



SQL FOR MySQL DEVELOPERS

A Comprehensive Tutorial and Reference

Rick F. van der Lans

SQL for MySQL Developers

This page intentionally left blank

SQL for MySQL Developers

A Comprehensive Tutorial and Reference

Rick F. van der Lans

Translated by Diane Cools

◆ Addison-Wesley

Upper Saddle River, NJ ■ Boston ■ Indianapolis ■ San Francisco

New York ■ Toronto ■ Montreal ■ London ■ Munich ■ Paris ■ Madrid

Cape Town ■ Sydney ■ Tokyo ■ Singapore ■ Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com



The Safari® Enabled icon on the cover of your favorite technology book means the book is available through Safari Bookshelf. When you buy this book, you get free access to the online edition for 45 days. Safari Bookshelf is an electronic reference library that lets you easily search thousands of technical books, find code samples, download chapters, and access technical information whenever and wherever you need it.

To gain 45-day Safari Enabled access to this book:

- Go to <http://www.awprofessional.com/safarienabled>
- Complete the brief registration form
- Enter the coupon code FFJ5-JWCL-7C3G-CUKJ-89CM

If you have difficulty registering on Safari Bookshelf or accessing the online edition, please e-mail customer-service@safaribooksonline.com.

Visit us on the Web: www.awprofessional.com

Library of Congress Cataloging-in-Publication Data

Lans, Rick F. van der.

SQL for MySQL developers : a comprehensive tutorial and reference / Rick F. van der Lans.
p. cm.

ISBN 978-0-13-149735-1 (pbk. : alk. paper) 1. SQL (Computer program language) 2. MySQL
(Electronic resource) I. Title.

QA76.73.S67L345 2007

005.13'3—dc22

2007000578

Copyright © 2007 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN 0131497359

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing, April 2007

Dedicated to Alyssa.

This page intentionally left blank

Contents

PART I	Introduction.	1
CHAPTER 1	Introduction to MySQL.	3
	1.1 Introduction	3
	1.2 Database, Database Server, and Database Language.	4
	1.3 The Relational Model	6
	1.4 What Is SQL?	11
	1.5 The History of SQL.	16
	1.6 From Monolithic via Client/Server to the Internet	18
	1.7 Standardization of SQL	21
	1.8 What Is Open Source Software?	25
	1.9 The History of MySQL	26
	1.10 The Structure of This Book.	27
CHAPTER 2	The Tennis Club Sample Database	29
	2.1 Introduction	29
	2.2 Description of the Tennis Club	29
	2.3 The Contents of the Tables.	33
	2.4 Integrity Constraints.	35
CHAPTER 3	Installing the Software	37
	3.1 Introduction	37
	3.2 Downloading MySQL.	37
	3.3 Installation of MySQL.	38

	3.4 Installing a Query Tool	38
	3.5 Downloading SQL Statements from the Web Site	38
	3.6 Ready?	39
CHAPTER 4	SQL in a Nutshell	41
	4.1 Introduction	41
	4.2 Logging On to the MySQL Database Server	41
	4.3 Creating New SQL Users	43
	4.4 Creating Databases	45
	4.5 Selecting the Current Database	45
	4.6 Creating Tables	46
	4.7 Populating Tables with Data	48
	4.8 Querying Tables	49
	4.9 Updating and Deleting Rows	52
	4.10 Optimizing Query Processing with Indexes	54
	4.11 Views	55
	4.12 Users and Data Security	57
	4.13 Deleting Database Objects	57
	4.14 System Variables	58
	4.15 Grouping of SQL Statements	59
	4.16 The Catalog Tables	60
	4.17 Retrieving Errors and Warnings	68
	4.18 Definitions of SQL Statements	69
PART II	Querying and Updating Data	71
CHAPTER 5	SELECT Statement: Common Elements	73
	5.1 Introduction	73
	5.2 Literals and Their Data Types	74
	5.3 Expressions	88
	5.4 Assigning Names to Result Columns	92
	5.5 The Column Specification	94
	5.6 The User Variable and the SET Statement	95
	5.7 The System Variable	97
	5.8 The Case Expression	101
	5.9 The Scalar Expression Between Brackets	106

5.10 The Scalar Function	107
5.11 Casting of Expressions	111
5.12 The Null Value as an Expression	114
5.13 The Compound Scalar Expression	115
5.14 The Aggregation Function and the Scalar Subquery.	136
5.15 The Row Expression	137
5.16 The Table Expression	139
5.17 Answers	140
CHAPTER 6	
SELECT Statements, Table Expressions, and Subqueries	145
6.1 Introduction	145
6.2 The Definition of the SELECT Statement	145
6.3 Processing the Clauses in a Select Block.	150
6.4 Possible Forms of a Table Expression	156
6.5 What Is a SELECT Statement?	159
6.6 What Is a Subquery?	160
6.7 Answers	166
CHAPTER 7	
SELECT Statement: The FROM Clause.	171
7.1 Introduction	171
7.2 Table Specifications in the FROM Clause	171
7.3 Again, the Column Specification.	173
7.4 Multiple Table Specifications in the FROM Clause.	174
7.5 Pseudonyms for Table Names.	178
7.6 Various Examples of Joins	179
7.7 Mandatory Use of Pseudonyms	183
7.8 Tables of Different Databases	185
7.9 Explicit Joins in the FROM Clause.	185
7.10 Outer Joins	189
7.11 The Natural Join	195
7.12 Additional Conditions in the Join Condition.	196
7.13 The Cross Join.	199
7.14 Replacing Join Conditions with USING.	199
7.15 The FROM Clause with Table Expressions	200
7.16 Answers	208

CHAPTER 8	SELECT Statement: The WHERE Clause	213
	8.1 Introduction	213
	8.2 Conditions Using Comparison Operators	215
	8.3 Comparison Operators with Subqueries	222
	8.4 Comparison Operators with Correlated Subqueries.	227
	8.5 Conditions Without a Comparison Operator.	229
	8.6 Conditions Coupled with AND, OR, XOR, and NOT	231
	8.7 The IN Operator with Expression List.	235
	8.8 The IN Operator with Subquery	241
	8.9 The BETWEEN Operator	250
	8.10 The LIKE Operator	252
	8.11 The REGEXP Operator	255
	8.12 The MATCH Operator	264
	8.13 The IS NULL Operator	276
	8.14 The EXISTS Operator	278
	8.15 The ALL and ANY Operators	281
	8.16 Scope of Columns in Subqueries	289
	8.17 More Examples with Correlated Subqueries	294
	8.18 Conditions with Negation.	299
	8.19 Answers	302
CHAPTER 9	SELECT Statement: SELECT Clause and Aggregation Functions	315
	9.1 Introduction	315
	9.2 Selecting All Columns (*)	316
	9.3 Expressions in the SELECT Clause	317
	9.4 Removing Duplicate Rows with DISTINCT.	318
	9.5 When Are Two Rows Equal?	321
	9.6 More Select Options.	323
	9.7 An Introduction to Aggregation Functions.	324
	9.8 COUNT Function	327
	9.9 MAX and MIN Functions	331
	9.10 The SUM and AVG Function.	336
	9.11 The VARIANCE and STDDEV Functions.	341
	9.12 The VAR_SAMP and STDDEV_SAMP Functions	343
	9.13 The BIT_AND, BIT_OR, and BIT_XOR Functions	343
	9.14 Answers	345

CHAPTER 10	SELECT Statement: The GROUP BY Clause	349
	10.1 Introduction	349
	10.2 Grouping on One Column	350
	10.3 Grouping on Two or More Columns	353
	10.4 Grouping on Expressions	356
	10.5 Grouping of Null Values	357
	10.6 Grouping with Sorting	358
	10.7 General Rules for the GROUP BY Clause	359
	10.8 The GROUP_CONCAT Function	362
	10.9 Complex Examples with GROUP BY	363
	10.10 Grouping with WITH ROLLUP	369
	10.11 Answers	372
CHAPTER 11	SELECT Statement: The HAVING Clause	375
	11.1 Introduction	375
	11.2 Examples of the HAVING Clause	376
	11.3 A HAVING Clause but not a GROUP BY Clause	378
	11.4 General Rule for the HAVING Clause	379
	11.5 Answers	381
CHAPTER 12	SELECT Statement: The ORDER BY Clause	383
	12.1 Introduction	383
	12.2 Sorting on Column Names	383
	12.3 Sorting on Expressions	385
	12.4 Sorting with Sequence Numbers	387
	12.5 Sorting in Ascending and Descending Order	389
	12.6 Sorting Null Values	392
	12.7 Answers	393
CHAPTER 13	SELECT Statement: The LIMIT Clause	395
	13.1 Introduction	395
	13.2 Get the Top	398
	13.3 Subqueries with a LIMIT Clause	402
	13.4 Limit with an Offset	404
	13.5 The Select Option SQL_CALC_FOUND_ROWS	405
	13.6 Answers	406

CHAPTER 14 Combining Table Expressions 409

14.1 Introduction 409

14.2 Combining with UNION. 410

14.3 Rules for Using UNION 413

14.4 Keeping Duplicate Rows. 416

14.5 Set Operators and the Null Value. 417

14.6 Answers 418

CHAPTER 15 The User Variable and the SET Statement. 421

15.1 Introduction 421

15.2 Defining Variables with the SET Statement 421

15.3 Defining Variables with the SELECT Statement 423

15.4 Application Areas for User Variables 425

15.5 Life Span of User Variables. 426

15.6 The DO Statement 428

15.7 Answers 428

CHAPTER 16 The HANDLER Statement. 429

16.1 Introduction 429

16.2 A Simple Example of the HANDLER Statement 429

16.3 Opening a Handler. 430

16.4 Browsing the Rows of a Handler 431

16.5 Closing a Handler. 435

16.6 Answers 435

CHAPTER 17 Updating Tables 437

17.1 Introduction 437

17.2 Inserting New Rows 437

17.3 Populating a Table with Rows from Another Table 442

17.4 Updating Values in Rows 444

17.5 Updating Values in Multiple Tables 450

17.6 Substituting Existing Rows 452

17.7 Deleting Rows from a Table 454

17.8 Deleting Rows from Multiple Tables. 456

17.9 The TRUNCATE Statement 458

17.10 Answers 458

CHAPTER 18	Loading and Unloading Data	461
	18.1 Introduction	461
	18.2 Unloading Data	461
	18.3 Loading Data	465
CHAPTER 19	Working with XML Documents	471
	19.1 XML in a Nutshell	471
	19.2 Storing XML Documents	473
	19.3 Querying XML Documents	476
	19.4 Querying Using Positions	484
	19.5 The Extended Notation of XPath	486
	19.6 XPath Expressions with Conditions	488
	19.7 Changing XML Documents	489
PART III	Creating Database Objects.	491
CHAPTER 20	Creating Tables	493
	20.1 Introduction	493
	20.2 Creating New Tables	493
	20.3 Data Types of Columns	496
	20.4 Adding Data Type Options	508
	20.5 Creating Temporary Tables	514
	20.6 What If the Table Already Exists?	515
	20.7 Copying Tables	516
	20.8 Naming Tables and Columns	521
	20.9 Column Options: Default and Comment	522
	20.10 Table Options	524
	20.11 The CSV Storage Engine	532
	20.12 Tables and the Catalog	534
	20.13 Answers	537
CHAPTER 21	Specifying Integrity Constraints	539
	21.1 Introduction	539
	21.2 Primary Keys	541
	21.3 Alternate Keys	544
	21.4 Foreign Keys	546
	21.5 The Referencing Action	550
	21.6 Check Integrity Constraints	553

	21.7 Naming Integrity Constraints	556
	21.8 Deleting Integrity Constraints.	557
	21.9 Integrity Constraints and the Catalog	557
	21.10 Answers	558
CHAPTER 22	Character Sets and Collations	561
	22.1 Introduction	561
	22.2 Available Character Sets and Collations.	563
	22.3 Assigning Character Sets to Columns.	564
	22.4 Assigning Collations to Columns	566
	22.5 Expressions with Character Sets and Collations.	568
	22.6 Sorting and Grouping with Collations	571
	22.7 The Coercibility of Expressions.	573
	22.8 Related System Variables	574
	22.9 Character Sets and the Catalog	576
	22.10 Answers	576
CHAPTER 23	The ENUM and SET Types	577
	23.1 Introduction	577
	23.2 The ENUM Data Type	578
	23.3 The SET Data Type.	582
	23.4 Answers	589
CHAPTER 24	Changing and Dropping Tables	591
	24.1 Introduction	591
	24.2 Deleting Entire Tables.	591
	24.3 Renaming Tables	593
	24.4 Changing the Table Structure.	593
	24.5 Changing Columns.	595
	24.6 Changing Integrity Constraints.	599
	24.7 Answers	602
CHAPTER 25	Using Indexes	603
	25.1 Introduction	603
	25.2 Rows, Tables, and Files.	604
	25.3 How Does an Index Work?.	605
	25.4 Processing a SELECT Statement: The Steps	610
	25.5 Creating Indexes.	614

	25.6 Defining Indexes Together with the Tables	617
	25.7 Dropping Indexes	618
	25.8 Indexes and Primary Keys	619
	25.9 The Big PLAYERS_XXL Table	620
	25.10 Choosing Columns for Indexes	622
	25.11 Indexes and the Catalog	627
	25.12 Answers	630
CHAPTER 26	Views	631
	26.1 Introduction	631
	26.2 Creating Views	631
	26.3 The Column Names of Views	635
	26.4 Updating Views: WITH CHECK OPTION	636
	26.5 Options of Views	638
	26.6 Deleting Views	639
	26.7 Views and the Catalog	640
	26.8 Restrictions on Updating Views	641
	26.9 Processing View Statements	642
	26.10 Application Areas for Views	645
	26.11 Answers	650
CHAPTER 27	Creating Databases.	653
	27.1 Introduction	653
	27.2 Databases and the Catalog.	653
	27.3 Creating Databases	654
	27.4 Changing Databases.	655
	27.5 Dropping Databases.	656
CHAPTER 28	Users and Data Security	659
	28.1 Introduction	659
	28.2 Adding and Removing Users	660
	28.3 Changing the Names of Users	662
	28.4 Changing Passwords	663
	28.5 Granting Table and Column Privileges	664
	28.6 Granting Database Privileges	667
	28.7 Granting User Privileges	670
	28.8 Passing on Privileges: WITH GRANT OPTION	673
	28.9 Restricting Privileges.	674

	28.10 Recording Privileges in the Catalog	675
	28.11 Revoking Privileges	677
	28.12 Security of and Through Views	680
	28.13 Answers	682
CHAPTER 29	Statements for Table Maintenance	683
	29.1 Introduction	683
	29.2 The ANALYZE TABLE Statement	684
	29.3 The CHECKSUM TABLE Statement	685
	29.4 The OPTIMIZE TABLE Statement.	686
	29.5 The CHECK TABLE Statement	687
	29.6 The REPAIR TABLE Statement	689
	29.7 The BACKUP TABLE Statement	690
	29.8 The RESTORE TABLE Statement	691
CHAPTER 30	The SHOW, DESCRIBE, and HELP Statements.	693
	30.1 Introduction	693
	30.2 Overview of SHOW Statements	693
	30.3 Additional SHOW Statements	698
	30.4 The DESCRIBE Statement	699
	30.5 The HELP Statement	699
PART IV	Procedural Database Objects.	701
CHAPTER 31	Stored Procedures.	703
	31.1 Introduction	703
	31.2 An Example of a Stored Procedure.	704
	31.3 The Parameters of a Stored Procedure.	706
	31.4 The Body of a Stored Procedure.	707
	31.5 Local Variables	709
	31.6 The SET Statement.	712
	31.7 Flow-Control Statements	712
	31.8 Calling Stored Procedures	719
	31.9 Querying Data with SELECT INTO.	722
	31.10 Error Messages, Handlers, and Conditions	726
	31.11 Retrieving Data with a Cursor.	731
	31.12 Including SELECT Statements Without Cursors.	736
	31.13 Stored Procedures and User Variables	737
	31.14 Characteristics of Stored Procedures	737

	31.15 Stored Procedures and the Catalog	740
	31.16 Removing Stored Procedures	741
	31.17 Security with Stored Procedures	742
	31.18 Advantages of Stored Procedures	743
CHAPTER 32	Stored Functions	745
	32.1 Introduction	745
	32.2 Examples of Stored Functions	746
	32.3 More on Stored Functions	752
	32.4 Removing Stored Functions	753
CHAPTER 33	Triggers	755
	33.1 Introduction	755
	33.2 An Example of a Trigger	756
	33.3 More Complex Examples	759
	33.4 Triggers as Integrity Constraints	763
	33.5 Removing Triggers	765
	33.6 Triggers and the Catalog	765
	33.7 Answers	765
CHAPTER 34	Events	767
	34.1 What Is an Event?	767
	34.2 Creating Events	768
	34.3 Properties of Events	777
	34.4 Changing Events	778
	34.5 Removing Events	779
	34.6 Events and Privileges	779
	34.7 Events and the Catalog	780
PART V	Programming with SQL	783
CHAPTER 35	MySQL and PHP	785
	35.1 Introduction	785
	35.2 Logging On to MySQL	786
	35.3 Selecting a Database	787
	35.4 Creating an Index	788
	35.5 Retrieving Error Messages	790
	35.6 Multiple Connections Within One Session	791
	35.7 SQL Statements with Parameters	793
	35.8 SELECT Statement with One Row	794

	35.9 SELECT Statement with Multiple Rows	796
	35.10 SELECT Statement with Null Values	800
	35.11 Querying Data About Expressions	801
	35.12 Querying the Catalog	803
	35.13 Remaining MYSQL Functions	805
CHAPTER 36	Dynamic SQL with Prepared Statement.	807
	36.1 Introduction	807
	36.2 Working with Prepared SQL Statements	807
	36.3 Prepared Statements with User Variables.	810
	36.4 Prepared Statements with Parameters	810
	36.5 Prepared Statements in Stored Procedures	811
CHAPTER 37	Transactions and Multiuser Usage.	815
	37.1 Introduction	815
	37.2 What Is a Transaction?	815
	37.3 Starting Transactions	821
	37.4 Savepoints	822
	37.5 Stored Procedures and Transactions	824
	37.6 Problems with Multiuser Usage	825
	37.7 Locking	829
	37.8 Deadlocks	830
	37.9 The LOCK TABLE and UNLOCK TABLE Statements	830
	37.10 The Isolation Level	832
	37.11 Waiting for a Lock	834
	37.12 Moment of Processing Statements	834
	37.13 Working with Application Locks	835
	37.14 Answers	837
APPENDIX A	Syntax of SQL	839
	A.1 Introduction	839
	A.2 The BNF Notation	839
	A.3 Reserved Words in SQL	843
	A.4 Syntax Definitions of SQL Statements	845
APPENDIX B	Scalar Functions	903
APPENDIX C	System Variables	953
APPENDIX D	Bibliography	963
	Index	967

About the Author

Rick F. van der Lans is author of the classic *Introduction to SQL*, the definitive SQL guide that database developers have relied on for more than 20 years. This book has been translated into various languages and has sold more than 100,000 copies.

