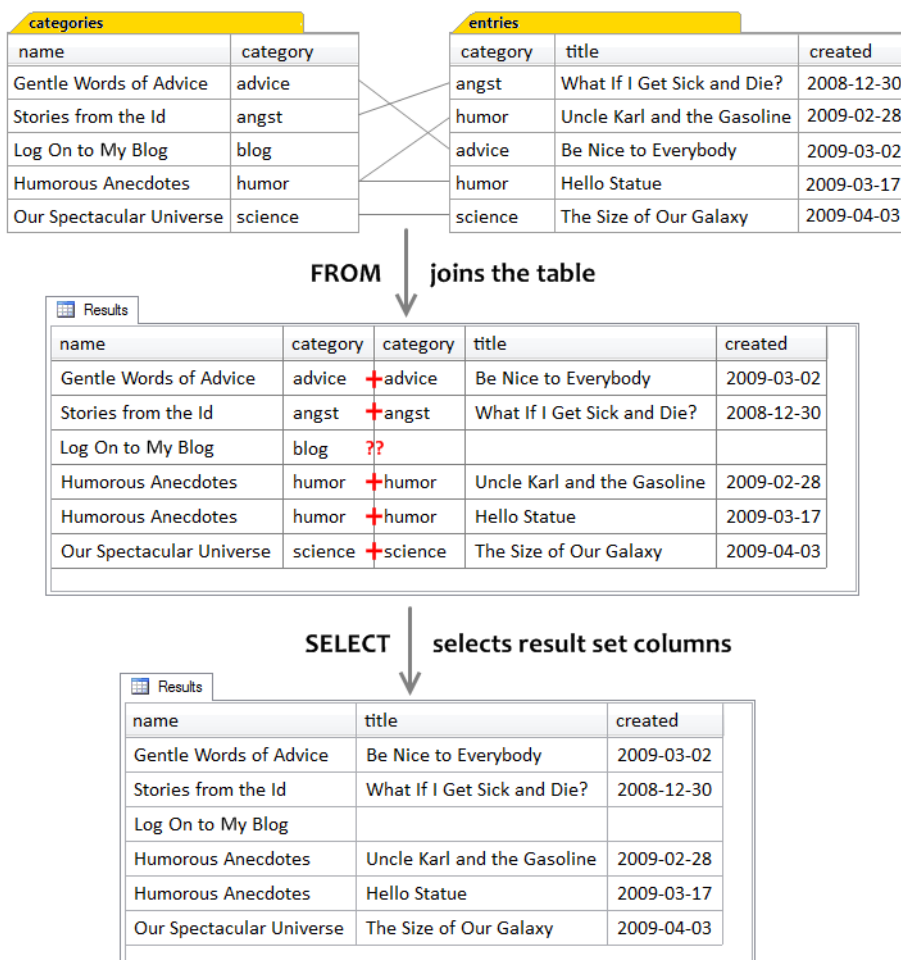


Figure 3.14. How a left outer join query is actually processed

To make it more obvious which table is the left one and which table is the right one, we could write the join without line breaks and spacing so `categories` is more obviously the left table in this join:

```
FROM categories LEFT OUTER JOIN entries
```

Let's take another look at the results of our left outer join, because there is one more important characteristic of outer joins that I need to point out.



An Application for Left Outer Joins: a Sitemap

Looking at the results of our `LEFT OUTER JOIN` query, it's easy enough to see how they could form the basis of a sitemap for the CMS. For example, the HTML for the sitemap that can be produced by these query results might be:

```
<h2>Gentle Words of Advice</h2>
<ul>
  <li>Be Nice to Everybody (2009-03-02)</li>
</ul>

<h2>Stories from the Id</h2>
<ul>
  <li>What If I Get Sick and Die? (2008-12-30)</li>
</ul>

<h2>Log On to My Blog</h2>

<h2>Humorous Anecdotes</h2>
<ul>
  <li>Hello Statue (2009-03-17)</li>
  <li>Uncle Karl and the Gasoline (2009-02-28)</li>
</ul>

<h2>Our Spectacular Universe</h2>
<ul>
  <li>The Size of Our Galaxy (2009-04-03)</li>
</ul>
```

If you're an experienced web developer, you can probably see how you'd make the transformation from query results to HTML using your particular application language.

Notice that the Log On to My Blog category has no entries, but is included in the result (because it's a left outer join). Therefore, the application logic needs to detect this situation, and not produce the unordered list (``) tags for entries in that category. Without going into the details of application programming logic, let me just say that it's done by detecting the `NULL`s in the entries columns of that result row.

Outer Joins Produce NULLs

Our left outer join includes rows from the left table that have no match in the right table, as shown in Figure 3.13. So what exactly are the values in the `title` and `created` columns of the `blog` category result row? Remember, these columns come from the `entries` table.

The answer is: they are **NULL**.

NULL is a special value in SQL, which stands for the *absence* of a value. In a left outer join, columns that come from the right table for unmatched rows from the left table are NULL in the result set. This literally means that there is no value there, which makes sense because there is no matching row from the right table for that particular row of the left table.

Working with NULLs is part of daily life when it comes to working with databases. We first came across NULL (albeit briefly) in Chapter 1, where it was used in a sample `CREATE TABLE` statement and we'll see NULL again throughout the book.