He is an independent consultant, author, and lecturer specializing in database technology, development tools, data warehousing, and XML. As managing director of the Netherlands-based R20/Consultancy, he has advised many large companies on defining their IT architectures.

Rick is an internationally acclaimed lecturer. Throughout his career, he has lectured in many European countries, South America, USA, and Australia. He chairs the European Meta Data Conference and DB2 Symposium, and writes columns for several magazines. He was a member of the Dutch ISO committee responsible for the ISO SQL Standard for seven years.

You can contact Rick via email at sql@r20.nl.

Preface

INTRODUCTION

Many books have been written about MySQL, the best-known open source database server. Then why another book? Most books about MySQL discuss a wide variety of topics, such as the installation of MySQL, using MySQL from PHP, and security. As a result, each topic cannot be explained in detail, and many questions of readers cannot be answered. This book focuses on one aspect of MySQL: the language that drives MySQL, which is *SQL* (Structured Query Language). Every developer working with MySQL should master this language thoroughly.

Especially in the more recent versions, SQL has been extended considerably. Unfortunately, many developers still limit themselves to those features that were available in the first versions. Not all the features of MySQL are fully used, which means that the product is not employed in the best way possible. The result is that complex statements and programs must be built needlessly. When you buy a house, you also do not restrict yourself to 20 percent of the rooms, do you? That is why this book contains a complete and detailed description of the SQL dialect as implemented in MySQL version 5.0.18. It should be seen primarily as a textbook rather than as a reference book; it will teach you the language, and you can complete the exercises to test your knowledge. After reading this book, you should be familiar with all the statements and features and some idiosyncrasies of MySQL's SQL, and you should be able to use it efficiently and effectively.

TOPICS

This book is completely devoted to the SQL dialect as implemented in MySQL. It discusses every aspect of the language thoroughly and critically. These aspects of SQL among others, are covered:

- Querying data (joins, functions, and subqueries)
- Updating data
- Creating tables and views
- Specifying primary and foreign keys and other integrity constraints
- Using indexes
- Considering data security
- Developing stored procedures and triggers
- Developing programs with PHP
- Working with transactions
- Using the catalog

FOR WHOM IS THIS BOOK INTENDED?

We recommend this book on MySQL's SQL dialect to those who want to use the full power of MySQL effectively and efficiently in practice. This book is therefore suitable for the following groups of people:

- **Developers** who develop applications with the help of MySQL
- **Database managers** who have to know the possibilities and impossibilities of SQL
- **Students** in higher education, including those in technical colleges, polytechnics, universities, and sixth-form colleges
- **Designers, analysts, and consultants** who have to deal, directly or indirectly, with MySQL and/or SQL and want to know about its possibilities and impossibilities
- **Home students** who are interested in MySQL and/or SQL
- **Users** who have the authority to use SQL to query the MySQL database of the company or institute for which they are working

- **Web site developers** who are creating web sites with the help of MySQL and languages such as PHP and Python
- **IT hobbyists** who are interested in MySQL and want to develop an SQL application using MySQL themselves

A PRACTICAL BOOK

This book should be seen primarily as a *textbook* and less as a reference work. To this end, it contains many examples and exercises (with answers). Do not ignore the exercises. Experience shows that you will learn the language more thoroughly and more quickly by practicing often and doing many exercises.

THE BOOK'S WEB SITE

When you leaf through the book, you will come across numerous SQL statements. Sometimes these are examples, and sometimes they are answers to questions. After you have installed MySQL, you can run through these statements to see whether they work and see their effects. You could type in all the statements again like a real Spartan, but you can also make life easy for yourself by downloading all the statements from the Internet. A special web site for this book, www.r20.nl, includes all the SQL statements.

We also have used the web site for these purposes:

- The web site includes an installation process and instructions for MySQL. You will find useful tips for installing MySQL under Windows. The site also explains the installation process of the example database.
- If an error is found in the book, the web site will rectify the mistake.
- Reader comments that could be of interest to others will be added periodically to site.
- We even will consider making additional chapters available on the web site in the future.

Therefore, keep an eye on this web site.

PREREQUISITE KNOWLEDGE

Some general knowledge of programming languages and database servers is required.

THE HISTORY OF THIS BOOK

It was 1984, and the database world was under the spell of a revolution. SQL had started its triumphal procession. Vendors such as IBM and Oracle had introduced the commercial versions of their SQL database servers, and the marketing machine went at full speed. The market reacted positively to this rise of first-generation SQL database servers. Many organizations decided to buy such a database server and gradually phase out their existing products.

My employer at that time had decided to get involved in this tumult as well. The company also wanted to make money with this new database language, and the plan was to start organizing SQL courses. Because of my background knowledge, I was charged with this task. That SQL would become such a success and that my agreement to present the courses would have far-reaching consequences (personally as well as professionally) never entered my mind.

After studying SQL closely, I started to develop the material for the course. After teaching SQL for two years with great pleasure, I got an idea to write a book about SQL. It would have to be a book completely dedicated to this language, with its many possibilities and idiosyncrasies.

After producing gallons of blood, sweat, and tears, I completed the first Dutch edition in 1986, entitled *Het SQL Leerboek*. The book did not focus on a specific SQL database server, but on the SQL standard. Barely before the book was published, I was asked to write an English version. That book, *Introduction to SQL*, was published in 1987 as the first English book completely devoted to SQL. After that, I wrote versions in German and Italian. Obviously, a need existed for information about SQL. Everyone wanted to learn about SQL, but not much information was available.

Because SQL was still young, development went fast. Statements were added, extended, and improved. New implementations became available, new application areas were discovered, and new versions of the SQL standard appeared. Soon a new edition of the book had to be written. And more was to come. And this will not be the last because SQL has gloriously won the revolution in the database world, and no competition is in sight on the horizon.

Through the years, many vendors have implemented SQL. At first, all these products had much in common, but slowly the number of differences increased. For that reason, I decided in 2003 to write a book specifically dedicated to the SQL dialect of MySQL. I thought it would be a piece of cake. I would use *Introduction to SQL* as an example, add some details of MySQL, and remove a few general aspects. How long could that take? Two weeks of hard work and some speed typing, and I'd have the book ready. However, that appeared to be a serious underestimation. To give a complete view of all the features, I had to dive deeply into the SQL dialect of MySQL. This book, which definitely took more than two weeks of writing, is the result of that time-consuming effort. Obviously, it is related to the book from which it is derived; however, it contains many MySQL-related details not included in *Introduction to SQL*.

AND FINALLY...

Writing this book was not a solo project. Many people have contributed to this book or previous editions. I would like to use this preface to thank them for their help, contributions, ideas, comments, mental support, and patience.

It does not matter how many times a writer reads his own work; editors remain indispensable. A writer reads not what he has written, but what he thinks he has written. In this respect, writing is like programming. That is why I owe a great deal to the following persons for making critical comments and giving very helpful advice: Klaas Brant, Marc van Cappellen, Ian Cargill, Corine Cools, Richard van Dijk, Rose Endres, Wim Frederiks, Andrea Gray, Ed Jedeloo, Josien van der Laan, Oda van der Lans, Deborah Leendertse, Arjen Lentz, Onno de Maar, Andrea Maurino, Sandor Nieuwenhuijs, Henk Schreij, Dave Slayton, Aad Speksnijder, Nok van Veen, John Vicherek, and David van der Waaij. They all have read this manuscript (or parts of it) or the manuscript of a previous edition, a translation of it, or an adjusted version.

I would like to thank Wim Frederiks and Roland Bouman separately for all the hours they spent editing this book. Both patiently studied each page and pointed out the errors and inconsistencies. I am very grateful to them for all the work they put into this project.

I would also like to thank the thousands of students across the world whom I have taught SQL over the past years. Their comments and recommendations have been invaluable in revising this book. In addition, a large number of readers of the previous edition responded to my request to send comments and suggestions. I want to thank them for the trouble they took to put these in writing.

From the first day I started working on the project, I had the support of the MySQL organization. They helped me by making the required software available. I want to thank this group very much for the support and help.

Again, I owe Diane Cools many thanks. As an editor, she made this book readable to others. For a writer, it is also reassuring to find someone who, especially in difficult times, keeps stimulating and motivating you. Thanks, Diane!

Finally, again I would like to ask readers to send comments, opinions, ideas, and suggestions concerning the contents of the book to sql@r20.nl, referencing *SQL for MySQL Developers*. Many thanks, in anticipation of your cooperation.

Rick F. van der Lans

Den Haag, The Netherlands, March 2007

This page intentionally left blank

Part I

Introduction

SQL is a compact and powerful language for working with databases. Despite this compactness, it cannot be described simply in a few chapters. We would do the language no justice then. And that certainly applies to MySQL's SQL dialect, which has many, many possibilities. For this reason, we start this book with a number of introductory chapters that form the first part.

In Chapter 1, “Introduction to MySQL,” we provide an overall description of SQL, including its background and history, and the history of MySQL. MySQL is *open source software*; in Section 1.8, we explain what that really means. We also describe a number of concepts in the relational model (the theory behind SQL).

This book contains many examples and exercises. So that you do not have to learn a new database for each example, we use the same database for most of these examples and exercises. This database forms the basis for the administration of an international tennis league. Chapter 2, “The Tennis Club Sample Database,” describes the structure of this database. Look closely at this before you begin the exercises.

We strongly recommend that you use MySQL when doing the exercises and get some hands-on experience. For this, you have to download and install the software, and create the example database. Chapter 3, “Installing the Software,” describes how to do that. Note that for several aspects, we refer to the web site of the book.

This part closes with Chapter 4, “SQL in a Nutshell,” which reviews all the important SQL statements. After reading this part, you should have both a general idea of what SQL offers as a language and an overall impression of what this book discusses.

Introduction to MySQL

1.1 INTRODUCTION

MySQL is a relational database server that supports the well-known SQL (Structured Query Language) database language. Therefore, MySQL is named after the language that developers use to store, query, and later update data in a MySQL database. In short, SQL is the native language of MySQL.

This chapter discusses the following topics. None of these topics is really important for studying MySQL's SQL. When you are familiar with these background topics, you can jump to the next chapter.

- The chapter starts with an explanation of basic subjects, such as the database, database server, and database language.
- SQL is based on theories of the *relational model*. To use SQL, some knowledge of this model is invaluable. Therefore, Section 1.3 describes the relational model.
- Section 1.4 briefly describes what SQL is, what can be done with the language, and how it differs from other languages (such as Java, Visual Basic, or PHP).
- Section 1.5 covers the history of SQL.
- Section 1.7 presents the most important current standards for SQL.
- MySQL is open source software. Section 1.8 explains what that really means.
- Section 1.9 discusses the history of MySQL and its vendors.
- The chapter closes with a description of the structure of the book. The book consists of several parts, with each part summarized in a few sentences.

1.2 DATABASE, DATABASE SERVER, AND DATABASE LANGUAGE

SQL (Structured Query Language) is a database language used for formulating statements processed by a database server. In this case, the database server is MySQL. The first sentence of this paragraph contains three important concepts: *database*, *database server*, and *database language*. We begin with an explanation of each of these terms.

What is a *database*? This book uses a definition derived from Chris J. Date's definition (see [DATE95]):

A **database** consists of some collection of persistent data that is used by the application systems of some given enterprise and managed by a database-management system.

Therefore, card index files do not constitute a database. On the other hand, the large files of banks, insurance companies, telephone companies, and the state transport department can be considered databases. These databases contain data about addresses, account balances, car registration plates, weights of vehicles, and so on. For example, the company you work for probably has its own computers, which are used to store salary-related data.

Data in a database becomes useful only if something is done with it. According to the definition, data in the database is managed by a separate programming system. This system is called a *database server* or *database management system* (DBMS). MySQL is such a database server. A database server enables users to process data stored in a database. Without a database server, it is impossible to look at data in the database or to update or delete obsolete data. The database server alone knows where and how data is stored. A definition of a database server appears in [ELMA06], by R. Elmasri:

A **database server** is a collection of programs that enables users to create and maintain a database.

A database server never changes or deletes the data in a database by itself; someone or something has to give the command for this to happen. Examples of commands that a user could give to the database server are 'delete all data about the vehicle with the registration plate number DR-12-DP' or 'give the names of all the companies that haven't paid the invoices of last March.' However, users cannot communicate with the database server directly; an application must present the

commands to a database server. An application always exists between the user and the database server. Section 1.4 discusses this in more detail.

The definition of the term *database* also contains the word *persistent*. This means that data in a database remains there permanently until it is changed or deleted explicitly. If you store new data in a database and the database server sends back the message that the storage operation was successful, you can be sure that the data will still be there tomorrow (even if you switch off your computer). This is unlike the data stored in the internal memory of a computer. If the computer is switched off, that data is lost forever; it is not persistent.

Commands are relayed to a database server with the help of special languages, called *database languages*. Users enter commands, also known as statements, that are formulated according to the rules of the database language, using special software; the database server then processes these commands. Every database server, regardless of manufacturer, possesses a database language. Some systems support more than one. All these languages are different, which makes it possible to divide them into groups. The *relational database languages* form one of these groups. An example of such a language is SQL.

How does a database server store data in a database? A database server uses neither a chest of drawers nor a filing cabinet to hold information; instead, computers work with storage media such as tapes, floppy disks, and magnetic and optical disks. The manner in which a database server stores information on these media is very complex and technical, and this book does not explain the details. In fact, you do not need this technical knowledge because one of the most important tasks of a database server is to offer *data independence*. This means that users do not need to know how or where data is stored. To users, a database is simply a large reservoir of information. Storage methods are also completely independent of the database language being used. In a way, this resembles the process of checking in luggage at an airport. Travelers do not care where and how the airline stores their luggage; they are interested only in whether the luggage arrives at their destinations.

Another important task of a database server is to maintain the *integrity* of the data stored in a database. This means, first, that the database server has to make sure that database data always satisfies the rules that apply in the real world. Take, for example, the case of an employee who is allowed to work for one department only. In a database managed by a database server, the database should not permit any employee to be registered as working for two or more departments. Second, integrity means that two different pieces of database data do not contradict one another. This is also known as *data consistency*. (As an example, in one place in a database, Mr. Johnson might be recorded as being born on August 4, 1964, and in another place he might have a birth date of December 14, 1946. These two pieces

of data are obviously inconsistent.) Each database server is designed to recognize statements that can be used to specify *constraints*. After these rules are entered, the database server takes care of their implementation.

1.3 THE RELATIONAL MODEL

SQL is based on a formal and mathematical theory. This theory, which consists of a set of concepts and definitions, is called the *relational model*. E. F. Codd defined the relational model in 1970 while at IBM. He introduced the relational model in the almost legendary article entitled “A Relational Model of Data for Large Shared Data Banks” (see [CODD70]). This relational model provides a theoretical basis for database languages. It consists of a small number of simple concepts for recording data in a database, together with a number of operators to manipulate the data. These concepts and operators are principally borrowed from *set theory* and *predicate logic*. Later, in 1979, Codd presented his ideas for an improved version of the model; see [CODD79] and [CODD90].

The relational model has served as an example for the development of various database languages, including QUEL (see [STON86]), SQUARE (see [BOYC73a]), and, of course, SQL. These database languages are based on the concepts and ideas of that relational model and, therefore, are called *relational database languages*; SQL is an example. The rest of this part concentrates on the following terms used in the relational model, which appear extensively in this book:

- Table
- Column
- Row
- Null value
- Constraint or integrity constraint
- Primary key
- Candidate key
- Alternate key
- Foreign key or referential key

Note that this is not a complete list of all the terms the relational model uses. Part III, “Creating Database Objects,” discusses most of these terms. For more extensive descriptions, see [CODD90] and [DATE95].

1.3.1 Table, Column, and Row

Data can be stored in a relational database in only one format: in *tables*. The official name for a table is actually *relation*, and the term *relational* model stems from this name. We have chosen to use the term *table* because SQL uses that word.

Informally, a table is a set of *rows*, with each row consisting of a set of *values*. All the rows in a certain table have the same number of values. Figure 1.1 shows an example of a table called the PLAYERS table. This table contains data about five players who are members of a tennis club.

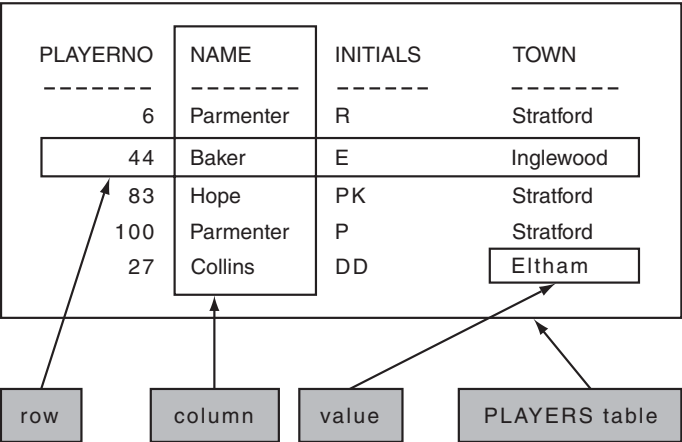


FIGURE 1.1 The concepts value, row, column, and table

This PLAYERS table has five *rows*, one for each player. A row with values can be considered a set of data elements that belong together. For example, in this table, the first row consists of the values 6, Parmenter, R, and Stratford. This information tells us that there is a player with number 6, that his last name is Parmenter and his initial is R, and that he lives in the town Stratford.

PLAYERNO, NAME, INITIALS, and TOWN are the names of the *columns* in the table. The PLAYERNO column contains the values 6, 44, 83, 100, and 27. This set of values is also known as the *population* of the PLAYERNO column. Each row has a value for each column. Therefore, the first row contains a value for the PLAYERNO column and a value for the NAME column, and so on.

A table has two special properties:

- The intersection of a row and a column can consist of only one value, an *atomic value*. An atomic value is an indivisible unit. The database server can deal with such a value only in its entirety.

- The rows in a table have no specific order; you should not think in terms of the first row, the last three rows, or the next row. Instead, consider the contents of a table to be a *set* of rows in the true sense of the word.

1.3.2 Null Value

Columns are filled with atomic values. For example, such a value can be a number, a word, or a date. A special value is the *null value*. The null value is comparable to “value unknown” or “value not present.” Consider Figure 1.1 as an example again. If we do not know the town of player 27, we could store the null value in the TOWN column for the row belonging to player 27.

A null value must not be confused with the number zero or spaces. It should be seen as a missing value. A null value is never equal to another null value, so two null values are not equal to each other, but they are also not unequal. If we knew whether two null values were equal or unequal, we would know *something* about those null values. Then we could not say that the two values were (completely) unknown. We discuss this later in more detail.

The term *null value* is, in fact, not entirely correct; we should be using the term *null* instead. The reason is that it is not a value, but rather a gap in a table or a signal indicating that the value is missing. However, this book uses that term to stay in line with various standards and products.

1.3.3 Constraints

The first section of this chapter described the integrity of the data stored in tables, the database data. The contents of a table must satisfy certain rules, the so-called *integrity constraints* (integrity rules). Two examples of integrity constraints are that the player number of a player may not be negative, and two different players may not have the same player number. Integrity constraints can be compared to road signs. They also indicate what is allowed and what is not allowed.

A relational database server should enforce integrity constraints. Each time a table is updated, the database server has to check whether the new data satisfies the relevant integrity constraints. This is a task of the database server. The integrity constraints must be specified first so that the database server knows what they are.

Integrity constraints can have several forms. Because some are used so frequently, they have special names, such as primary key, candidate key, alternate key, and foreign key. The analogy with the road signs applies here as well. Special symbols have been invented for frequently used road signs, and these also have been given names, such as a right-of-way sign or a stop sign. We explain those named integrity constraints in the following sections.



FIGURE 1.2 Integrity constraints are the road signs of a database

1.3.4 Primary Key

The *primary key* of a table is a column (or a combination of columns) used as a unique identification of rows in that table. In other words, two different rows in a table may never have the same value in their primary key, and for every row in the table, the primary key must always have one value. The `PLAYERNO` column in the `PLAYERS` table is the primary key for this table. Therefore, two players may never have the same number, and a player may never lack a number. The latter means that null values are not allowed in a primary key.

We come across primary keys everywhere. For example, the table in which a bank stores data about bank accounts has the column bank account number as primary key. Similarly, a table in which different cars are registered uses the license plate as primary key (see Figure 1.3).



FIGURE 1.3 License plate as possible primary key

1.3.5 Candidate Key

Some tables contain more than one column (or combination of columns) that can act as a primary key. These columns all possess the uniqueness property of a primary key. Here, also, null values are not allowed. These columns are called *candidate keys*. However, only one is designated as the primary key. Therefore, a table always has at least one candidate key.

If we assume that passport numbers are also included in the PLAYERS table, that column will be used as the candidate key because passport numbers are unique. Two players can never have the same passport number. This column could also be designated as the primary key.

1.3.6 Alternate Key

A candidate key that is not the primary key of a table is called an *alternate key*. Zero or more alternate keys can be defined for a specific table. The term *candidate key* is a general term for all primary and alternate keys. If every player is required to have a passport, and if we would store that passport number in the PLAYERS table, PASSPORTNO would be an alternate key.

1.3.7 Foreign Key

A *foreign key* is a column (or combination of columns) in a table in which the population is a subset of the population of the primary key of a table (this does not have to be another table). Foreign keys are sometimes called referential keys.

Imagine that, in addition to the PLAYERS table, a TEAMS table exists; see Figure 1.4. The TEAMNO column is the primary key of this table. The PLAYERNO column in this table represents the captain of each particular team. This has to be an existing player number, occurring in the PLAYERS table. The population of this column represents a subset of the population of the PLAYERNO column in the PLAYERS table. PLAYERNO in the TEAMS table is called a foreign key.

Now you can see that we can combine two tables. We do this by including the PLAYERNO column in the TEAMS table, thus establishing a link with the PLAYERNO column of the PLAYERS table.

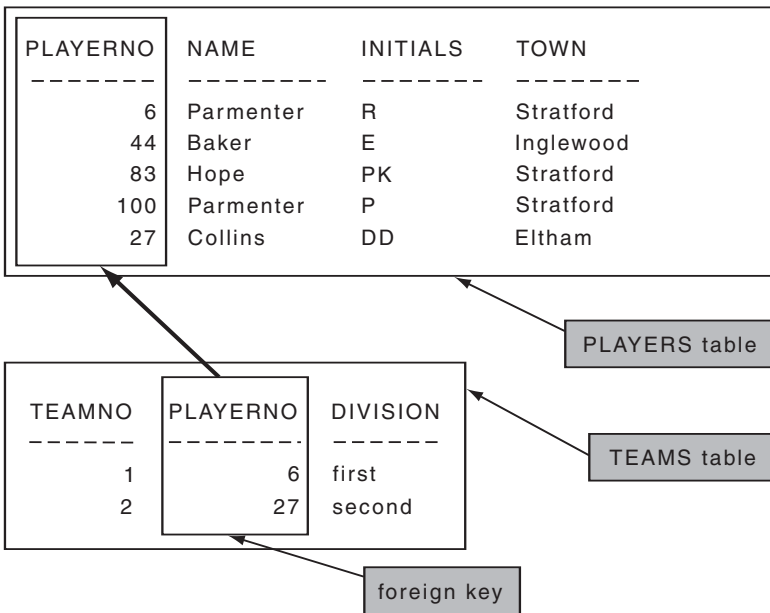


FIGURE 1.4 The foreign key

1.4 WHAT IS SQL?

As already stated, SQL (Structured Query Language) is a *relational database language*. Among other things, the language consists of statements to insert, update, delete, query, and protect data. The following statements can be formulated with SQL:

- Insert the address of a new employee.
- Delete all the stock data for product ABC.
- Show the address of employee Johnson.
- Show the sales figures of shoes for every region and for every month.
- Show how many products have been sold in London the last three months.
- Make sure that Mr. Johnson cannot see the salary data any longer.

Many vendors already have implemented SQL as the database language for their database server. MySQL is not the only available database server in which SQL has been implemented as database language. IBM, Microsoft, Oracle, and Sybase have manufactured SQL products as well. Thus, SQL is not the name of a certain product that has been brought to market only by MySQL.

We call SQL a relational database language because it is associated with data that has been defined according to the rules of the relational model. (However, we must note that, on particular points, the theory and SQL differ; see [CODD90].) Because SQL is a relational database language, for a long time it has been grouped with the declarative or nonprocedural database languages. By *declarative* and *non-procedural*, we mean that users (with the help of statements) have to specify only *which* data elements they want, not *how* they must be accessed one by one. Well-known languages such as C, C++, Java, PHP, Pascal, and Visual Basic are examples of procedural languages.

Nowadays, however, SQL can no longer be called a pure declarative language. Since the early 1990s, many vendors have added procedural extensions to SQL. These make it possible to create procedural database objects such as *triggers* and *stored procedures*; see Part IV, “Procedural Database Objects.” Traditional statements such as IF-THEN-ELSE and WHILE-DO have also been added. Although most of the well-known SQL statements are still not procedural by nature, SQL has changed into a hybrid language consisting of procedural and nonprocedural statements. Recently, MySQL has also been extended with these procedural database objects.

SQL can be used in two ways. First, SQL can be used *interactively*. For example, a user enters an SQL statement on the spot, and the database server processes it immediately. The result is also immediately visible. Interactive SQL is intended for application developers and for end users who want to create reports themselves.

The products that support interactive SQL can be split in two groups: the somewhat old-fashioned products with a terminal-like interface and those with a modern graphical interface. MySQL includes a product with a terminal-like interface that bears the same name as the database server: `mysql`. Figure 1.5 shows this program. First, an SQL statement is entered (`SELECT * FROM PLAYERS`); the result is shown underneath as a table.

However, some products have a more graphical interface available for interactive use, such as MySQL Query Browser from MySQL, SQLyog from Webyog, phpMyAdmin, Navicat from PremiumSoft (see Figure 1.6), and WinSQL from Synametrics (see Figure 1.7).

```

mysql> SELECT * FROM PLAYERS;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| PLAYERNO | NAME      | INITIALS | BIRTH_DATE | SEX | JOINED | STREET      | HOUSENO | POSTCODE | TOWN      | PHONENO | LEAGUENO |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 2 | Everett  | R        | 1948-09-01 | M   | 1975   | Stoney Road | 43       | 3575NH   | Stratford | 070-237893 | 2411 |
| 6 | Parmenter | R        | 1964-06-25 | M   | 1977   | Haseltine Lane | 88       | 1234KK   | Stratford | 070-476537 | 8467 |
| 7 | Wise     | GWS      | 1963-05-11 | M   | 1981   | Edgecombe Way | 39       | 9758UD   | Stratford | 070-347689 | NULL |
| 8 | Newcastle | B        | 1962-07-08 | F   | 1980   | Station Road | 4        | 6384UD   | Inglewood | 070-438458 | 2983 |
| 27 | Collins  | DD       | 1964-12-28 | F   | 1983   | Long Drive    | 884      | 8457DK   | Eltham    | 079-234857 | 2513 |
| 28 | Collins  | C        | 1963-06-22 | F   | 1983   | Old Main Road | 10       | 1294QK   | Highurst  | 010-659599 | NULL |
| 39 | Bishop   | D        | 1956-10-29 | M   | 1980   | Eaton Square | 78       | 9629GD   | Stratford | 070-393435 | NULL |
| 44 | Baker    | E        | 1963-01-09 | M   | 1980   | Lewis Street  | 23       | 4444LJ   | Inglewood | 070-368753 | 1124 |
| 57 | Brown    | M        | 1971-08-17 | M   | 1985   | Edgecombe Way | 15       | 4372CB   | Stratford | 070-423458 | 6489 |
| 83 | Hope     | PK       | 1956-11-11 | M   | 1982   | Magdalene Road | 16A      | 1812UP   | Stratford | 070-353548 | 1688 |
| 95 | Miller   | P        | 1963-05-14 | M   | 1972   | High Street   | 33A      | 5746OP   | Douglas   | 070-867564 | NULL |
| 100 | Parmenter | P       | 1963-02-28 | M   | 1979   | Baseline Lane | 88       | 6494SG   | Stratford | 070-494592 | 6524 |
| 104 | Moorman  | D        | 1970-05-10 | F   | 1984   | Stout Street  | 65       | 9437AQ   | Eltham    | 079-987571 | 7860 |
| 112 | Bailey   | IP       | 1963-10-01 | F   | 1984   | Vixen Road    | 8        | 6392LR   | Plymouth  | 010-548745 | 1319 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
14 rows in set (0.00 sec)

```

FIGURE 1.5 An example of the query program called `mysql` that can be used to specify the SQL statements interactively

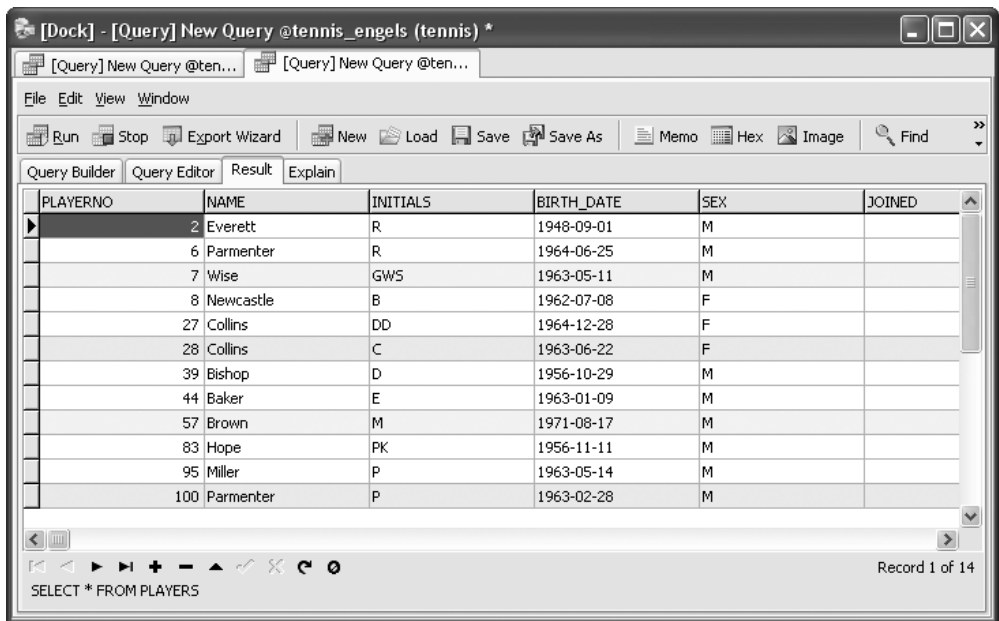


FIGURE 1.6 An example of the query program Navicat

WinSQL Lite - [MySQL via root: Query - Untitled 1]

File Edit View Query Window Help

Queries: #1 SELECT * FROM PLAYER...

Query Result Catalog

PLAYERNO	NAME	INITIALS	BIRTH_DATE	SEX	JOINED	STREET	HO
2	Everett	R	1948-09-01	M	1975	Stoney Road	43
6	Parmenter	R	1964-06-25	M	1977	Haseltine Lane	80
7	Wise	GWS	1963-05-11	M	1981	Edgecombe Way	39
8	Newcastle	B	1962-07-08	F	1980	Station Road	4
27	Collins	DD	1964-12-28	F	1983	Long Drive	80
28	Collins	C	1963-06-22	F	1983	Old Main Road	10
39	Bishop	D	1956-10-29	M	1980	Eaton Square	78
44	Baker	E	1963-01-09	M	1980	Lewis Street	23
57	Brown	M	1971-08-17	M	1985	Edgecombe Way	16
83	Hope	PK	1956-11-11	M	1982	Magdalene Road	16
95	Miller	P	1963-05-14	M	1972	High Street	33
100	Parmenter	P	1963-02-28	M	1979	Haseltine Lane	80
104	Moorman	D	1970-05-10	F	1984	Stout Street	65
112	Bailey	IP	1963-10-01	F	1984	Vixen Road	8

14 Row(s) affected

Line 21, Pos 0 Conn.: MySQL via root (MySQL) Execution time: 0:0:0.62

FIGURE 1.7 An example of the query program WinSQL

The second way in which SQL can be used is called *preprogrammed* SQL. Here, the SQL statements are embedded in an application that is written in another programming language. Results from these statements are not immediately visible to the user but are processed by the *enveloping* application. Preprogrammed SQL appears mainly in applications developed for end users. These end users do not need to learn SQL to access the data, but they work from simple screens and menus designed for their applications. Examples are applications to record customer information and applications to handle stock management. Figure 1.8 shows an example of a screen with fields in which the user can enter the address without any knowledge of SQL. The application behind this screen has been programmed to pass certain SQL statements to the database server. The application therefore uses SQL statements to transfer the information that has been entered into the database.

In the early stages of the development of SQL, only one method existed for preprogrammed SQL, called *embedded* SQL. In the 1980s, other methods appeared. The most important is called *call level interface* SQL (CLI SQL). Many variations of CLI SQL exist, such as ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity). The most important ones are described in this book. The different methods of preprogrammed SQL are also called the *binding styles*.

New Recipient - [R20]

Personal Information

Title: [] [v] First Name: [] Last: []

Company: []

Address1: []

Address2: []

City: [] State/Country: [] Zip Code: []

Notes: []

Billing: [] Misc: []

Connections

	Country	Area	Local Number	Extension	
Fax [v]	[]	[]	[]	x []	Programs...
Voice:	[]	[]	[]	x []	
Mail [v]	[]				Find...

Send From

Office: Fax [v] Home: Fax [v] Away: Fax [v]

Last Values OK Cancel

FIGURE 1.8 SQL is shielded in many applications; users can see only the input fields.

The statements and features of interactive and preprogrammed SQL are virtually the same. By this, we mean that most statements that can be entered and processed interactively can also be included (embedded) in an SQL application. Preprogrammed SQL has been extended with a number of statements that were added only to make it possible to merge the SQL statements with the non-SQL statements. This book is primarily focused on interactive SQL. Preprogrammed SQL is dealt with later in the book.

Three important components are involved in the interactive and preprogrammed processing of SQL statements: the user, the application, and the database server (see Figure 1.9). The database server is responsible for storing and accessing data on disk; the application and the user have nothing to do with this. The database server processes the SQL statements that the application delivers. In a defined way, the application and the database server can send SQL statements between them. The result of an SQL statement is then returned to the user.