Right Outer Join: Entries and Categories

The following right outer join query produces exactly the same results as the left join query we just covered:

CMS_06_Categories_LEFT_OUTER_JOIN_Entries.sql (excerpt)

```
SELECT
    categories.name
  , entries.title
  , entries.created
FROM
    entries
    RIGHT OUTER JOIN categories
      ON entries.category = categories.category
```

But how can this be?

Hopefully you've spotted the answer: I've switched the order of the tables! In the right outer join query, I wrote:

```
FROM entries RIGHT OUTER JOIN categories
```


In the preceding left outer join query, I had:

```
FROM categories LEFT OUTER JOIN entries
```

The lesson to be learned from this deviousness is simply that left and right outer joins are completely equivalent, it's just a matter of which table is the **outer table**: the one which will have all of its rows included in the result set. Because of this, many practitioners avoid writing right outer queries, converting them to left outer joins instead by changing the order of the tables; that way the table from which all rows are to be returned is always on the left. Left outer joins seem to be much easier to understand than right outer joins for most people.

Right Outer Join: Categories and Entries

What if I hadn't switched the order of the tables in the preceding right outer join? Suppose the query had been:

CMS_07_Categories_RIGHT_OUTER_JOIN_Entries.sql (excerpt)

```
SELECT
    categories.name
,   entries.title
,   entries.created
FROM
    categories
    RIGHT OUTER JOIN entries
        ON entries.category = categories.category
```

This time, as in our first left outer join, the `categories` table is on the left, and the `entries` table is on the right. Figure 3.15 shows the results of this query are the same as the results from our earlier inner join.

Results		
name	title	created
Stories from the Id	What If I Get Sick and Die?	2008-12-30
Humorous Anecdotes	Uncle Karl and the Gasoline	2009-02-28
Gentle Words of Advice	Be Nice to Everybody	2009-03-02
Humorous Anecdotes	Hello Statue	2009-03-17
Our Spectacular Universe	The Size of Our Galaxy	2009-04-03

Figure 3.15. The results of the right outer join query

How can this be? Is this more deviousness? No, not this time; the reason is because it's the actual contents of the tables. Remember, a right outer join returns all rows of the right table, with or without matching rows from the left table. The `entries` table is the right table, but in this particular instance, every entry has a matching category. All the entries are returned, and there are no unmatched rows.

So it wasn't really devious to show that the right outer join produces the same results as the inner join, because it emphasized the rule for outer joins that all rows from the outer table are returned, with or without matching rows, *if any*. In this case, there weren't any.

To really see the right outer join in action, we'd need an entry that lacks a matching category. Let's add an entry to the `entries` table, for a new category called `computers`, as shown in Figure 3.16.

entries		
category	title	created
angst	What If I Get Sick and Die?	2008-12-30
humor	Uncle Karl and the Gasoline	2009-02-28
advice	Be Nice to Everybody	2009-03-02
humor	Hello Statue	2009-03-17
science	The Size of Our Galaxy	2009-04-03
computers	Windows Media Center Rocks	2009-04-29

Figure 3.16. A new addition to the `entries` table



Trying Out Your SQL

The `INSERT` statement that adds this extra row to the `entries` table can be found in the section called “Content Management System” in Appendix C.

Figure 3.17 shows that when we re-run the right outer join query with the new category, the results are as expected.

Results		
name	title	created
Stories from the Id	What If I Get Sick and Die?	2008-12-30
Humorous Anecdotes	Uncle Karl and the Gasoline	2009-02-28
Gentle Words of Advice	Be Nice to Everybody	2009-03-02
Humorous Anecdotes	Hello Statue	2009-03-17
Our Spectacular Universe	The Size of Our Galaxy	2009-04-03
	Windows Media Center Rocks	2009-04-29

Figure 3.17. The results of the right outer join query—take two

This time, we see the unmatched entry in the query results, because there’s no row in the `categories` table for the `computers` category.

Full Outer Join: Categories and Entries

Our next example join query is the full outer join. The full outer join query syntax, as I’m sure you can predict, is remarkably similar to the other join types we’ve seen so far:

`CMS_08_Categories_FULL_OUTER_JOIN_Entries.sql` (excerpt)

```
SELECT
    categories.name
,   entries.title
,   entries.created
FROM
    categories
    FULL OUTER JOIN entries
        ON entries.category = categories.category
```

This time, the join keywords are `FULL OUTER JOIN`, but an unfortunate error happens in at least one common database system. In MySQL, which doesn't support `FULL OUTER JOIN` despite it being standard SQL, the result is a syntax error: `SQL Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'OUTER JOIN entries ON ...'`

Figure 3.18 shows the result in other database systems that do support `FULL OUTER JOIN`.



The screenshot shows a window titled 'Results' containing a table with three columns: 'name', 'title', and 'created'. The table contains eight rows of data, representing the result of a full outer join between two tables. The first two rows are from the left table, the next two are from the right table, and the last four are the joined rows.

name	title	created
Gentle Words of Advice	Be Nice to Everybody	2009-03-02
Stories from the Id	What If I Get Sick and Die?	2008-12-30
Log On to My Blog		
Humorous Anecdotes	Uncle Karl and the Gasoline	2009-02-28
Humorous Anecdotes	Hello Statue	2009-03-17
Our Spectacular Universe	The Size of Our Galaxy	2009-04-03
	Windows Media Center Rocks	2009-04-29

Figure 3.18. The results of the full outer join query

Notice that the result set includes unmatched rows from both the left and the right tables. This is the distinguishing feature of full outer joins that we saw earlier; both tables are outer tables, so unmatched rows from both are included. It's for this reason that full outer joins are rare in web development as there are few situations that call for them. In contrast, inner joins and left outer joins are quite common.

UNION Queries

If your database system does not support the `FULL OUTER JOIN` syntax, the same results can be obtained by a slightly more complex query, called a **union**. Union queries are not joins per se. However, most people think of the results produced by a union query as consisting of two results sets concatenated or appended together. `UNION` queries perform a join only in a very loose sense of the word.

Let's have a look at a union query:

CMS_09_Left_outer_join_UNION_right_outer_join.sql (excerpt)

```
SELECT
    categories.name
, entries.title
, entries.created
FROM
    categories
    LEFT OUTER JOIN entries
        ON entries.category = categories.category
UNION
SELECT
    categories.name
, entries.title
, entries.created
FROM
    categories
    RIGHT OUTER JOIN entries
        ON entries.category = categories.category
```

As you can see, the left outer join and right outer join queries we saw earlier in this chapter have simply been concatenated together using the **UNION** keyword. A union query consists of a number of **SELECT** statements combined with the **UNION** operator. They're called **subselects** in this context because they're *subordinate* to the whole **UNION** query; they're only *part* of the query, rather than being a query executed on its own. Sometimes they're also called subqueries, although this term is generally used for a more specific situation, which we shall meet shortly.

When executed, a **UNION** operation simply combines the result sets produced by each of its subselect queries into a single result set. Figure 3.19 shows how this works for the example above:

I mentioned earlier that a join operation can best be imagined as actually concatenating a row from one table onto the end of a row from the other table—a horizontal concatenation, if you will. The union operation is therefore like a vertical concatenation—a second result set is appended onto the end of the first result set.

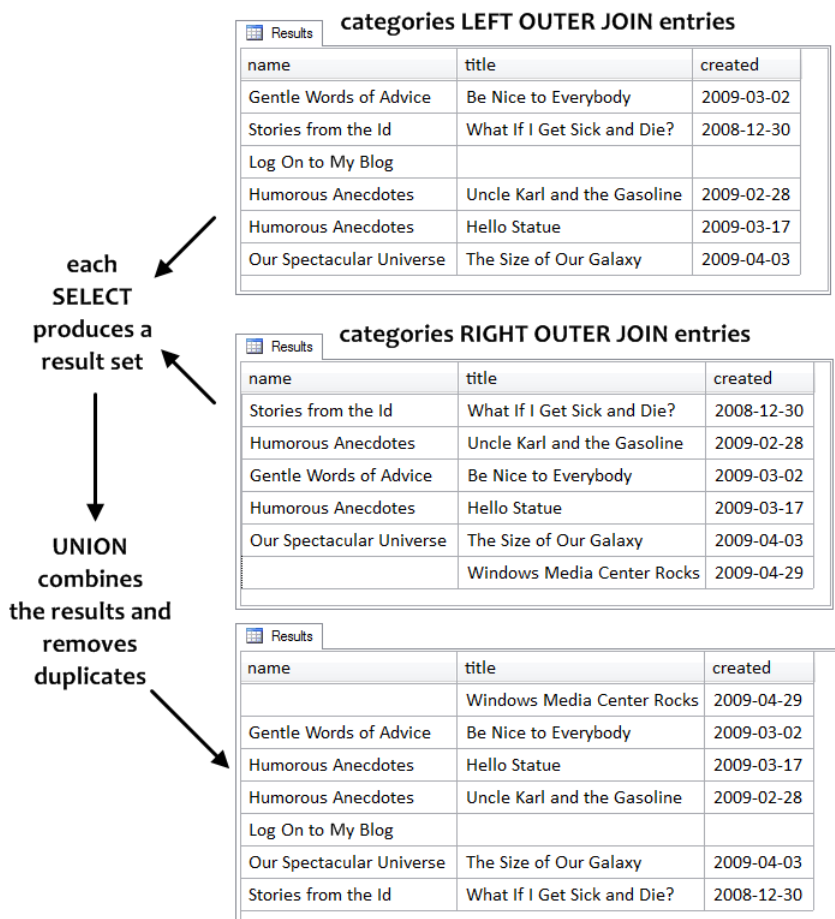


Figure 3.19. How a union query works

The interesting feature is that duplicates are removed. You can see the duplicates easily enough—they are entire rows in which every column value is identical. The reason that duplicates are produced in this example is due to both of the sub-selects—the left outer join and the right outer join—returning rows from the same two tables which match the the same join conditions. Thus, matched rows are returned by both subselects, creating duplicate rows in the intermediate results. Only the unmatched rows are not duplicated.

You might wonder why UNION removes duplicates; the answer is simply that it's designed that way. It's how the UNION operator is supposed to work.



UNION and UNION ALL

Sometimes it's important to retain all rows produced by a union operation, and not have the duplicate rows removed. This can be accomplished by using the keywords `UNION ALL` instead of `UNION`.

- `UNION` removes duplicate rows. Only one row from each set of duplicate rows is included in the result set.
- `UNION ALL` retains all rows produced by the subselects of the union, maintaining duplicate rows.

`UNION ALL` is significantly faster because the need to search for duplicate rows—in order to remove them—is redundant.

The fact that our union query removed the duplicate rows means that the above union query produces the same results as the full outer join. Of course, this example was contrived to do just that.

There is more to be said about union queries, but for now, let's finish this section with one point: union queries, like join queries, produce a *tabular structure* as their result set.