MySQL does not support embedded SQL. A CLI must be used to be capable of working with preprogrammed SQL. MySQL has a CLI for all modern programming languages, such as Java, PHP, Python, Perl, Ruby, and Visual Basic. Therefore, the lack of embedded SQL is not a real problem.

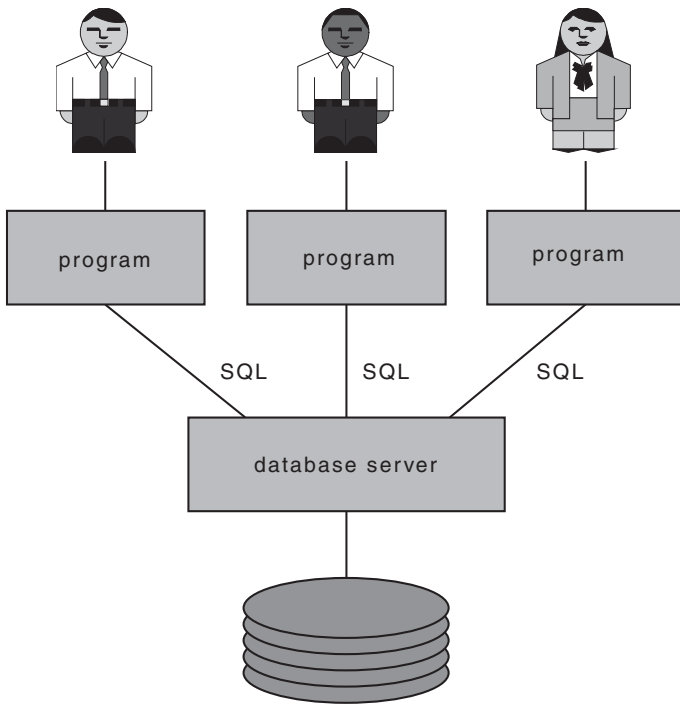


FIGURE 1.9 The user, the application, and the database server are pivotal for the processing of SQL.

1.5 THE HISTORY OF SQL

The history of SQL is closely tied to the history of an IBM project called *System R*. The purpose of this project was to develop an experimental relational database server that bore the same name as the project: System R. This system was built in the IBM research laboratory in San Jose, California. The project was intended to demonstrate that the positive usability features of the relational model could be implemented in a system that satisfied the demands of a modern database server.

The System R project had to solve the problem that no relational database languages existed. A language called *Sequel* was developed as the database language for System R. Designers R. F. Boyce and D. D. Chamberlin wrote the first articles about this language; see [BOYC73a] and [CHAM76]. During the project, the language was renamed SQL because the name *Sequel* conflicted with an existing trademark. (However, the language is still often pronounced as ‘sequel’).



FIGURE 1.10
Don Chamberlin, one
of the designers of SQL

The System R project was carried out in three phases. In the first phase, phase zero (from 1974 to 1975), only a part of SQL was implemented. For example, the join (for linking data from various tables) was not implemented yet, and only a single-user version of the system was built. The purpose of this phase was to see whether implementation of such a system was possible. This phase ended successfully; see [ASTR80].

Phase 1 started in 1976. All the program code written for Phase 0 was put aside for a fresh start. Phase 1 comprised the total system. This meant, among other things, incorporating the multiuser capability and the join. Development of Phase 1 took place between 1976 and 1977.

The final phase evaluated System R. The system was installed at various places within IBM and with a large number of major IBM clients. The evaluation took place in 1978 and 1979. The results of this evaluation are described in [CHAM80], as well as in other publications. The System R project was finished in 1979.

Developers used the knowledge acquired and the technology developed in these three phases to build SQL/DS. SQL/DS was the first commercially available IBM relational database server. In 1981, SQL/DS came onto the market for the operating system DOS/VSE, and the VM/CMS version arrived in 1983. In that same year, DB2 was announced. Currently, DB2 is available for many operating systems.

IBM has published a great deal about the development of System R, which happened at a time when conferences and seminars focused greatly on relational database servers. Therefore, it is not surprising that other companies began to build

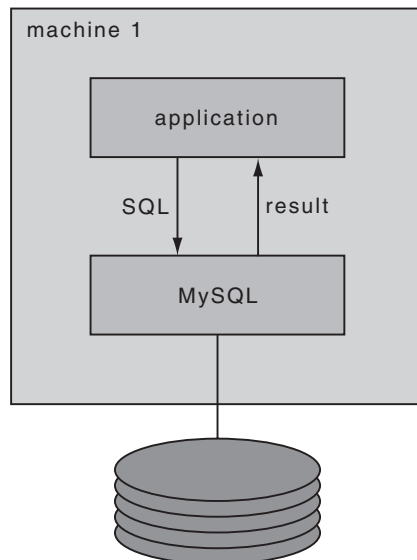
relational systems as well. Some of them, such as Oracle, implemented SQL as the database language. In the last few years, many SQL products have appeared. As a result, SQL is now available for every possible system, large or small. Existing database servers have also been extended to include SQL support.

1.6 FROM MONOLITHIC VIA CLIENT/SERVER TO THE INTERNET

Section 1.4 describes the relationship between the database server MySQL and the calling application. Applications send SQL statements to MySQL to have them processed. The latter processes the statements and returns the results to the application. Finally, the results are presented to the users. It is not necessary for MySQL and the applications to run on the same machine for them to communicate with each other. Roughly, three solutions or architectures are available; among them are the client/server and Internet architectures.

The most simple architecture is the *monolithic architecture* (see Figure 1.11). In a monolithic architecture, everything runs on the same machine. This machine can be a large mainframe, a small PC, or a midrange computer with an operating system such as UNIX or Windows. Because both the application and MySQL run on the same computer, communication is possible through very fast internal communication lines. In fact, this involves two processes that communicate internally.

FIGURE 1.11
The monolithic
architecture



The second architecture is the *client/server architecture*. Several subforms of this architecture exist, but we will not discuss them all here. It is important to realize that in a client/server architecture, the application runs on a different machine than MySQL (see Figure 1.12). This is called working with a *remote database server*. Internal communication usually takes place through a local area network (LAN) and occasionally through a wide area network (WAN). A user could start an application on a PC in Paris and retrieve data from a database located in Sydney. Communication would then probably take place through a satellite link.

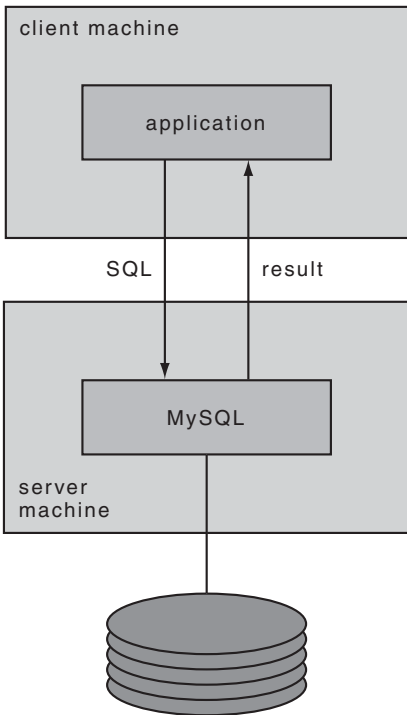


FIGURE 1.12
The client/server
architecture

The third architecture is the *Internet architecture*. In this architecture, the application running in a client/server architecture on the client machine is divided into two parts (see the left part of Figure 1.13). The part that deals with the user, or the user interface, runs on the client machine. The part that communicates with the database server, also called the *application logic*, runs on the *server machine*. In this book, these two parts are called, respectively, the client and the server application.

Probably no SQL statements exist in the *client application*, but there are statements that call the server application. Languages such as HTML, JavaScript, and VBScript are often used for the client application. The call goes via the Internet or an intranet to the second machine; the well-known HyperText Transport Protocol

(HTTP) is mostly used for this. The call comes in at a *web server*. The web server acts as a kind of switchboard operator and knows which call has been sent to which server application.

Next, the call arrives at the server application. The server application sends the needed SQL statements to MySQL. Many server applications run under the supervision of Java application servers, such as WebLogic from Bea Systems and WebSphere from IBM.

MySQL returns the results of the SQL statements. In some way, the server application translates this SQL result to an HTML page and returns the page to the web server. As the switchboard operator, the web server knows the client application to which the HTML answer must be returned.

The right part of Figure 1.13 shows a variant of the Internet architecture in which the server application and MySQL have also been placed on different machines.

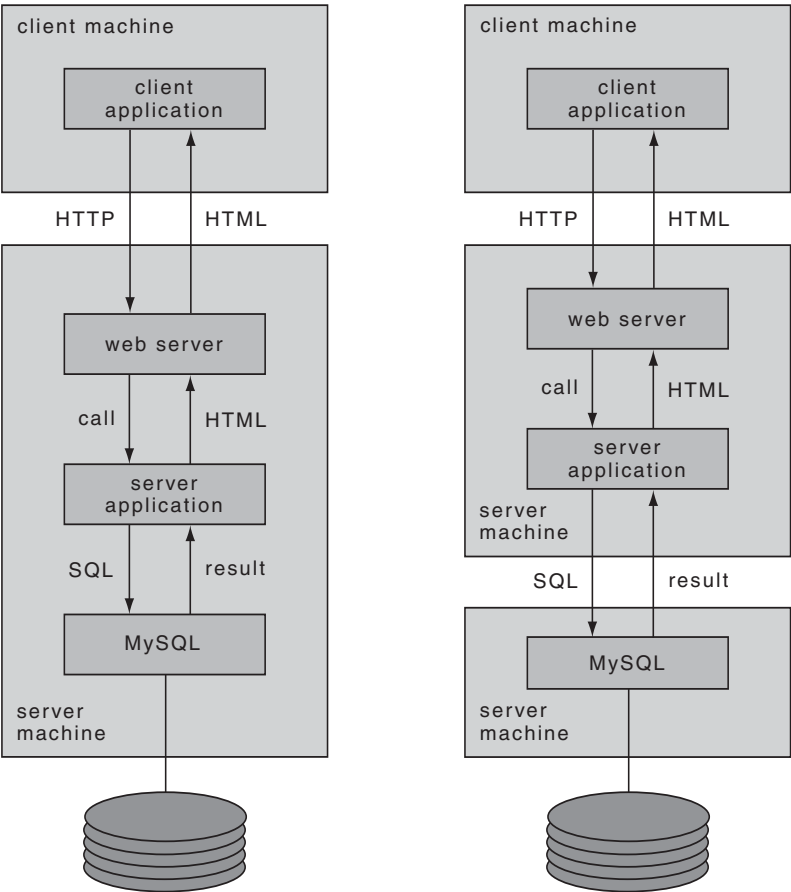


FIGURE 1.13 The Internet architecture

The fact that MySQL and the database are remote is completely transparent to the programmer who is responsible for writing the application and the SQL statements. However, it is not irrelevant. Regarding the language and efficiency aspects of SQL, it is important to know which architecture is used: monolithic, client/server, or Internet. In this book, we will use the first one, but where relevant, we discuss the effect of client/server or Internet architectures.

1.7 STANDARDIZATION OF SQL

As mentioned before, each SQL database server has its own dialect. All these dialects resemble each other, but they are not completely identical. They differ in the statements they support, or some products contain more SQL statements than others; the possibilities of statements can vary as well. Sometimes two products support the same statement, but the result of that statement might vary among products.

To avoid differences among the many database servers from several vendors, it was decided early to define a standard for SQL. The idea was that when the database servers grew too much apart, acceptance by the SQL market would diminish. A standard would ensure that an application with SQL statements would be easier to transfer from one database server to another.

In about 1983, the International Standardization Organization (ISO) and the American National Standards Institute (ANSI) started work on the development of an SQL standard. The ISO is the leading internationally oriented normalization and standardization organization; its objectives include the promotion of international, regional, and national normalization. Many countries have local representatives of the ISO. ANSI is the American branch of the ISO.

After many meetings and several false starts, the first ANSI edition of the SQL standard appeared in 1986. This is described in the document ANSI X3.135-1986, “Database Language SQL.” This *SQL-86 standard* is unofficially called *SQL1*. One year later, the ISO edition, called ISO 9075-1987, “Database Language SQL,” was completed; see [ISO87]. This report was developed under the auspices of Technical Committee TC97. The area of activity of TC97 is described as Computing and Information Processing. Its Subcommittee SC21 caused the standard to be developed. This means that the standards of ISO and ANSI for SQL1 or SQL-86 are identical.

SQL1 consists of two levels. Level 2 comprises the complete document, and Level 1 is a subset of Level 2. This implies that not all specifications of SQL1 belong to Level 1. If a vendor claims that its database server complies with the standard, the supporting level must be stated as well. This is done to improve the support and adoption of SQL1. It means that vendors can support the standard in two phases, first Level 1 and then Level 2.

The SQL1 standard is very moderate with respect to integrity. For this reason, it was extended in 1989 by including, among other things, the concepts of primary and foreign keys. This version of the SQL standard is called *SQL89*. The companion ISO document is called, appropriately, ISO 9075:1989, “Database Language SQL with Integrity Enhancements.” The ANSI version was completed simultaneously.

Immediately after the completion of SQL1 in 1987, the development of a new SQL standard began; see [ISO92]. This planned successor to SQL89 was called *SQL2* because the date of publication was not known at the start. In fact, SQL89 and SQL2 were developed simultaneously. Finally, SQL2 was published in 1992 and replaced SQL89, the current standard at that time. The new SQL92 standard is an expansion of the SQL1 standard. Many new statements and extensions to existing statements have been added. For a complete description of SQL92, see [DATE97].

Just like SQL1, SQL92 has *levels*. The levels have names instead of numbers: *entry*, *intermediate*, and *full*. Full SQL is the complete standard. In terms of functionality, intermediate SQL is a subset of full SQL, and entry SQL is a subset of intermediate SQL. Entry SQL can roughly be compared to SQL1 Level 2, although with some specifications extended. All the levels together can be seen as the rings of an onion; see Figure 1.14. A ring represents a certain amount of functionality. The bigger the ring, the more functionality is defined within that level. When a ring falls within the other ring, it defines a subset of functionality.

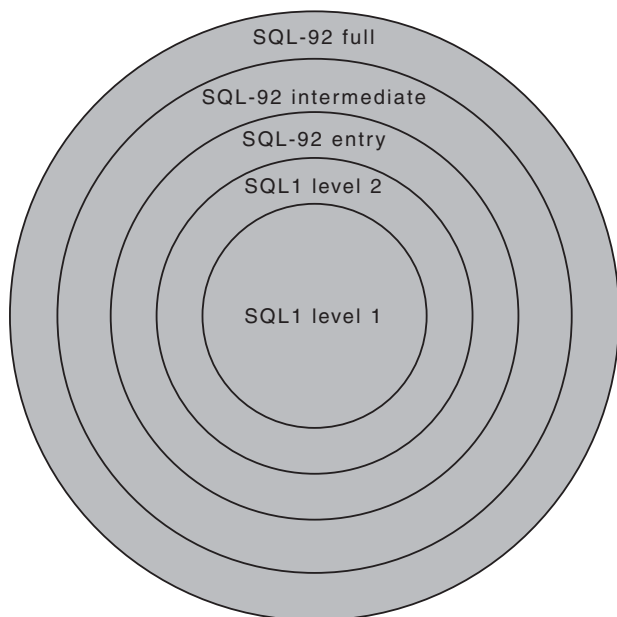


FIGURE 1.14 The various levels of SQL1 and SQL92 represented as rings

At the time of this writing, many available products support entry SQL92. Some even claim to support intermediate SQL92, but not one product supports full SQL92. Hopefully, the support of the SQL92 levels will improve in the coming years.

Since the publication of SQL92, several additional documents have been added that extend the capabilities of the language. In 1995, *SQL/CLI* (Call Level Interface) was published. Later the name was changed to CLI95; the end of this section includes more about CLI95. The following year, *SQL/PSM* (Persistent Stored Modules), or PSM-96, appeared. The most recent addition, PSM96, describes functionality for creating so-called stored procedures. Chapter 31, “Stored Procedures,” deals with this concept extensively. Two years after PSM96, *SQL/OLB* (Object Language Bindings), or OLB-98, was published. This document describes how SQL statements had to be included within the programming language Java.

Even before the completion of SQL92, the development of its successor began: *SQL3*. In 1999, the standard was published and bore the name SQL:1999. To be more in line with the names of other ISO standards, the hyphen that was used in the names of the previous editions was replaced by a colon. And because of the problems around the year 2000, it was decided that 1999 would not be shortened to 99. See [GULU99], [MELT01], and [MELT03] for more detailed descriptions of this standard.

When SQL:1999 was completed, it consisted of five parts: SQL/Framework, SQL/Foundation, SQL/CLI, SQL/PSM, and SQL/Bindings. SQL/OLAP, SQL/MED (Management of External Data), SQL/OLB, SQL/Schemata and SQL/JRT (Routines and Types using the Java Programming Language), and SQL/XML (XML-Related Specifications) were added later, among other things. Thus, the current SQL standard of ISO consists of a series of documents. They all begin with the ISO code 9075. For example, the complete designation of the SQL/Framework is ISO/IEC 9075-1:2003.

Besides the 9075 documents, another group of documents focuses on SQL. The term used for this group is usually *SQL/MM*, short for SQL Multimedia and Application Packages. All these documents bear the ISO code 13249. SQL/MM consists of five parts. SQL/MM Part 1 is the SQL/MM Framework, Part 2 focuses on text retrieval (working with text), Part 3 is dedicated to spatial applications, Part 4 involves still images (such as photos), and Part 5 deals with data mining (looking for trends and patterns in data).

In 2003, a new edition of SQL/Foundation appeared, along with new editions of some other documents, such as SQL/JRT and SQL/Schemata. At this moment, this group of documents can be seen as the most recent version of the international SQL standard. We refer to it by the abbreviation *SQL:2003*.