Views

A **view** is another type of database object that we can create, like a table. Views are insubstantial, though, because they don't actually store data (unlike tables). Views are `SELECT` statements (often complex ones) that have been given a name for ease of reference and reuse, and can be used for many purposes:

- They can *customize* a `SELECT` statement, by providing column aliases.
- They can be an alias to the result set produced by the `SELECT` statement in their definition. If the `SELECT` statement in the view contains joins between a number of tables, they are effectively *pre-joined* by the database in advance of a query against the view. All this second query then sees is a single table to query against. This is probably the most important benefit of using views.
- They can enforce security on the database. Users of a database might be restricted from looking at the underlying tables altogether; instead, they might only be granted access to views. The classic example is the `employees` table, which

contains columns like `name`, `department`, and `salary`. Because of the confidential nature of `salary`, very few people are granted permission to access such a table directly; rather, a special view is made available that excludes the confidential columns.

To demonstrate, here's how you define the inner join query used earlier as a view:

CMS_10_CREATE_VIEW.sql (excerpt)

```
CREATE VIEW
  entries_with_category
AS
SELECT
  entries.title
, entries.created
, categories.name AS category_name
FROM
  entries
  INNER JOIN categories
    ON categories.category = entries.category
```

This statement defines a view called `entries_with_category`. It uses the `AS` keyword to associate the name `entries_with_category` with the `SELECT` statement which defines the view. With the view defined, we can query it as if it were a table:

CMS_10_CREATE_VIEW.sql (excerpt)

```
SELECT
  title
, category_name
FROM
  entries_with_category
```

Of course, it's not a table—the view itself does not actually store the result set produced by its `SELECT` statement. The use of the view name here works by executing the view's underlying `SELECT` statement, storing its results in an intermediate table, and using that table as the result of the `FROM` clause. The results of the above query, shown in Figure 3.20, are quite familiar.

Results	
title	category_name
What If I Get Sick and Die?	Stories from the Id
Uncle Karl and the Gasoline	Humorous Anecdotes
Be Nice to Everybody	Gentle Words of Advice
Hello Statue	Humorous Anecdotes
The Size of Our Galaxy	Our Spectacular Universe

Figure 3.20. Selecting from a view

This result set is similar to the result set produced by the inner join query which defines the view. Notice that only two columns have been returned, because the `SELECT` statement which uses the view in its `FROM` clause (as opposed to the `SELECT` statement which defines the view) only asked for two. Also, notice that a column alias called `category_name` was assigned to the `categories` table's `name` column in the view definition; this is the column name that must be used in any `SELECT` statement which uses the view, and it's the column name used in the result set.

One particular implication of the view definition is that *only* the columns defined in the view's `SELECT` statement are available to any query that uses the view. Even though the `entries` table has a `content` column, this column is unknown to the view and will generate a syntax error if referenced in a query using the view.

Views in Web Development

How do views relate to our day-to-day tasks as web developers?

- When working on a large project in a team environment, you may be granted access to views only, not the underlying tables. For example, a Database Administrator (DBA) may have built the database, and you're just using it. You might not even be aware that you're using views. This is because, syntactically, both tables and views are used in the `FROM` clause in exactly the same way.
- When you build your own database, you may wish to create views for the sake of convenience. For example, if you often need to display a list of entries and their category on different pages within the site, it's a lot easier to write `FROM entries_with_category` than the underlying join.

Subqueries and Derived Tables

We started this chapter by examining the `FROM` clause, working our way up from simple tables through the various types of joins. We briefly saw a `UNION` query and its subselects, and we've also seen how views make complex join expressions easier to use. To finish this chapter, we'll take a quick look at **derived tables**. Here's an example:

CMS_11_Derived_tables.sql (excerpt)

```
SELECT
  title
, category_name
FROM
  ( SELECT
    entries.title
  , entries.created
  , categories.name AS category_name
  FROM
    entries
    INNER JOIN categories
      ON categories.category = entries.category
  ) AS entries_with_category
```

The **derived table** here is the entire `SELECT` query in parentheses (the parentheses are required in the syntax, to delimit the enclosed query). A derived table is a common type of **subquery**, which is a query that's subordinate to—or nested within—another query (much like the subselects in the union query).

It looks familiar, too, doesn't it? This subquery is the same query used in the `entries_with_categories` view defined in the previous section. Indeed, just as every view needs a name, every derived table must be also given a name, also using the `AS` keyword (on the last line) to assign the name `entries_with_category` as a table alias for the derived table. With these similarities in mind, derived tables are often also called **inline views**. That is, they define a tabular structure—the result set produced by the subquery—directly inline in (or within) the SQL statement, and the tabular structure produced by the subquery, in turn, is used as the source of the data for the `FROM` clause of outer or main query.

In short, anything which produces a tabular structure can be specified as a source of data in the `FROM` clause. Even a `UNION` query, which we discussed briefly, can also

be used in the `FROM` clause, if it's specified as a derived table; the entire `UNION` query would go into the parentheses that delimit the derived table.

Derived tables are incredibly useful in SQL. We'll see several of them throughout the book.

Wrapping Up: the FROM Clause

In this chapter, we examined the `FROM` clause, and how it specifies the source of the data for the `SELECT` statement. There are many different types of tabular structures that can be specified in the `FROM` clause:

- single tables
- joined tables
- views
- subqueries or derived tables

Finally—and this is one of the key concepts in the book—not only does the `FROM` clause specify one or more tabular structures from which to extract data, but the result of the execution of the `FROM` clause is also another tabular structure, referred to as the *intermediate result set* or *intermediate table*. In general, this intermediate table is produced first, before the `SELECT` clause is processed by the database system.