Other organizations previously worked on the standardization of SQL, including The Open Group (then called the X/Open Group) and the SQL Access Group. The first does not get much attention any longer, so this book does not discuss it.

In July 1989, a number of mainly American vendors of SQL database servers (among them Informix, Ingres, and Oracle) set up a committee called the *SQL Access Group*. The objective of the SQL Access Group is to define standards for the *interoperability* of SQL applications. This means that SQL applications developed using those specifications are portable between the database servers of the associated vendors and that these applications can simultaneously access a number of different database servers. At the end of 1990, the first report of the SQL Access Group was published and defined the syntax of a so-called SQL application interface. The first demonstrations in this field emerged in 1991. Eventually, the ISO adopted the resulting document, and it was published under the name SQL/CLI. This document was mentioned earlier.

The most important technology that is derived from the work of the Open SQL Access Group—and, therefore from SQL/CLI—is Open Database Connectivity (ODBC), from Microsoft.

Finally, an organization called the Object Database Management Group (ODMG) is aimed at the creation of standards for object-oriented databases; see [CATT97]. Part of these standards is a declarative language to query and update databases, called Object Query Language (OQL). It is claimed that SQL has served as a foundation for OQL and, although the languages are not the same, they have a lot in common.

It is correct to say that a lot of time and money has been invested in the standardization of SQL. But is a standard that important? The following practical advantages would accrue if all database servers supported exactly the same standardized database language.

- **Increased portability**—An application could be developed for one database server and could run at another without many changes.
- **Improved interchangeability**—Because database servers speak the same language, they could communicate internally with each other. Applications also could access different databases more simply.
- **Reduced training costs**—Programmers could switch faster from one database server to another because the language would remain the same; they would not have to learn a new database language.
- **Extended life span**—Standardized languages tend to survive longer, and this also applies to the applications written in such languages. COBOL is a good example of this.

MySQL supports a considerable part of the SQL92 standard. Especially since Version 4, MySQL has been extended considerably in this field. Currently, the objective seems to be to develop MySQL more according to the standard. In other words, when the MySQL organization wants to add something new to MySQL and something is written about it in the standard, the group keeps to that standard.

1.8 WHAT IS OPEN SOURCE SOFTWARE?

MySQL is *open source software*. But what is open source software? Most software products that we buy and use could be called closed source software. The source code of this software cannot be adjusted. We do not have access to the source code; what we buy is compiled code. For example, we cannot modify the hyphenation algorithm of Microsoft Word. This code was written by a Microsoft programmer somewhere in Seattle and cannot be changed; it is blocked for everyone. When you want to change something, you have to pass on your demands to Microsoft.

The opposite applies to the source code of open source software. Open source code can actually be modified because the vendor includes the source code. This also applies to the source code of MySQL. When you think that you can improve MySQL or extend its functionality, you go ahead and try. You try to find the part in the source code that you want to improve and apply the desired changes. Next you compile and link the existing code to the code that you just wrote, and you have created an improved version. In short, the source code is open and accessible to you.

You can even go further. When you think your improved code is really good and useful, you can send it to the vendor of the open source software product. The developers then decide whether they want to add your code to the standard code. If they do, others can enjoy your work in the future. If they don't, you can become such a vendor yourself, as long as you provide your new source code publicly. So either way, an open source license ensures that open source software is improved and is spread into the world.

In short, open source software—therefore, also MySQL—is changeable. That is easy to understand. Most open source software is also free to use. However, when we talk about selling software that includes open source software, it becomes a different story. MySQL is supplied according to the use and payment rules recorded in the GNU General Public License (GPL). For details, refer to the documentation of MySQL; we recommend that you study this carefully.

1.9 THE HISTORY OF MySQL

At first, MySQL was not intended to be a commercial product. A new application had to be written that would access index sequential files. Normally, a programmer has to use a very simplistic interface to manipulate the data in such files. Much code has to be written, and that surely does not help the productivity of the programmers. The developers of this application wanted to use an SQL interface as interface to these files.

This need faced the final founders of MySQL: David Axmark, Allan Larsson, and Michael “Monty” Widenius. They decided to search the market for a product that already offered that SQL interface. They found a product called *Mini SQL*, often shortened to *mSQL*. This product still is supplied by the Australian Hughes Technologies.

After trying out this product, the developers felt that Mini SQL was not powerful enough for their application. They decided to develop a product comparable to Mini SQL themselves. With that, MySQL was born. However, they liked the interface of Mini SQL, which is why the interfaces of MySQL and Mini SQL still resemble each other.

Initially, the company MySQL AB was founded in Sweden, and the initial development was done there as well. Nowadays, the developers can be found all over the world, from the United States to Russia. This is an example of a modern company that relies heavily on technologies such as the Internet and e-mail and on the advantages of open source software to develop its database server.

Version 3.11.0, the first version shown to the outside world, was launched in 1996. Before that, only the developers themselves used MySQL. From the beginning, it was an open source product. Since 2000, the product has been released according to the rules specified in the GPL.

Only three years after the introduction, in 1999, the company MySQL AB was founded. Before that, a somewhat informally operating group of developers managed the software.

This book describes Version 5.0.18 of MySQL, which was released in the summer of 2006. Much has changed since that first commercial version—in particular, the SQL dialect has been extended considerably. For years, much has been done to bring MySQL more in line with the SQL92 standard. That also has increased the portability between MySQL on one hand and other SQL database servers, such as DB2 from IBM, SQL Server from Microsoft, and Oracle10g from Oracle, on the other hand.

Despite the extensions, many customers still use the SQL dialect of Version 3, even when they run Versions 4 or 5. The consequence of this restriction is that they do not use the full power of MySQL. Restricting yourself with respect to SQL leads to unnecessarily complex applications. Many lines of code can be reduced to one simple SQL statement.

Finally, how MySQL got its name has remained a mystery for a long time. However, Monty, one of the founders, has admitted that his eldest daughter is called My.

1.10 THE STRUCTURE OF THIS BOOK

This chapter concludes by describing the structure of this book. Because of the many chapters in the book, we divided it into sections.

Part I, “Introduction,” consists of several introductory topics and includes this chapter. Chapter 2, “The Tennis Club Sample Database,” contains a detailed description of the database used in most of the examples and exercises. This database is modeled on the administration of a tennis club’s competitions. Chapter 4, “SQL in a Nutshell,” gives a general overview of SQL. After reading this chapter, you should have a general overview of the capabilities of SQL and a good idea of what awaits you in the rest of this book.

Part II, “Querying and Updating Data,” focuses completely on querying and updating tables. It is largely devoted to the `SELECT` statement. Many examples illustrate all its features. We devote a great deal of space to this `SELECT` statement because this is the statement most often used and because many other statements are based on it. Chapter 19, “Working with XML Documents,” describes how existing database data can be updated and deleted, and how new rows can be added to tables.

Part III, “Creating Database Objects,” describes the creation of *database objects*. The term *database object* is the generic name for all objects from which a database is built. For instance, this chapter discusses tables; primary, alternate, and foreign keys; indexes; and views. This part also describes data security.

Part IV, “Procedural Database Objects,” describes stored procedures, stored functions, triggers, and events. Stored procedures and stored functions are pieces of code stored in the database that can be called from applications. Triggers are pieces of code as well, but they are invoked by MySQL itself, for example, to perform checks or to update data automatically. Informally, events are triggers that are automatically started on a certain time of the day.

Part V, “Programming with SQL,” deals with programming in SQL. MySQL can be called from many programming languages; those used most are PHP, Python, and Perl. This part uses PHP to illustrate how SQL statements are embedded inside a programming language. The following concepts are explained in this part: transaction, savepoint, rollback of transactions, isolation level, and repeatable read.

The book ends with a number of appendices and an index. Appendix A, “Syntax of SQL,” contains the definitions of all the SQL statements discussed in the book. Appendix B, “Scalar Functions,” describes all the functions that SQL supports. Appendix C, “System Variables,” lists all the system variables, and Appendix D, “Bibliography,” contains a list of references.

The Tennis Club Sample Database

2.1 INTRODUCTION

This chapter describes a database that a tennis club could use to record its players' progress in a competition. Most of the examples and exercises in this book are based on this database, so you should study it carefully.

2.2 DESCRIPTION OF THE TENNIS CLUB

The tennis club was founded in 1970. From the beginning, some administrative data was stored in a database. This database consists of the following tables:

- PLAYERS
- TEAMS
- MATCHES
- PENALTIES
- COMMITTEE_MEMBERS

The PLAYERS table contains data about players who are members of the club, such as names, addresses, and dates of birth. Players can join the club only at the first of January of a year. Players cannot join the club in the middle of the year.

The PLAYERS table contains no historical data. Any player who gives up membership disappears from the table. If a player moves, the new address overwrites the old address. In other words, the old address is not retained anywhere.

The tennis club has two types of members: *recreational players* and *competition players*. The first group plays matches only among themselves (that is, no matches

against players from other clubs). The results of these friendly matches are not recorded. Competition players play in teams against other clubs, and the results of these matches are recorded. Regardless of whether he or she plays competitively, each player has a unique number assigned by the club. Each competition player must also be registered with the tennis league, and this national organization gives each player a unique *league number*. This league number usually contains digits, but it can also consist of letters. If a competition player stops playing in the competition and becomes a recreational player, his or her league number correspondingly disappears. Therefore, recreational players have no league number, but they do have a player number.

The club has a number of teams taking part in competitions. The captain of each team and the division in which it is currently competing are recorded. It is not necessary for the captain to have played a match for the team. It is possible for a certain player to be a captain of two or more teams at a certain time. Again, this table records no historical data. If a team is promoted or relegated to another division, the new information simply overwrites the record. The same goes for the captain of the team; when a new captain is appointed, the number of the former captain is overwritten.

A team consists of a number of players. When a team plays against a team from another tennis club, each player of that team plays against a player of the opposing team (for the sake of simplicity, assume that matches in which couples play against each other, the so-called doubles and mixes, do not occur). The team for which the most players win their matches is the winner.

A team does not always consist of the same people, and reserves are sometimes needed when the regular players are sick or on vacation. A player can play matches for several teams. So when we say “the players of a team,” we mean the players who have played at least one match in that team. Again, only players with league numbers are allowed to play official matches.

Each match consists of a number of *sets*. The player who wins the most sets is the winner. Before the match begins, it is agreed how many sets must be won to win the match. Generally, the match stops after one of the two players has won two or three sets. Possible end results of a tennis match are 2–1 or 2–0 if play continues until one player wins two sets (best of three), or 3–2, 3–1, or 3–0 if three sets need to be won (best of five). A player either wins or loses a match; a draw is not possible. The MATCHES table records for each match separately which player was in the match and for which team he played. In addition, it records how many *sets* the player won and lost. From this, we can conclude whether the player won the match.

If a player behaves badly (arrives late, behaves aggressively, or does not show up) the league imposes a penalty in the form of a fine. The club pays these fines and

records them in a PENALTIES table. As long as the player continues to play competitively, the record of all his or her penalties remains in this table.

If a player leaves the club, all his or her data in the five tables is destroyed. If the club withdraws a team, all data for that team is removed from the TEAMS and MATCHES tables. If a competition player stops playing matches and becomes a recreational player again, all matches and penalty data is deleted from the relevant tables.

Since January 1, 1990, a COMMITTEE_MEMBERS table has kept information about who is on the committee. Four positions exist: chairman, treasurer, secretary, and general member. On January 1 of each year, a new committee is elected. If a player is on the committee, the beginning and ending dates of his or her committee are recorded. If someone is still active, the end date remains open. Figure 2.1 shows which player was on the committee in which period.

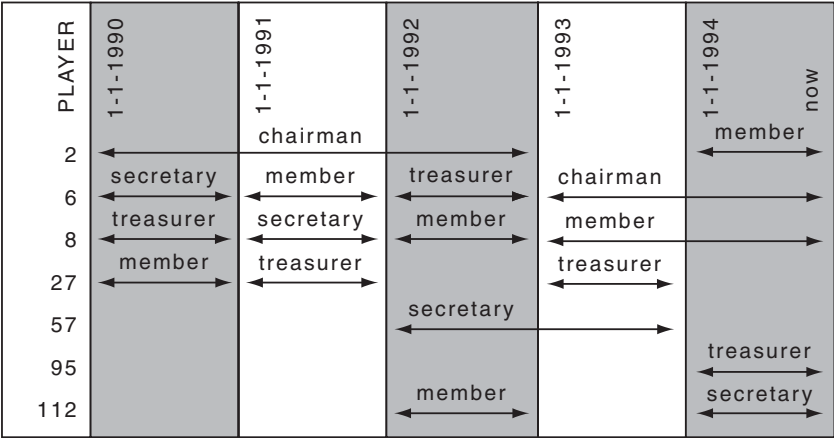


FIGURE 2.1 Which player occupied which position on the committee in which period?

Following is a description of the columns in each of the tables.

PLAYERS	
PLAYERNO	Unique player number assigned by the club.
NAME	Surname of the player, without initials.
INITIALS	Initials of the player. (No full stops or spaces are used.)
BIRTH_DATE	Date on which the player was born.
SEX	Sex of the player: M(ale) or F(emale).

continues

JOINED	Year in which the player joined the club. (This value cannot be smaller than 1970, the year in which the club was founded.)
STREET	Name of the street on which the player lives.
HOUSENO	Number of the house.
POSTCODE	Postcode.
TOWN	Town or city in which the player lives. Assume in this example that place-names are unique for town or cities; in other words, there can never be two towns with the same name.
PHONENO	Area code followed by a hyphen and then the subscriber's number.
LEAGUENO	League number assigned by the league; a league number is unique.
TEAMS	
TEAMNO	Unique team number assigned by the club.
PLAYERNO	Player number of the player who captains the team. In principle a player may captain several teams.
DIVISION	Division in which the league has placed the team.
MATCHES	
MATCHNO	Unique match number assigned by the club.
TEAMNO	Number of the team.
PLAYERNO	Number of the player.
WON	Number of sets that the player won in the match.
LOST	Number of sets that the player lost in the match.
PENALTIES	
PAYMENTNO	Unique number for each penalty the club has paid. The club assigns this number.
PLAYERNO	Number of the player who has incurred the penalty.
PAYMENT_DATE	Date on which the penalty was paid. The year of this date should not be earlier than 1970, the year in which the club was founded.
AMOUNT	Amount in dollars incurred for the penalty.
COMMITTEE_MEMBERS	
PLAYERNO	Number of the player.
BEGIN_DATE	Date on which the player became an active member of the committee. This date should not be earlier than January 1, 1990, because this is the date on which the club started to record this data.
END_DATE	Date on which the player resigned his position in the committee. This date should not be earlier than the BEGIN_DATE but can be absent.
POSITION	Name of the position.

2.3 THE CONTENTS OF THE TABLES

The contents of the tables are shown here. These rows of data form the basis of most of the examples and exercises. Some of the column names in the PLAYERS table have been shortened because of space constraints.

The PLAYERS table:

PLAYERNO	NAME	INIT	BIRTH_DATE	SEX	JOINED	STREET	...
2	Everett	R	1948-09-01	M	1975	Stoney Road	...
6	Parmenter	R	1964-06-25	M	1977	Haseltine Lane	...
7	Wise	GWS	1963-05-11	M	1981	Edgecombe Way	...
8	Newcastle	B	1962-07-08	F	1980	Station Road	...
27	Collins	DD	1964-12-28	F	1983	Long Drive	...
28	Collins	C	1963-06-22	F	1983	Old Main Road	...
39	Bishop	D	1956-10-29	M	1980	Eaton Square	...
44	Baker	E	1963-01-09	M	1980	Lewis Street	...
57	Brown	M	1971-08-17	M	1985	Edgecombe Way	...
83	Hope	PK	1956-11-11	M	1982	Magdalene Road	...
95	Miller	P	1963-05-14	M	1972	High Street	...
100	Parmenter	P	1963-02-28	M	1979	Haseltine Lane	...
104	Moorman	D	1970-05-10	F	1984	Stout Street	...
112	Bailey	IP	1963-10-01	F	1984	Vixen Road	...

The PLAYERS table (continued):

PLAYERNO	...	HOUSENO	POSTCODE	TOWN	PHONENO	LEAGUENO
2	...	43	3575NH	Stratford	070-237893	2411
6	...	80	1234KK	Stratford	070-476537	8467
7	...	39	9758VB	Stratford	070-347689	?
8	...	4	6584RO	Inglewood	070-458458	2983
27	...	804	8457DK	Eltham	079-234857	2513
28	...	10	1294QK	Midhurst	071-659599	?
39	...	78	9629CD	Stratford	070-393435	?
44	...	23	4444LJ	Inglewood	070-368753	1124
57	...	16	4377CB	Stratford	070-473458	6409
83	...	16A	1812UP	Stratford	070-353548	1608
95	...	33A	57460P	Douglas	070-867564	?
100	...	80	1234KK	Stratford	070-494593	6524
104	...	65	9437AO	Eltham	079-987571	7060
112	...	8	6392LK	Plymouth	010-548745	1319

The TEAMS table:

TEAMNO	PLAYERNO	DIVISION
1	6	first
2	27	second

The MATCHES table:

MATCHNO	TEAMNO	PLAYERNO	WON	LOST
1	1	6	3	1
2	1	6	2	3
3	1	6	3	0
4	1	44	3	2
5	1	83	0	3
6	1	2	1	3
7	1	57	3	0
8	1	8	0	3
9	2	27	3	2
10	2	104	3	2
11	2	112	2	3
12	2	112	1	3
13	2	8	0	3

The PENALTIES table:

PAYMENTNO	PLAYERNO	PAYMENT_DATE	AMOUNT
1	6	1980-12-08	100.00
2	44	1981-05-05	75.00
3	27	1983-09-10	100.00
4	104	1984-12-08	50.00
5	44	1980-12-08	25.00
6	8	1980-12-08	25.00
7	44	1982-12-30	30.00
8	27	1984-11-12	75.00

The COMMITTEE_MEMBERS table:

PLAYERNO	BEGIN_DATE	END_DATE	POSITION
2	1990-01-01	1992-12-31	Chairman
2	1994-01-01	?	Member
6	1990-01-01	1990-12-31	Secretary
6	1991-01-01	1992-12-31	Member
6	1992-01-01	1993-12-31	Treasurer
6	1993-01-01	?	Chairman
8	1990-01-01	1990-12-31	Treasurer
8	1991-01-01	1991-12-31	Secretary
8	1993-01-01	1993-12-31	Member
8	1994-01-01	?	Member
27	1990-01-01	1990-12-31	Member
27	1991-01-01	1991-12-31	Treasurer
27	1993-01-01	1993-12-31	Treasurer
57	1992-01-01	1992-12-31	Secretary
95	1994-01-01	?	Treasurer
112	1992-01-01	1992-12-31	Member
112	1994-01-01	?	Secretary

2.4 INTEGRITY CONSTRAINTS

Of course, the contents of the tables must satisfy a number of integrity constraints. For example, two players may not have the same player number, and every player number in the PENALTIES table must also appear in the MATCHES table. This section lists all the applicable integrity constraints.