In the Chapter 4, we'll see how the `WHERE` clause can be used to filter the tabular structure produced by the `FROM` clause.

What's Next?

If you've enjoyed these chapters from *Simply SQL*, order yourself a copy today!

Simply SQL is a practical, step-by-step guide that'll show you how to make the most of your data using best-practice SQL code. Rather than bore you with theory, it focuses on the practical use of SQL with common databases and uses plenty of diagrams, easy-to-read text, and examples to help make learning SQL fun. You'll also gain access to the code archive download, so you can try out all the examples without retyping!

In the remaining chapters, you'll learn:

- ❑ standard SQL syntax
- ❑ best-practice database design techniques
- ❑ advanced SQL syntax like joins, groups, and subqueries
- ❑ all about SQL data types
- ❑ and a whole lot more ...

[Order the complete printed edition now!](#)



Index

A

ADD CONSTRAINT keyword, 228

aggregate, 109

aggregate function, 30

aggregate functions, 109, 120, 140, 141–149

 COUNT (*), 146–149

 COUNT (DISTINCT), 144–146

 ignore NULLs, 144

 without GROUP BY, 142–144

aggregate functions and GROUP BY clause, 117

aggregate row, 109

aggregation, 29

ALTER statement, 7, 10

AND and OR

 combining, 89–90

 compound conditions with, 86

 use parentheses when mixing, 90

ASC (keyword), 32

ASC and ORDER BY clause, 164–166

attributes (data modelling), 213–219

autonumbers (surrogate key), 234–235

B

BETWEEN operators, 83–86

BIGINT (data type), 185, 186, 193

Binary Large Objects (BLOBs), 117

BLOB column, 198–199

C

cardinality, 215

cart_id, 105

cartitems table, 105, 283

carts table, 282

carts.id, 109, 163

CASE function, 152–154

case-sensitivity, 9

CAST function, 153

categories table, 47, 270–271

 full outer join, 63–67

 inner join, 48–56

 left outer join, 56–61

 one-to-many relationships, 50

 right outer join, 61–63

category column, 47, 242

CHAR column, 194–195

CHAR_LENGTH, 153

CHARACTER column, 194

character data types, 194–200

 CHAR, 194–195

 CLOB and BLOB, 198–199

 NCHAR and NVARCHAR, 198

 numeric or character, 196–198

 string functions, 199–200

 VARCHAR, 195–196

Character Large Objects (CLOBs), 117

CHECK constraints, 209–210

clauses, 4, 19–20

CLOB column, 198–199

COALESCE function, 151–152

code highlighting, 255

coding style, 4

Collations, 81

column, 3, 7

column alias, 30, 244

column alias or aggregate expression,

- column constraints, data types (SQL), 208–210
 - CHECK constraints, 209–210
 - DEFAULT, 209
 - NULL or NOT NULL, 208
- column name, 3
- column names
 - qualifying, 54–55
- columns, 25, 28, 33, 184
 - grouping, 137
 - numeric data type, 184
 - selecting, 135–138
- columns with certain large data types, 117–122
- columns, order of, 8
- comments table, 273
- Common Era calendar, 200
- comparison operators, 79–82
- composite key, 221
- compound condition, 75
- compound conditions with AND and OR, 86–90
 - combining AND and OR, 89–90
 - truth tables, 87–89
- concatenation operator, 154–155
- condition (WHERE clause), 28
- conference (column), 8
- Constants, 3, 13
- CONSTRAINT keyword, 222
- Content Management System (CMS), 24, 25–27, 46, 118, 214, 259, 261–262, 268–274
- contents table, 272–273
- contents.content (TEXT column), 120
- conversions in numeric calculations
 - numeric data types, 193
- correlated subqueries, 93–96

- correlated subquery, 97
- correlation, 93
- COUNT (DISTINCT), 144–146
- COUNT function, 120, 144, 146
- COUNT(*), 30, 146–149
- COUNT(entry_id) aggregate function, 122
- COUNT(items.name), 138
- CREATE statement, 7–9, 12
- CREATE TABLE statement, 8, 9, 183
- cross join, 38, 44–45
- current/previous logic, 166
- CURRENT_DATE - INTERVAL 5 DAY, 85
- CURRENT_DATE (keyword), 85
- CURRENT_DATE (time function), 207
- CURRENT_TIME (time function), 207
- CURRENT_TIMESTAMP (time function), 207
- customer table, 87
- customers table, 281–282
- customers.name, 109
- customers.name column, 137

D

- data, 6
- Data Definition Language (DDL), 6–11, 22
 - ALTER statement, 7, 10
 - CREATE statement, 7–9, 12
 - DROP statement, 7, 10
 - starting over, 10–11
- Data Manipulation Language (DML), 6, 11–20, 22
 - DELETE statement, 18
 - INSERT statement, 12–15
 - SELECT statement, 18