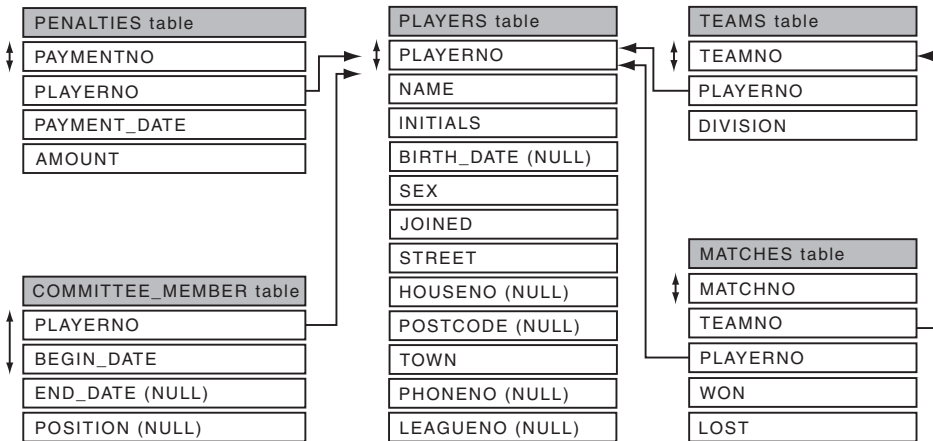


FIGURE 2.2 Diagram of the relationships between the tennis club database tables

A primary key has been defined for each table. The following columns are the primary keys for their respective tables. Figure 2.2 contains a diagram of the database. A double-headed arrow at the side of a column (or combination of columns) indicates the primary key of a table:

- PLAYERNO of PLAYERS
- TEAMNO of TEAMS
- MATCHNO of MATCHES
- PAYMENTNO of PENALTIES
- PLAYERNO plus BEGIN_DATE of COMMITTEE_MEMBERS

The example database has no alternate keys. The LEAGUENO column in the PLAYERS table looks like one but isn't. All the values are unique, but the column also allows null values and, therefore, can be no alternate key.

The database supports five foreign keys. In Figure 2.2, single-headed arrows show the foreign keys; these run from one table to another (this notation, in which the arrows point to the primary key, is used in [DATE95] and elsewhere). The foreign keys are as follows:

- **From TEAMS to PLAYERS**—Each captain of a team is also a player. The set of player numbers from the TEAMS table is a subset of the set of player numbers from the PLAYERS table.
- **From MATCHES to PLAYERS**—Each player who competes for a particular team must appear in the PLAYERS table. The set of player numbers from the MATCHES table is a subset of the set of player numbers from the PLAYERS table.
- **From MATCHES to TEAMS**—Each team that appears in the MATCHES table must also be present in the TEAMS table because a player can compete for only a registered team. The set of team numbers from the MATCHES table is a subset of the set of team numbers from the TEAMS table.
- **From PENALTIES to PLAYERS**—A penalty can be imposed on only players appearing in the PLAYERS table. The set of player numbers from the PENALTIES table is a subset of the set of player numbers from the PLAYERS table.
- **From COMMITTEE_MEMBERS to PLAYERS**—Each player who is or was a member of the committee must also be present in the PLAYERS table. The set of player numbers from the COMMITTEE_MEMBERS table is a subset of the set of player numbers from the PLAYERS table.

The following integrity constraints also hold:

- Two players cannot have identical league numbers.
- The year of birth of a player must be earlier than the year in which he or she joined the club.
- The sex of a player should always be M or F.
- The year in which the player joined the club should be greater than 1969 because the tennis club was founded in 1970.
- The postcode must always be a code of six characters.
- The division of a team can be nothing but first or second.
- Both the columns WON and LOST must have a value between 0 and 3.
- The payment date should be January 1, 1970, or later.
- Each penalty amount must always be greater than zero.
- The begin date in the COMMITTEE_MEMBERS table should always be later than or equal to January 1, 1990, because recording of this data was started on that day.
- The end date on which the player ended service as a committee member must always be later than the begin date.

Installing the Software

3.1 INTRODUCTION

As already mentioned in the preface, we advise that you replay the examples in this book and do the exercises. This will definitely improve your knowledge of MySQL and pleasure in reading this book.

This chapter describes where to find the required software and the information needed to install all the software necessary. It also indicates how to download the code for the many examples. For practical reasons, we refer frequently to the book's web site. Here you will find useful information.

3.2 DOWNLOADING MySQL

You can download MySQL free from the web site of the vendor, www.mysql.com, where you will find the software for many different operating systems. Choose the version that suits you best. This book assumes that you will be using Version 5.0 or higher. Of course, you can also process the SQL statements in this book with newer versions of MySQL.

This book deliberately does not indicate where on the web site you can find the software and the documentation. The structure of this web site changes rather frequently, so this book would contain out-of-date descriptions too quickly.

3.3 INSTALLATION OF MYSQL

On the vendor's web site, you will find documentation describing how to install MySQL. You can use this documentation or visit the book's web site: www.r20.nl. Here you will find a detailed plan that describes the installation step by step, including many screen shots. This plan might be easier to understand than the vendor's documentation.

If you have comments on the installation description, please let me know so we can improve the web site if necessary.



NOTE

We have deliberately chosen not to include the installation process in this book because it differs for each operating system and can change with every new version of MySQL.

3.4 INSTALLING A QUERY TOOL

This book assumes that you will use a query tool such as MySQL Query Browser, SQLyog, or WinSQL to process your SQL statements. However, these are not database servers, but programs that enable you to simply enter SQL statements interactively under Windows or Linux. They work together with MySQL and most other database servers. You also can download most of these query tools for free from the vendor's web site. Again, as with MySQL, we strongly recommend that you install one of those query tools.

3.5 DOWNLOADING SQL STATEMENTS FROM THE WEB SITE

As mentioned in the preface, the accompanying web site contains all the SQL statements used in this book. This section briefly describes how you can download them. This is a good time to do so because you'll need these statements to create the sample database.

The URL of the book's web site is www.r20.nl. The statements are stored in simple text files; by cutting and pasting, you can easily copy them to any product. You can open them with any text editor.

A separate file exists for each chapter, as clearly indicated on the web site. In the file, you will find in front of each SQL statement an identification to help you search for them. For example, Example 7.1 (the first example in Chapter 7, “SELECT Statement: The FROM Clause,”) has this as its identification:

Example 7.1:

Likewise, the following text is included to find Answer 12.6:

Answer 12.6:

3.6 READY?

If all went well, you have now installed MySQL and a query tool. If you want, you can start to play with SQL. However, the sample database is missing. The next chapter describes how to create that database.

This page intentionally left blank

SQL in a Nutshell

4.1 INTRODUCTION

This chapter uses examples to illustrate the capabilities of the database language SQL. We discuss most SQL statements briefly; other chapters describe the details and all the features. The purpose of this chapter is to give you a feeling of what SQL looks like and what this book covers.

The first sections also explain how to create the sample database. Be sure to execute the statements from these sections because almost all the examples and exercises in the rest of this book are based upon this database.

4.2 LOGGING ON TO THE MYSQL DATABASE SERVER

To do anything with SQL (this applies to creating the sample database as well), you must log on to the MySQL database server. MySQL requires that applications identify themselves before manipulating the data in the database. In other words, the user needs to *log on* by using an application. Identification is done with the help of a *user name* and a *password*. Therefore, this chapter begins by describing how to log on to MySQL.

First, you need a user name. However, to create a user (with a name and password), you must log on first—a classic example of a chicken-and-egg problem. To end this deadlock, most database servers create several users during the installation procedure. Otherwise, it would be impossible to log on after the installation. One of these users is called *root* and has an identical password (if you have followed the installation procedure described in the previous chapter).

How logging on really takes place depends on the application that is used. For example, with the query tool WinSQL, the logon screen looks similar to Figure 4.1.

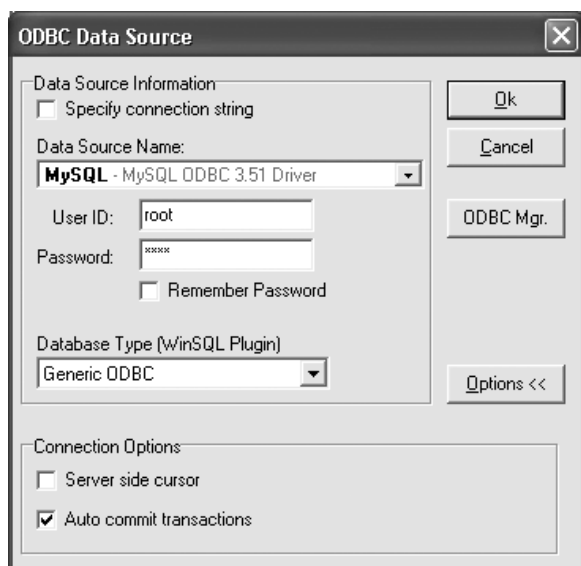


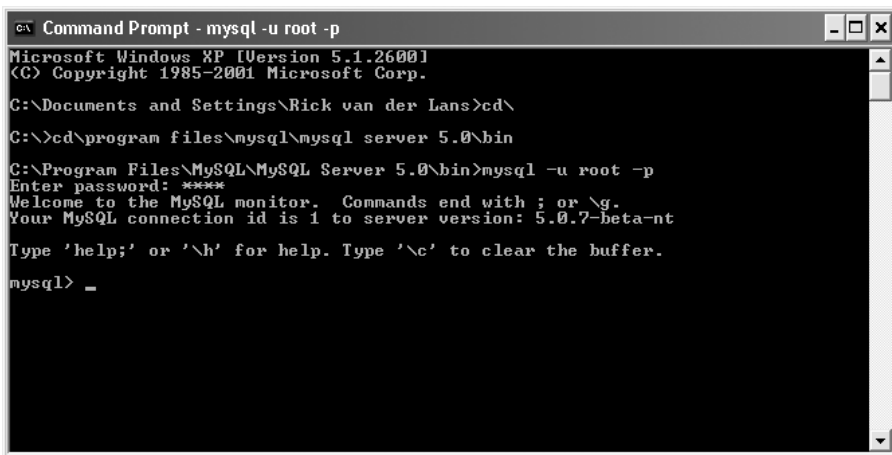
FIGURE 4.1 The logon screen of WinSQL

The user name is entered in the User ID text box, and the password in the Password text box. In both cases, you type *root*. For security reasons, the password characters appear as asterisks. User names and passwords are case sensitive, so be sure you type them correctly—not with capitals. After you enter the name and password, you can log on and start entering SQL statements.

When you use the client application called *mysql* that is included with MySQL, the process of logging on looks different but is still comparable (see Figure 4.2). The code *-u* stands for *user*, behind which the user name (*root*) is specified, followed by the code *-p*. Next the application wants to know the password. Later sections explain this in more detail.

The web site of this book contains detailed information about how to log on with different programs.

After you have logged on successfully with the users that are created during the installation procedure, you can introduce new users and create new tables.



```
Command Prompt - mysql -u root -p
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Rick van der Lans>cd\
C:\>cd\program files\mysql\mysql server 5.0\bin
C:\Program Files\MySQL\MySQL Server 5.0\bin>mysql -u root -p
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.7-beta-nt
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> _
```

FIGURE 4.2 Logging on with mysql

4.3 CREATING NEW SQL USERS

Section 1.4 described the concept of a user and also mentioned briefly the respective roles of users and applications. A user starts up an application. This application passes SQL statements to MySQL that processes them. A user can enter these SQL statements “live” (interactive SQL), or they can be included in the application code (preprogrammed SQL).

A clear distinction should be made between the real, *human* user and the user name used to log on. To avoid confusion, we call the latter the *SQL user*. SQL users can be granted *privileges*. A privilege is a specification indicating what a certain SQL user can do. For example, one user might be allowed to create new tables, another might be authorized to update existing tables, and a third might be able to only query tables. The relationship between human users and SQL users can be one-to-one, but that is not required. A human user is allowed to log on under different SQL user names with different privileges. Additionally, an entire group of human users is allowed to use the same SQL user name with the same privileges. Therefore, the relationship between users and SQL users is a many-to-many relationship. You need to define these relationships.

So to be able to log on, you need to have an SQL user. Several SQL users have already been created during the installation procedure, to prevent the chicken-and-egg problem. Therefore, you do not need to create one. However, if you want to create your own SQL users, you can do that with a special SQL statement.

Imagine that we log on with the SQL user called `root`. Next, we can use the *CREATE USER statement* to create our own new SQL users. We give a new SQL user a name and also a password.

Example 4.1: Introduce a new user called `BOOKSQL` with the password `BOOKSQLPW`.

```
CREATE USER 'BOOKSQL'@'localhost' IDENTIFIED BY 'BOOKSQLPW'
```

Explanation: The name of the new SQL user is created with the specification `'BOOKSQL'@'localhost'`. Another chapter explains the meaning of `localhost`. The statement ends with the password, which, in this case, is `BOOKSQLPW`. Make sure that quotation marks surround the user name, the term `localhost`, and the password.

When an application logs on to MySQL with an SQL user name, a so-called *connection* is started. A connection is a unique link between the application and the MySQL database server for the specific SQL user. It is like a telephone cable between that application and MySQL. The privileges of the SQL user determine what the user is allowed to send over the cable. Through the connection, the user has access to all the databases that the database server manages. A new SQL user is allowed to log on, but this user does not have any other privileges yet. We need to grant those privileges to `BOOKSQL` first with the *GRANT statement*.

The `GRANT` statement has extensive features. Chapter 28, “Users and Data Security,” discusses this statement and related topics.” However, to get you started, the next example contains the statement that grants the new SQL user called `BOOKSQL` enough privileges to create tables and manipulate them afterward.

Example 4.2: Give the SQL user `BOOKSQL` the privileges to create and manipulate tables.

```
GRANT ALL PRIVILEGES
ON      *.*
TO      'BOOKSQL'@'localhost'
WITH    GRANT OPTION
```

`BOOKSQL` now can log on and execute all the statements in the following chapters.

Note: The rest of the book assumes that you log on as user `BOOKSQL` with the password `BOOKSQLPW` and that you have sufficient privileges.

4.4 CREATING DATABASES

Section 1.2 defined the concept of a database. Using this definition, a database acts as a container for a set of tables. For MySQL, each table must also be created within an existing database. Therefore, when you want to build a table, you first need to create a database.

Example 4.3: Create a database with the name TENNIS for the tables of the tennis club.

```
CREATE DATABASE TENNIS
```

Explanation: After this `CREATE DATABASE` *statement* is processed, the database exists but is still empty. This book assumes that you have logged on as `BOOKSQL` before you enter this statement.

4.5 SELECTING THE CURRENT DATABASE

A MySQL database server can offer access to more than one database. When a user has opened a connection with MySQL and wants, for example, to create new tables or query existing tables, the user must specify the database he wants to work with. This is called the *current database*. Only one current database can exist, and

If no current database has been specified, you still can manipulate tables. In addition, you can access tables from a database other than the current database. For both situations, you must explicitly specify the database in which those tables reside.

To make a specific database current, MySQL supports the *USE statement*.

Example 4.4: Make TENNIS the current database.

```
USE TENNIS
```

Explanation: This statement can also be used to “jump” from one database to another.

After processing a `CREATE DATABASE` statement (see the earlier section), the created database does *not* automatically become the current database—an extra `USE` statement is needed for that.

No database is current when you log on using the technique described earlier. As an alternative to the *USE statement*, you can make a database current by specifying it when you log on.

```
mysql -u BOOKSQL -p TENNIS
```

The rest of the book assumes that you log on as user BOOKSQL with the password BOOKSQLPW, that you have sufficient privileges, and that the TENNIS database is the current database.

4.6 CREATING TABLES

Databases in MySQL are made up of database objects. The best-known and most important database object is probably the table. The *CREATE TABLE* statement is used to develop new tables. The next example contains the *CREATE TABLE* statements that are needed to create the tables from the sample database.

Example 4.5: Create the five tables that form the sample database.

```
CREATE TABLE PLAYERS
(PLYERNO    INTEGER    NOT NULL,
NAME        CHAR(15)   NOT NULL,
INITIALS    CHAR(3)    NOT NULL,
BIRTH_DATE  DATE       ,
SEX         CHAR(1)    NOT NULL,
JOINED      SMALLINT   NOT NULL,
STREET      VARCHAR(30) NOT NULL,
HOUSENO     CHAR(4)    ,
POSTCODE    CHAR(6)    ,
TOWN        VARCHAR(30) NOT NULL,
PHONENO     CHAR(13)   ,
LEAGUENO    CHAR(4)    ,
PRIMARY KEY (PLYERNO) )

CREATE TABLE TEAMS
(TTEAMNO    INTEGER    NOT NULL,
PLYERNO     INTEGER    NOT NULL,
DIVISION    CHAR(6)    NOT NULL,
PRIMARY KEY (TTEAMNO) )

CREATE TABLE MATCHES
(MATCHNO    INTEGER    NOT NULL,
TEAMNO      INTEGER    NOT NULL,
PLYERNO     INTEGER    NOT NULL,
WON         SMALLINT   NOT NULL,
LOST        SMALLINT   NOT NULL,
PRIMARY KEY (MATCHNO) )
```

```
CREATE TABLE PENALTIES
(PAYMENTNO      INTEGER      NOT NULL,
 PLAYERNO      INTEGER      NOT NULL,
 PAYMENT_DATE   DATE         NOT NULL,
 AMOUNT        DECIMAL(7,2) NOT NULL,
 PRIMARY KEY    (PAYMENTNO) )

CREATE TABLE COMMITTEE_MEMBERS
(PAYERNO      INTEGER      NOT NULL,
 BEGIN_DATE    DATE         NOT NULL,
 END_DATE      DATE         ,
 POSITION       CHAR(20)     ,
 PRIMARY KEY    (PLAYERNO, BEGIN_DATE))
```

Explanation: MySQL does not require the statements to be entered in this exact way. This book uses a certain layout style for all SQL statements, to make them easier to read. However, MySQL does not care whether everything is written neatly in a row (still separated by spaces or commas, of course) or nicely below each other.

As indicated in Chapter 2, “The Tennis Club Sample Database,” several integrity constraints apply for these tables. We excluded most of them here because we do not need them in the first two parts of this book. Chapter 21, “Specifying Integrity Constraints,” explains all the integrity rules in SQL.

With a `CREATE TABLE` statement, several properties are defined, including the name of the table, the columns of the table, and the primary key. The name of the table is specified first: `CREATE TABLE PLAYERS`. The columns of a table are listed between brackets. For each column name, a data type is specified, as in `CHAR`, `SMALLINT`, `INTEGER`, `DECIMAL`, or `DATE`. The data type defines the type of value that may be entered into the specific column. The next section explains the specification `NOT NULL`.

Figure 2.2 shows the primary key of the tables, among other things. A primary key of a table is a column (or combination of columns) in which every value can appear only once. By defining the primary key in the `PLAYERS` table, we indicate that each player number can appear only once in the `PLAYERNO` column. A primary key is a certain type of integrity constraint. In SQL, primary keys are specified within the `CREATE TABLE` statement with the words `PRIMARY KEY`. This is one of two ways to specify a primary key. After listing all the columns, `PRIMARY KEY` is specified followed by the column or columns belonging to that primary key. Chapter 21 discusses the other way to specify a primary key.

It is not always necessary to specify primary keys for a table, but it is important. Chapter 21 explains why. For now, we advise you to define a primary key for each table you create.

In the definition of a column, you are allowed to specify `NOT NULL`. This means that every row of the column *must* be filled. In other words, null values are not allowed in a `NOT NULL` column. For example, each player must have a `NAME`, but a `LEAGUENO` is not required.

4.7 POPULATING TABLES WITH DATA

The tables have been created and can now be filled with data. For this, we use `INSERT` statements.

Example 4.6: Fill all tables from the sample database with data. See Section 2.3 for a listing of all data.

For the sake of convenience, only two examples of `INSERT` statements are given for each table. At the web site of the book, you will find all the `INSERT` statements.

```
INSERT INTO PLAYERS VALUES
(6, 'Parmenter', 'R', '1964-06-25', 'M', 1977,
'Haseltine Lane', '80', '1234KK', 'Stratford',
'070-476537', '8467')
```

```
INSERT INTO PLAYERS VALUES
(7, 'Wise', 'GWS', '1963-05-11', 'M', 1981,
'Edgecombe Way', '39', '9758VB', 'Stratford',
'070-347689', NULL)
```

```
INSERT INTO TEAMS VALUES (1, 6, 'first')
```

```
INSERT INTO TEAMS VALUES (2, 27, 'second')
```

```
INSERT INTO MATCHES VALUES (1, 1, 6, 3, 1)
```

```
INSERT INTO MATCHES VALUES (4, 1, 44, 3, 2)
```

```
INSERT INTO PENALTIES VALUES (1, 6, '1980-12-08', 100)
```

```
INSERT INTO PENALTIES VALUES (2, 44, '1981-05-05', 75)
```

```
INSERT INTO COMMITTEE_MEMBERS VALUES
(6, '1990-01-01', '1990-12-31', 'Secretary')
```

```
INSERT INTO COMMITTEE_MEMBERS VALUES
(6, '1991-01-01', '1992-12-31', 'Member')
```

Explanation: Each statement corresponds to one (new) row in a table. After the term `INSERT INTO`, the table name is specified, and the values for the new row come after `VALUES`. Each row consists of one or more values. Different kinds of values may be used. For example, numeric and alphanumeric values, dates, and times exist.

Each alphanumeric value, such as `Parmenter` and `Stratford` (see the first `INSERT` statement), must be enclosed in single quotation marks. The (column) values are separated by commas. Because MySQL remembers the sequence in which the columns were specified in the `CREATE TABLE` statement, the system also knows the column to which every value corresponds. For the `PLAYERS` table, therefore, the first value is `PLAYERNO`, the second value is `NAME`, and the last value is `LEAGUENO`.

Specifying dates and times is more difficult than specifying numeric and alphanumeric values because they have to adhere to certain rules. A date such as December 8, 1980, must be specified as `'1980-12-08'`. This form of expression, described in detail in Section 5.2.5, turns an alphanumeric value into a correct date. However, the alphanumeric value must be written correctly. A date consists of three components: year, month, and day. Hyphens separate the components.

In the second `INSERT` statement, the word `NULL` is specified as the twelfth value. This enables us to enter a null value explicitly. In this case, it means that the league number of player number 7 is unknown.

4.8 QUERYING TABLES

`SELECT` statements are used to retrieve data from tables. A number of examples illustrate the diverse features of this statement.

Example 4.7: Get the number, name, and date of birth of each player resident in Stratford; sort the result in alphabetical order of name (note that Stratford starts with a capital letter).

```
SELECT  PLAYERNO, NAME, BIRTH_DATE
FROM    PLAYERS
WHERE   TOWN = 'Stratford'
ORDER BY NAME
```

The result is:

PLAYERNO	NAME	BIRTH_DATE
39	Bishop	1956-10-29
57	Brown	1971-08-17
2	Everett	1948-09-01
83	Hope	1956-11-11
6	Parmenter	1964-06-25
100	Parmenter	1963-02-28
7	Wise	1963-05-11

Explanation: This SELECT statement should be read as follows: Get the number, name, and date of birth (SELECT PLAYERNO, NAME, BIRTH_DATE) of each player (FROM PLAYERS) resident in Stratford (WHERE TOWN = 'Stratford'); sort the result in alphabetical order of name (ORDER BY NAME). After FROM, we specify which table we want to query. The condition that the requested data must satisfy comes after WHERE. SELECT enables us to choose which columns we want to see. Figure 4.3 illustrates this in a graphical way. And after ORDER BY, we specify the column names on which the final result should be sorted.



FIGURE 4.3 An illustration of a SELECT statement

This book presents the result of a SELECT statement somewhat differently from the way MySQL does. The “default” layout used throughout this book is as follows.

First, the width of a column is determined by the width of the data type of the column. Second, the name of a column heading is equal to the name of the column in the SELECT statement. Third, the values in columns with an alphanumeric data type are left-justified and those in numeric columns are right-justified. Fourth, two spaces separate two columns. Fifth, a null value is displayed as a question mark. Finally, if a result is very long, some rows are left out and colons are presented.

Example 4.8: Get the number of each player who joined the club after 1980 and is resident in Stratford; order the result by player number.

```
SELECT  PLAYERNO
FROM    PLAYERS
WHERE   JOINED > 1980
AND     TOWN = 'Stratford'
ORDER BY PLAYERNO
```

The result is:

```
PLAYERNO
-----
          7
         57
         83
```

Explanation: Get the number (SELECT PLAYERNO) of each player (FROM PLAYERS) who joined the club after 1980 (WHERE JOINED > 1980) and is resident in Stratford (AND TOWN = 'Stratford'); sort the result by player number (ORDER BY PLAYERNO).

Example 4.9: Get all the information about each penalty.

```
SELECT  *
FROM    PENALTIES
```

The result is:

PAYMENTNO	PLAYERNO	PAYMENT_DATE	AMOUNT
1	6	1980-12-08	100.00
2	44	1981-05-05	75.00
3	27	1983-09-10	100.00
4	104	1984-12-08	50.00
5	44	1980-12-08	25.00
6	8	1980-12-08	25.00
7	44	1982-12-30	30.00
8	27	1984-11-12	75.00

Explanation: Get all column values (SELECT *) for each penalty (FROM PENALTIES). This statement returns the whole PENALTIES table. The * character is a shorthand notation for “all columns.” In this result, you can also see how dates are presented in this book.

Example 4.10: How much is 33 times 121?

```
SELECT    33 * 121
```

The result is:

```
33 * 121
-----
 3993
```

Explanation: This example shows that a SELECT statement does not always have to retrieve data from tables; it can also be used to perform straightforward calculations. If no tables are specified, the statement returns one row as result. This row contains the answers to the calculations.

4.9 UPDATING AND DELETING ROWS

Section 4.7 described how to add new rows to a table. This section covers updating and deleting existing rows.

A warning in advance: If you execute the statements described in this section, you will change the contents of the database. The subsequent sections assume that the original contents of the database are intact. You can restore the values by rerunning statements found at www.r20.nl.

The UPDATE *statement* is used to change values in rows, and the DELETE *statement* is used to remove complete rows from a table. Let us look at examples of both statements.

Example 4.11: Change the amount of each penalty incurred by player 44 to \$200.

```
UPDATE    PENALTIES
SET        AMOUNT = 200
WHERE      PLAYERNO = 44
```

Explanation: For each penalty (UPDATE PENALTIES) incurred by player 44 (WHERE PLAYERNO = 44), change the amount to \$200 (SET AMOUNT = 200). So the use of the WHERE clause in the UPDATE statement is equivalent to that of the SELECT statement—it indicates which rows must be changed. After the word SET, the columns that will have a new value are specified. The change is executed regardless of the existing value.

Issuing a SELECT statement can show the effect of the change. Before the update, the next SELECT statement

```
SELECT  PLAYERNO, AMOUNT
FROM    PENALTIES
WHERE   PLAYERNO = 44
```

gave the following result:

PLAYERNO	AMOUNT
44	75.00
44	25.00
44	30.00

After the change with the UPDATE statement, the result of the previous SELECT statement is different:

PLAYERNO	AMOUNT
44	200.00
44	200.00
44	200.00

Example 4.12: Remove each penalty with an amount greater than \$100 (we assume the changed contents of the PENALTIES table).

```
DELETE
FROM    PENALTIES
WHERE   AMOUNT > 100
```

Explanation: Remove the penalties (DELETE FROM PENALTIES) with an amount greater than \$100 (WHERE AMOUNT > 100). Again, the use of the WHERE clause is equivalent to that in the SELECT and UPDATE statements.

After this statement, the `PENALTIES` table looks as follows (shown by issuing a `SELECT` statement):

PAYMENTNO	PLAYERNO	PAYMENT_DATE	AMOUNT
1	6	1980-12-08	100.00
3	27	1983-09-10	100.00
4	104	1984-12-08	50.00
6	8	1980-12-08	25.00
8	27	1984-11-12	75.00

4.10 OPTIMIZING QUERY PROCESSING WITH INDEXES

We now look at how `SELECT` statements are processed—how MySQL arrives at the correct answer. The following `SELECT` statement illustrates this (assume the original contents of the `PENALTIES` table).

```
SELECT *
FROM   PENALTIES
WHERE  AMOUNT = 25
```

To process this statement, MySQL scans the entire `PENALTIES` table row by row. If the value of `AMOUNT` equals 25, that row is included in the result. As in this example, if the table contains only a few rows, MySQL can work quickly. However, if a table has thousands of rows and each must be checked, this could take a great deal of time. In such a case, defining an *index* can speed up the processing. For now, think of an index created with MySQL as similar to the index of a book. Chapter 25, “Using Indexes,” discusses this topic in more detail.

An index is defined on a column or combination of columns. See the following example.

Example 4.13: Create an index on the `AMOUNT` column of the `PENALTIES` table.

```
CREATE INDEX PENALTIES_AMOUNT ON
PENALTIES (AMOUNT)
```

Explanation: This statement defines an index called `PENALTIES_AMOUNT` for the `AMOUNT` column in the `PENALTIES` table.

This index ensures that in the earlier example, MySQL needs to look at only rows in the database that satisfy the `WHERE` condition. Therefore, it is quicker to produce an

answer. The index `PENALTIES_AMOUNT` provides direct access to these rows. Keep in mind the following points:

- Indexes are defined to optimize the processing of `SELECT` statements.
- An index is never explicitly referenced in a `SELECT` statement; the syntax of SQL does not allow this.
- During the processing of a statement, the database server itself determines whether an existing index will be used.
- An index may be created or deleted at any time.
- When updating, inserting, or deleting rows, MySQL also maintains the indexes on the impacted tables. This means that, on one hand, the processing time for `SELECT` statements is reduced; on the other hand, the processing time for update statements (such as `INSERT`, `UPDATE`, and `DELETE`) can increase.
- An index is also a database object.

A special type of index is the *unique* index. SQL also uses unique indexes to optimize the processing of statements. Unique indexes have another function as well: They guarantee that a particular column or combination of columns contains no duplicate values. A unique index is created by placing the word `UNIQUE` between the words `CREATE` and `INDEX`.

4.11 VIEWS

In a table, rows with data are actually stored. This means that a table occupies a particular amount of storage space—the more rows, the more storage space is required. *Views* are tables visible to users, but they do not occupy any storage space. A view, therefore, can also be referred to as a *virtual* or *derived* table. A view behaves as though it contains actual rows of data, but it contains none.

Example 4.14: Create a view in which the difference between the number of sets won and the number of sets lost are recorded for each match.

```
CREATE VIEW NUMBER_SETS (MATCHNO, DIFFERENCE) AS
SELECT MATCHNO, ABS(WON - LOST)
FROM MATCHES
```

Explanation: The previous statement defines a view with the name `NUMBER_SETS`. A `SELECT` statement is used to define the contents of the view. This view has only two columns: `MATCHNO` and `DIFFERENCE`. The value of the second column is determined by subtracting the number of sets lost from the number of sets won. The `ABS` function makes the value positive (Appendix B, “Scalar Functions,” discusses the precise meaning of `ABS`).

By using the `SELECT` statement shown here, you can see the (virtual) contents of the view:

```
SELECT      *
FROM        NUMBER_SETS
```

The result is:

MATCHNO	DIFFERENCE
1	2
2	1
3	3
4	1
5	3
6	2
7	3
8	3
9	1
10	1
11	1
12	2
13	3

The contents of the `NUMBER_SETS` view are *not* stored in the database but are derived at the moment a `SELECT` statement (or another statement) is executed. The use of views, therefore, costs nothing extra in storage space because the contents of a view can include only data that is already stored in other tables. Among other things, views can be used to do the following:

- Simplify the use of routine or repetitive statements
- Restructure the way in which tables are seen
- Develop `SELECT` statements in several steps
- Improve the security of data

Chapter 26, “Views,” looks at views more closely.

4.12 USERS AND DATA SECURITY

Data in a database should be protected against incorrect use and misuse. In other words, not everyone should have access to all the data in the database. As already shown in the beginning of this chapter, MySQL recognizes the concept of SQL user and privilege. Users need to make themselves known by logging on.

That same section showed an example of granting privileges to users. Here you will find more examples of the GRANT statement; assume that all the SQL users mentioned exist.

Example 4.15: Imagine that two SQL users, DIANE and PAUL, have been created. MySQL will reject most of their SQL statements as long as they have not been granted privileges. The following three statements give them the required privileges. Assume that a third SQL user, such as BOOKSQL, grants these privileges.

```
GRANT  SELECT
ON     PLAYERS
TO     DIANE
```

```
GRANT  SELECT, UPDATE
ON     PLAYERS
TO     PAUL
```

```
GRANT  SELECT, UPDATE
ON     TEAMS
TO     PAUL
```

When PAUL has logged on, he can query the TEAMS table, for example:

```
SELECT  *
FROM    TEAMS
```

4.13 DELETING DATABASE OBJECTS

For each type of database object for which a CREATE statement exists, a corresponding DROP statement with which the object can be deleted also exists. Consider a few examples.

Example 4.16: Delete the MATCHES table.

```
DROP TABLE MATCHES
```

Example 4.17: Delete the view `NUMBER_SETS`.

```
DROP VIEW NUMBER_SETS
```

Example 4.18: Delete the `PENALTIES_AMOUNT` index.

```
DROP INDEX PENALTIES_AMOUNT
```

Example 4.19: Delete the `TENNIS` database.

```
DROP DATABASE TENNIS
```

All dependent objects are also removed. For example, if the `PLAYERS` table is deleted, all indexes (which are defined on that table) and all privileges (which are dependent on that table) are automatically removed.

4.14 SYSTEM VARIABLES

MySQL has certain settings. When the MySQL database server is started, these settings are read to determine the next steps. For example, some settings define how data must be stored, others affect the processing speed, and still others relate to the system time and date. These settings are called *system variables*. Examples of system variables are `DATADIR` (the directory in which MySQL creates the databases), `LOG_WARNINGS`, `MAX_USER_CONNECTIONS`, and `TIME_ZONE`.

Sometimes it is important to know the value of a certain system variable. With a simple `SELECT` statement, you can retrieve its value.