- UPDATE statement, 15–17
- data model diagrams, 259–260
- data modelling, 212–219
 - entities and attributes, 213–214
 - entities and relationships, 214–219
- data retrieval, 2, 19–20
- data types, 180
- data types (SQL), 183–210
 - character data types, 194–200
 - column constraints, 208–210
 - numeric data types, 185–194
 - overview, 184
 - pros and cons of non-standard data types, 186
 - temporal data types, 200–208
 - wrapping up, 210
- database design, 2, 211
- database designer, 214
- database development, 10
- database environments, 2
- database objects, 6
- database optimizer, 100
- database structure, 6
- database system, 189
- database system software, 253–254
- database system, connecting to the, 255–256
- database systems, 21, 204
- DATE, 206
- DATE (keyword), 201
- DATE (temporal data types), 200–203
 - input format, storage format and display format, 201–203
- date arithmetic, 85
- DATE data type, 200
- date functions (temporal data types), 207–208
- date values, inserting, 201–203
- DATETIME, 265
- DB2, 7, 15, 22, 156, 185, 254
- DECIMAL, 187–188
- decimal data type
 - latitude and longitude, 190–191
 - precision, scale and accuracy, 190
- decimals
 - numeric data types, 187–191
- DEFAULT (column constraints), 209
- DELETE statement, 18
- deprecated (data type), 188
- derived tables, 70–71
- DESC (keyword), 32
- DESC and ORDER BY clause, 164–166
- designated database administrators (DBAs), 9
- detail query, 104
- detail rows, 104, 135–136
- Discussion Forum application, 138–140
 - forums table, 139
 - members table, 139
 - posts table, 140
 - threads table, 139–140
- Discussion Forums, 138
- Discussion Forums application, 213, 217, 221, 223, 224, 262, 275–278
 - forums table, 275
 - members table, 275–276
 - posts table, 276–278
 - threads table, 276
- Display Estimated Execution Plan, 258
- displaying query results on a Web Page, 27
- DISTINCT (keyword), 144
- dot notation, 48, 54
- drill-down SQL, 113–114

DROP statement, 7, 10

E

entities (data modelling), 213–219

entities and relationships (data modelling), 214–219

entity-relationship (ER) diagram, 214–219

entity-relationship (ER) modelling tools, 219

entries table, 25–27, 268–270

full outer join, 63–67

inner join, 48–56

left outer join, 56–61

one-to-many relationships, 50

right outer join, 61–63

some of it is hidden, 49

entries_with_category view, 272

entrykeywords table, 247, 248, 249, 261, 274

equals sign = (keyword), 3

error

full outer join, 64

execution plan for a query, 257

EXISTS conditions, 96–99

exponent, 191

EXTRACT, 153

F

floating-point data type, 193

floating-point numbers

numeric data types, 191–193

FOREIGN KEY keyword, 228

foreign keys (relational integrity), 224–232

declaring, 230

how they work, 225–226

must reference a key, 227–230

NULL, 230

ON DELETE and ON UPDATE, 231–232

using, 226–232

forums Table, 139

forums table, 275

FROM (keyword), 3

FROM clause, 1, 19–20, 24–27, 33, 115, 133, 134, 180, 182

derived tables, 70–71

more than one table using JOINS, 37–45

one table, 37

real world joins, 46–67

result set, 73

subqueries, 70–71

testing, 105

views, 67–69

Why start with?, 36–37

wrapping up, 71

front-end applications, 255–256

full outer join, 38, 42–43

categories and entries, 63–67

union queries, 64–67

FULL OUTER JOIN (keyword), 64

G

games table, 239, 267

granularity, 137

GROUP BY

using ORDER BY with, 175–176

GROUP BY clause, 20, 29–31, 34, 103–123, 133, 135, 159, 163, 166, 182

and aggregate functions, 117

- columns with certain large data types, 117
 - group rows, 125–126
 - HAVING without a, 128–132
 - how it works, 115–117
 - in context, 114–115
 - rules for, 117–122
 - subquery, 118, 120
 - wrapping up, 122–123
 - GROUP BY queries, 104
 - group condition, 127
 - group row, 109
 - group rows, 29, 104, 115–117, 136–138
 - GROUP_CONCAT, 250
 - Grouping, 104
 - grouping columns, 137
- ## H
- HAVING clause, 20, 29–31, 34, 125–132, 133, 134, 182
 - Are thresholds database or application logic?, 130–131
 - use of common aliases in the, 131–132
 - wrapping up, 132
 - HAVING filters group rows, 125–132
 - HAVING without a GROUP BY clause, 128–132
 - threshold alert, 129–132
 - help, SQL, 257–258
- ## I
- IBM DB2, 254, 256
 - IBM DB2 Express-C, 254
 - id (column), 8
 - id (identifier), 3
 - ID number, 25
 - Identifiers, 3
 - case-sensitivity, 9
 - identity (relational integrity), 212, 235
 - primary keys, 219–221
 - IN conditions, 91–93
 - with subqueries, 92–93
 - indexed retrieval, 101
 - indexes
 - WHERE clause performance, 100–102
 - Indexing, 258
 - inner join, 38, 39
 - all columns are available after a join, 52–53
 - categories and entries, 48–56
 - difference between outer join, 43
 - qualifying column names, 54–55
 - table aliases, 56
 - when a join is executed in a query, 54
 - INSERT statement, 12–15
 - INTEGER, 9
 - integers
 - numeric data types, 185–186
 - intervals (temporal data types), 206–207
 - IS NULL test, 99, 100
 - items table, 78, 105, 279–281
 - items.name, 163
- ## J
- JavaScript, 167
 - join condition, 38
 - joined tables, 37–67
 - all columns are available after a join, 52–53
 - cross, 44–45
 - difference between inner and outer joins, 43
 - full outer, 63–67

- inner, 39–40
- joining a table to itself, 240–246
- joining to a table twice, 237–240
- left outer, 56–61
- outer, 40–43
- query, 55
- real world, 46–56
- right outer, 61–63
- types of, 38–39
- when a join is executed in a query, 54

K

- keys (*see* foreign keys (relational integrity); natural keys (relational integrity); primary keys (data modeling); surrogate keys (relational integrity))
- keywords, 3, 246–251, 261, 285–286

L

- left outer join, 38, 40–41
 - categories and entries, 56–61
- LEFT OUTER JOIN, 99, 100
 - avoid using COUNT(*), 149
- LIKE operator, 82

M

- major-to-minor sequencing
 - ORDER BY clause, 162
- mantissa, 191
- manuals, database, 21–22
- many-to-many relationship, 216, 226, 246–251, 261, 263
- many-to-one relationship, 215
- mastering SQL, 2
- MEDIUMINT, 186

- members table, 139, 275–276
- Microsoft SQL Server, 254, 256
- Microsoft SQL Server Express, 253
- MONEY (data type), 188
- multiple row constructors, 15
- MySQL, 7, 22, 64, 76, 156, 185, 186, 201, 205, 229, 230, 250, 254, 256
 - CONCAT function, 155
 - SUBSTRING_INDEX, 155
- MySQL Community Server, 253
- MySQL DDL, 219

N

- name (column), 8
- name (identifier), 3
- natural keys (relational integrity), 232–235
 - autonumbers, 234–235
 - using, 234
- NCHAR column, 198
- non-standard data types, 186
- NOT (keyword), 75
- NOT EXIST condition, 98–99
- NOT IN condition, 98–99
- NOT NULL (column), 208
- NOT NULL (name column), 9
- NULL, 9, 77
- NULL (column), 208
- NULL (outer joins), 60
- NULL or NOT NULL (column constraints), 208
- NULL value, 220
- NULL values, 169–173, 180
- NULLIF function, 153–154
- NULLs by design, 144
- NUMERIC (decimal data type), 187–188
- numeric data types, 185–194

- conversions in numeric calculations, 193
 - decimals, 187–191
 - floating-point numbers, 191–193
 - integers, 185–186
 - numeric functions, 193–194
 - numeric functions
 - numeric data types, 193–194
 - numeric identifiers, 212
 - numeric operators, 154
 - NVARCHAR column, 198
- O**
- “One-to-Zero-or-Many” relationships, 107
 - ON clause, 39
 - ON DELETE clause, 231
 - ON UPDATE clause, 231
 - one-to-many relationship, 50, 215, 226, 242
 - one-to-many relationships, 247, 261
 - one-to-many table relationships, 250
 - one-to-one relationship, 216
 - online SQL reference manuals, 22
 - operator (keyword), 3
 - operators
 - BETWEEN, 83–86
 - comparison, 79–82
 - concatenation operator, 154–155
 - LIKE, 82
 - numeric, 154
 - SELECT clause, 154–156
 - temporal, 155–156
 - operators (WHERE clause), 79–86
 - Oracle, 7, 22, 156, 254, 256
 - Oracle Database Express Edition, 254
 - ORDER BY clause, 20, 32–33, 34, 108, 161–182
 - ASC and DESC, 164–166
 - dealing with problems, 169
 - detecting groupings in applications, 166
 - expressions, 176
 - how it works, 162–173
 - major-to-minor sequencing, 162
 - performance, 167–168
 - scope of, 173–181
 - sequence of values, 168–169
 - special sequencing, 176–177
 - syntax, 162
 - using with GROUP BY, 175–176
 - when it seems unnecessary, 168
 - with UNION queries, 178–181
 - wrapping up, 182
 - ORDER BY expressions, 176
 - ORDER BY syntax, 162
 - outer join, 40–43
 - (*see also* full outer join; left outer join; right outer join)
 - difference between inner join, 43
 - produce nulls, 60
 - OUTER JOIN (keyword), 40
 - outer table, 61
- P**
- parsing an SQL statement, 36–37
 - pattern matching, 82
 - performance problems, SQL, 257–258
 - PostgreSQL, 7, 15, 22, 185, 253, 254, 256
 - posts table, 140, 276–278
 - precision (decimals), 187
 - primary keys (data modelling), 219–221

Q

qualifying column names, 54–55

queries

correlated subqueries, 93–96

execution plan, 257–258

GROUP BY, 104

subqueries, 92–93

UNION, 178–181

queries, union, 64–67

query

SQL (Structured Query Language), 99,
257

query results on a web page, 27

query, join, 55

query, when a join is executed in a, 54

R

range test, 83

real world joins, 46–67

REFERENCES keyword, 228

referential integrity, 228

reflexive relationship, 242

relational integrity, 211–236

data modelling, 212–219

foreign keys, 224–232

identity, 212

natural versus surrogate keys, 232–235

primary keys, 219–221

UNIQUE constraints, 221–223

wrapping up, 235–236

result set, 20

right outer join, 38, 41–42

categories and entries, 60–63

row, 7, 12

row constructor, 14–15

row constructors, 265

rows

detail, 135–136

group, 136–138

S

sample applications, SQL, 259–263

scalar functions, 141, 149–154

scale (decimals), 187

script, 11

SELECT (keyword), 3, 157

SELECT *, 156–157

SELECT * (keyword), 105

SELECT clause, 5, 19–20, 24–27, 33,
106, 120, 133–159, 240

Discussion Forum application, 138–
140

functions, 140–154

operators, 154–156

SELECT *, 156–157

SELECT DISTINCT, 157–159

sequence in execution, 134–135

Which columns can be selected?, 135–
138

wrapping up, 159

SELECT DISTINCT, 157–159

SELECT SQL statement, 161

select star (*), 105

SELECT statement, 2, 18–20, 23–24, 33,
133, 135, 183

clauses, 19–20

data retrieval, 19–20

tabular structure, 20

self-documenting, 55

semantics, 5

sequence of values

ORDER BY clause, 168–173

Sequencing, 104

- Shopping Cart, 104
- shopping carts, 76–79, 263, 279–284
 - cartitems table, 283
 - carts table, 282
 - customers table, 281–282
 - items table, 279–281
 - vendors table, 284
- sitemap
 - application for left outer joins, 59
- SMALLINT, 185, 186
- SMALLINT column, 208
- SMALLINT values, 204
- special sequencing
 - ORDER BY clause, 176–177
- special structures, 237–251
 - implementing a many-to-many relationship, 246–251
 - joining a table to itself, 240–246
 - joining to a table twice, 237–240
 - wrapping up, 251
- splittime column, 205
- SQL (Structured Query Language), 1
 - aggregate functions, 141–149
 - functions, 140–154
 - help, 257–258
 - keywords, 285–286
 - language, 2
 - online reference manuals, 22
 - performanve problems, 257–258
 - query, 2, 99, 257
 - sample applications, 259–263
 - scalar functions, 149–154
 - script, 11
 - standards, 2, 20–22
 - structure, 1
 - syntax, 7
 - tabular structures, 1, 7, 13, 20
 - testing environment, 253–258
 - trying out the sample, 24
 - trying out your, 7
 - types of statements, 2
- SQL data types (*see* data types (SQL))
- SQL or Sequel, 2
- SQL reference, 254
- SQL reference manual, 201, 258, 285
- SQL scalar functions
 - commonly available in all database systems, 149–154
- SQL script library, 256–257
- SQL Server, 7, 22, 185, 186, 201
- SQL Server 2008 Express with Tools, 256
- SQL standard, 198, 201
- SQL statement, parsing an, 36–37
- SQL statements, 1, 7, 22, 133, 161
 - clauses, 4
 - Data Definition Language (DDL), 6
 - Data Manipulation Language (DML), 6
 - keywords, identifiers and constants, 2–3
 - overview, 1–6
 - script, 11
 - syntax, 5–6
 - types of, 2
- SQL, drill-down, 113–114
- SQL-1999, 185
- SQL-2003 standard, 185
- standards, SQL, 2, 20–22
- starting over, 10–11
- string functions (character data types), 199–200
- strings, 13
- structure, SQL, 1
- Structured Query Language (SQL), 1

- subqueries, 70–71
- subquery, GROUP BY clause, 118, 120–122
- subselects (union queries), 65
- SUBSTRING function, 149–151
- surrogate keys (relational integrity), 232–235
 - and redundancy, 234
 - autonumbers, 234–235
 - UNIQUE constraint, 235
- syntax, 5–6

T

- table, 3
- table aliases, 56, 244
- table name, 3
- table scan, 101
- tables
 - (*see also* joined tables)
 - creating, 39
 - single, 37
- tables, multiple, 37
- tables, single
 - FROM clause, 37
- tabular structure, 1, 7, 13, 20, 73, 105
- tags, 246
- teams (identifier), 3
- Teams and Games application, 266–267
- teams and games data model, 260
- teams table, 8, 239, 266
- temporal data types, 184, 200–208
 - DATE, 200–203
 - date functions, 207–208
 - intervals, 206–207
 - TIME, 203–205
 - TIMESTAMP, 205–206
- temporal operators, 155–156

- temporary table, 115
- testing environment, SQL, 253–258
- TEXT column, 28
- threads table, 139–140, 276
- threshold alert, 129–132
- thresholds, database or application logic, 130–131
- TIME, 206
- TIME (temporal data types), 203–205
 - times as duration, 204–205
 - times as point in time, 205
- time as duration, 204
- TIME data type, 200
- TIME values, 205
- times as points in time, 205
- TIMESTAMP, 265
- timestamp (temporal data types), 184
- TIMESTAMP (temporal data types), 205–206
- TINYINT, 186
- TINYINT values, 204
- truth tables, 87–89

U

- UNION (keyword), 65
- UNION ALL, 67
- union queries
 - full outer join, 64–67
- UNION Queries
 - ORDER BY with, 178–181
- UNIQUE constraint
 - NULL, 230
- UNIQUE constraints (relational integrity), 221–223, 235
 - dealing with keys in application code, 223
- UNKNOWN (WHERE clause), 75, 79, 88

UPDATE statement, 15–17

V

VARCHAR, 9

VARCHAR column, 195–196

vendors table, 284

Views, 67–69

W

web development

views, 69

WEEKDAY (date function), 207

WHERE (keyword), 3

WHERE clause, 20, 28–29, 33, 115, 133,
134, 168, 182

compound conditions with AND and
OR, 86–93

conditions, 74–75

correlated subqueries, 93–96

EXIST conditions, 96–99

operators, 79–86

performance, 99–102

shopping carts, 76–79

wrapping up, 102

WHERE condition, 74–75

conditions that are true, 74

when 'Not True' is preferable, 75

wildcard characters, 82

Wordpress, 25

Y

YYYY-MM-DD format, 202