Example 4.20: What is the most recent version of the MySQL database server that we use now?

```
SELECT @@VERSION
```

The result is:

```
@@VERSION
-----
5.0.7-beta-nt
```

Explanation: In MySQL, the value of the system variable `VERSION` is set to the version number. Specifying two `@` symbols before the name of the system variable returns its value.

Many system variables, such as `VERSION` and the system date, cannot be changed. However, some, including `SQL_MODE`, can be. To change system variables, use the *SET statement*.

Example 4.21: Change the value of the `SQL_MODE` parameter to `PIPES_AS_CONCAT`.

```
SET @@SQL_MODE = 'PIPES_AS_CONCAT'
```

Explanation: This change applies only to the current SQL user. In other words, different users can see different values for certain system variables.

Section 5.7 discusses system variables in detail. Some system variables are also described together with the SQL statement or clause that they have a relationship with.

Because the value of the `SQL_MODE` system variable affects the way of processing and the features of several SQL statements, we will discuss it in more detail. The value of `SQL_MODE` consists of a set of zero, one, or more settings that are separated by commas. For example, a possible value of `SQL_MODE` with four settings is shown here:

```
REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE
```

With a normal `SET` statement, we overwrite all the settings at once. If we want to add a setting, we can use the following statement.

Example 4.22: Add the setting `NO_ZERO_IN_DATE` to the `SQL_MODE` system variable.

```
SET @@SQL_MODE = CONCAT(@@SQL_MODE,  
    CASE @@SQL_MODE WHEN '' THEN '' ELSE ',' END,  
    'NO_ZERO_IN_DATE')
```

The meaning of these settings is explained later in this book.

4.15 GROUPING OF SQL STATEMENTS

SQL has many statements, but this chapter briefly describes only a few. In literature, it is customary to divide that large set of SQL statements into the following groups: Data Definition Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL), and procedural statements.

The *Data Definition Language (DDL)* consists of all the SQL statements that affect the structure of database objects, such as tables, indexes, and views. The `CREATE TABLE` statement is a clear example of a DDL statement, but so are `CREATE INDEX` and `DROP TABLE`.

SQL statements used to query and change the contents of tables belong to the *Data Manipulation Language (DML)* group. Examples of DML statements are `SELECT`, `UPDATE`, `DELETE`, and `INSERT`.

Data Control Language (DCL) statements relate to the security of data and the revoking of privileges. This chapter has discussed the `GRANT` statement; the `REVOKE` statement is also a DCL statement.

Examples of *procedural statements* are `IF-THEN-ELSE` and `WHILE-DO`. These classical statements have been added to SQL to create relatively new database objects, such as triggers and stored procedures.

The names of these groups sometimes assume that SQL consists of several individual languages, but this is incorrect. All SQL statements are part of one language and are grouped for the sake of clarity.

Appendix A, “Syntax of SQL,” which defines all SQL statements, indicates the group to which an SQL statement belongs.

4.16 THE CATALOG TABLES

MySQL maintains lists of user names and passwords and the sequence in which columns in the `CREATE TABLE` statements have been created (see Section 4.6). However, where is all this data stored? Where does SQL keep track of all these names, passwords, tables, columns, sequence numbers, and so on? MySQL has a number of tables for its own use in which this data is stored. These tables are called *catalog tables* or *system tables*; together they form the *catalog*.

Each catalog table is an “ordinary” table that can be queried using `SELECT` statements. Querying the catalog tables can have many uses, including these:

- As a *help function* for new users to determine which tables in the database are available and which columns the tables contain
- As a *control function* so that users can see, for example, which indexes, views, and privileges would be deleted if a particular table was dropped
- As a *processing function* for MySQL itself when it executes statements (as a help function for MySQL)

Catalog tables *cannot* be accessed using statements such as `UPDATE` and `DELETE`—the SQL database server maintains these tables itself.

MySQL has two databases in which catalog tables are included. The database called `MYSQL` contains data on privileges, users, and tables. The structure of these tables is somewhat cryptic and is unique for MySQL. In addition, the database called `INFORMATION_SCHEMA` contains catalog data that partly overlaps the data in the `MYSQL` database. The structure of `INFORMATION_SCHEMA` conforms to the SQL standard and looks similar to the structure of other SQL products.

The structure of the catalog tables is not simple. We have defined several simple views on the catalog tables of MySQL. These views are partly defined on the tables of the `MYSQL` database and partly on those of the `INFORMATION_SCHEMA` database. So actually, they are not catalog tables, but catalog views. In a simple and transparent way, they give access to the actual, underlying catalog tables.

Part III, “Creating Database Objects,” which discusses different database objects, such as tables and views, describes the different catalog tables that belong to the `INFORMATION_SCHEMA` database. In the first two parts of this book, the catalog views suffice.

If you are familiar with MySQL and have worked your way through most chapters in this book, we recommend that you look at the structure of the actual catalog tables. They are, after all, just tables that you can access with `SELECT` statements. Understanding the catalog will definitely increase your knowledge of MySQL.

In the rest of this book, we use these simple catalog views, so we recommend that you create these views. You can reference the web site of this book for assistance. You can adjust these catalog views later—you can add new columns and new catalog views. By studying how these views have been built makes it easier to understand the real catalog tables later.

Example 4.23: Create the following catalog views. These views must be created in the sequence specified because of interdependences.

```
CREATE OR REPLACE VIEW USERS
(USER_NAME) AS
SELECT DISTINCT UPPER(CONCAT(' ',USER,' '@',HOST,' '))
FROM MYSQL.USER

CREATE OR REPLACE VIEW TABLES
(TABLE_CREATOR, TABLE_NAME,
CREATE_TIMESTAMP, COMMENT) AS
SELECT UPPER(TABLE_SCHEMA), UPPER(TABLE_NAME),
CREATE_TIME, TABLE_COMMENT
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE IN ('BASE TABLE','TEMPORARY')
```



```

CREATE OR REPLACE VIEW COLUMNS
(TABLE_CREATOR, TABLE_NAME, COLUMN_NAME,
 COLUMN_NO, DATA_TYPE, CHAR_LENGTH,
 'PRECISION', SCALE, NULLABLE, COMMENT) AS
SELECT UPPER(TABLE_SCHEMA), UPPER(TABLE_NAME),
        UPPER(COLUMN_NAME), ORDINAL_POSITION,
        UPPER(DATA_TYPE), CHARACTER_MAXIMUM_LENGTH,
        NUMERIC_PRECISION, NUMERIC_SCALE, IS_NULLABLE,
        COLUMN_COMMENT
FROM INFORMATION_SCHEMA.COLUMNS

CREATE OR REPLACE VIEW VIEWS
(VIEW_CREATOR, VIEW_NAME, CREATE_TIMESTAMP,
 WITHCHECKOPT, IS_UPDATABLE, VIEWFORMULA, COMMENT) AS
SELECT UPPER(V.TABLE_SCHEMA), UPPER(V.TABLE_NAME),
        T.CREATE_TIME,
        CASE
            WHEN V.CHECK_OPTION = 'None' THEN 'NO'
            WHEN V.CHECK_OPTION = 'Cascaded' THEN 'CASCADED'
            WHEN V.CHECK_OPTION = 'Local' THEN 'LOCAL'
            ELSE 'Yes'
        END, V.IS_UPDATABLE, V.VIEW_DEFINITION, T.TABLE_COMMENT
FROM INFORMATION_SCHEMA.VIEWS AS V,
        INFORMATION_SCHEMA.TABLES AS T
WHERE V.TABLE_NAME = T.TABLE_NAME
AND V.TABLE_SCHEMA = T.TABLE_SCHEMA

CREATE OR REPLACE VIEW INDEXES
(INDEX_CREATOR, INDEX_NAME, CREATE_TIMESTAMP,
 TABLE_CREATOR, TABLE_NAME, UNIQUE_ID, INDEX_TYPE) AS
SELECT DISTINCT UPPER(I.INDEX_SCHEMA), UPPER(I.INDEX_NAME),
        T.CREATE_TIME, UPPER(I.TABLE_SCHEMA),
        UPPER(I.TABLE_NAME),
        CASE
            WHEN I.NON_UNIQUE = 0 THEN 'YES'
            ELSE 'NO'
        END,
        I.INDEX_TYPE
FROM INFORMATION_SCHEMA.STATISTICS AS I,
        INFORMATION_SCHEMA.TABLES AS T
WHERE I.TABLE_NAME = T.TABLE_NAME
AND I.TABLE_SCHEMA = T.TABLE_SCHEMA

```

```
CREATE OR REPLACE VIEW COLUMNS_IN_INDEX
(INDEX_CREATOR, INDEX_NAME,
TABLE_CREATOR, TABLE_NAME, COLUMN_NAME,
COLUMN_SEQ, ORDERING) AS
SELECT UPPER(INDEX_SCHEMA), UPPER(INDEX_NAME),
UPPER(TABLE_SCHEMA), UPPER(TABLE_NAME),
UPPER(COLUMN_NAME), SEQ_IN_INDEX,
CASE
    WHEN COLLATION = 'A' THEN 'ASCENDING'
    WHEN COLLATION = 'D' THEN 'DESCENDING'
    ELSE 'OTHER'
END
FROM INFORMATION_SCHEMA.STATISTICS

CREATE OR REPLACE VIEW USER_AUTHS
(GRANTOR, GRANTEE, PRIVILEGE, WITHGRANTOPT) AS
SELECT 'UNKNOWN', UPPER(GRANTEE), PRIVILEGE_TYPE, IS_GRANTABLE
FROM INFORMATION_SCHEMA.USER_PRIVILEGES

CREATE OR REPLACE VIEW DATABASE_AUTHS
(GRANTOR, GRANTEE, DATABASE_NAME, PRIVILEGE,
WITHGRANTOPT) AS
SELECT 'UNKNOWN', UPPER(GRANTEE), UPPER(TABLE_SCHEMA),
PRIVILEGE_TYPE, IS_GRANTABLE
FROM INFORMATION_SCHEMA.SCHEMA_PRIVILEGES

CREATE OR REPLACE VIEW TABLE_AUTHS
(GRANTOR, GRANTEE, TABLE_CREATOR, TABLE_NAME,
PRIVILEGE, WITHGRANTOPT) AS
SELECT 'UNKNOWN', UPPER(GRANTEE), UPPER(TABLE_SCHEMA),
UPPER(TABLE_NAME), PRIVILEGE_TYPE, IS_GRANTABLE
FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES

CREATE OR REPLACE VIEW COLUMN_AUTHS
(GRANTOR, GRANTEE, TABLE_CREATOR, TABLE_NAME,
COLUMN_NAME, PRIVILEGE, WITHGRANTOPT) AS
SELECT 'UNKNOWN', UPPER(GRANTEE), UPPER(TABLE_SCHEMA),
UPPER(TABLE_NAME), UPPER(COLUMN_NAME),
PRIVILEGE_TYPE, IS_GRANTABLE
FROM INFORMATION_SCHEMA.COLUMN_PRIVILEGES
```

Table 4.1 lists the catalog tables (catalog views) that are created.

TABLE 4.1 Examples of Catalog Views

TABLE NAME	EXPLANATION
USERS	Contains for each SQL user the names of the users who were not created during the installation procedure
TABLES	Contains for each table information such as the date and time the table was created
COLUMNS	Contains for each column (belonging to a table or view) information such as the data type, the table to which the column belongs, whether the null value is allowed, and the sequence number of the column in the table
VIEWS	Contains for each view information such as the view definition (the SELECT statement)
INDEXES	Contains for each index information such as the table and the columns on which the index is defined, and the manner in which the index is ordered
COLUMNS_IN_INDEX	Contains for each index the columns on which the index is defined
DATABASE_AUTHS	Contains the database privileges that are granted to users
TABLE_AUTHS	Contains the table privileges that are granted to users
COLUMN_AUTHS	Contains the column privileges that are granted to users

Consider the following examples of queries on the catalog table.

Example 4.24: Get the name, data type, and sequence number of each column in the PLAYERS table (which was created in the TENNIS database); order the result by sequence number.

```
SELECT  COLUMN_NAME, DATA_TYPE, COLUMN_NO
FROM    COLUMNS
WHERE   TABLE_NAME = 'PLAYERS'
AND     TABLE_CREATOR = 'TENNIS'
ORDER BY COLUMN_NO
```

The result is:

COLUMN_NAME	DATA_TYPE	COLUMN_NO
-----	-----	-----
PLAYERNO	INT	1
NAME	CHAR	2
INITIALS	CHAR	3
BIRTH_DATE	DATE	4
SEX	CHAR	5
JOINED	SMALLINT	6
STREET	VARCHAR	7
HOUSENO	CHAR	8
POSTCODE	CHAR	9
TOWN	VARCHAR	10
PHONONO	CHAR	11
LEAGUENO	CHAR	12

Explanation: Get the name, data type, and sequence number (SELECT COLUMN_NAME, DATA_TYPE, COLUMN_NO) of each column (FROM COLUMNS) in the PLAYERS table (WHERE TABLE_NAME = 'PLAYERS') that is created in the TENNIS database (AND TABLE_CREATOR = 'TENNIS'); order the result by sequence number (ORDER BY COLUMN_NO).

Example 4.25: Get the names of the indexes defined on the PENALTIES table.

```
SELECT INDEX_NAME
FROM INDEXES
WHERE TABLE_NAME = 'PENALTIES'
AND TABLE_CREATOR = 'TENNIS'
```

Result (for example):

```
INDEX_NAME
-----
PRIMARY
PENALTIES_AMOUNT
```

Explanation: MySQL created the index that is mentioned first, with the name PRIMARY, because a primary key was specified on the PLAYERS table. Chapter 25 returns to this topic. The second index was created in Example 4.13.

Other chapters describe the effect that processing particular statements can have on the contents of the catalog tables. The catalog tables are an integral part of SQL.

You can find the original catalog tables that MySQL created in two different databases, called `MYSQL` and `INFORMATION_SCHEMA`. Both were created during MySQL installation. You can access the tables of the databases directly, without the intervention of the catalog views (see the following example).

Example 4.26: Get the names of the indexes that have been defined on the `PENALTIES` table.

```
USE INFORMATION_SCHEMA

SELECT  DISTINCT INDEX_NAME
FROM    STATISTICS
WHERE   TABLE_NAME = 'PENALTIES'
```

The result is:

```
INDEX_NAME
-----
PRIMARY
PENALTIES_AMOUNT
```

Explanation: With the `USE` statement, we make `INFORMATION_SCHEMA` the current database. Then all the tables of the catalog can be accessed.

Example 4.27: Show the names of the tables that are stored in the `INFORMATION_SCHEMA` database.

```
SELECT  TABLE_NAME
FROM    TABLES
WHERE   TABLE_SCHEMA = 'INFORMATION_SCHEMA'
ORDER BY TABLE_NAME
```

The result is:

```
TABLE_NAME
-----
CHARACTER_SETS
COLLATIONS
COLLATION_CHARACTER_SET_APPLICABILITY
COLUMNS
COLUMN_PRIVILEGES
ENGINES
EVENTS
FILES
```

```

KEY_COLUMN_USAGE
PARTITIONS
PLUGINS
PROCESSLIST
REFERENTIAL_CONSTRAINTS
ROUTINES
SCHEMATA
SCHEMA_PRIVILEGES
STATISTICS
TABLES
TABLE_CONSTRAINTS
TABLE_PRIVILEGES
TRIGGERS
USER_PRIVILEGES
VIEWS

```

The special SQL statement called `SHOW` is another way to get access to all this descriptive catalog data. See the next two examples.

Example 4.28: Get the descriptive data of the columns belonging to the `PLAYERS` table.

```
SHOW COLUMNS FROM PLAYERS
```

And the result is:

Field	Type	Null	Key	Default	Extra
-----	-----	----	---	-----	-----
PLAYERNO	int(11)		PRI	0	
NAME	varchar(15)				
INITIALS	char(3)				
BIRTH_DATE	date	YES			
SEX	char(1)				
JOINED	smallint(6)			0	
STREET	varchar(30)				
HOUSENO	varchar(4)	YES			
POSTCODE	varchar(6)	YES			
TOWN	varchar(30)				
PHONENO	varchar(13)	YES			
LEAGUENO	varchar(4)	YES			

Example 4.29: Get the descriptive data of the indexes defined on the `PENALTIES` table.

```
SHOW INDEX FROM PENALTIES
```

An example result is:

Table	Non-unique	Key_name	Column_name	Collation
PENALTIES	0	PRIMARY	PAYMENTNO	A
PENALTIES	1	PENALTIES_AMOUNT	AMOUNT	A

Explanation: MySQL created the first index itself because we defined a primary key on the `PENALTIES` table. Chapter 25 returns to this. The second index was created in Example 4.13. This `SHOW` statement returns more than these columns, but we omitted them for the sake of convenience.

Try the next statement yourself and look at the result:

```
SHOW DATABASES
SHOW TABLES
SHOW CREATE TABLE PLAYERS
SHOW INDEX FROM PLAYERS
SHOW GRANTS FOR BOOKSQL@localhost
SHOW PRIVILEGES
```

4.17 RETRIEVING ERRORS AND WARNINGS

Sometimes things go wrong. If we do something wrong, MySQL presents one or more error messages. For example, if we make a typo in an SQL statement, we get an error message. MySQL won't even try to process the statement. Perhaps a statement is syntactically correct, but we are trying to do something that is impossible, such as create a table with a name that already exists. In this situation, MySQL also returns an error message. However, not all the error messages are presented—it depends on the seriousness of the error message. After an SQL statement has been processed, we can request all the error messages with the `SHOW WARNINGS` statement.

Example 4.30: What is the result of the calculation 10 divided by 0?

```
SELECT 10 / 0
```

The result of this statement is empty because we cannot divide by zero. But no error message is given. We can ask for them as follows:

```
SHOW WARNINGS
```

The result is:

Level	Code	Message
-----	-----	-----
Error	1305	FUNCTION tennis.chr does not exist
Error	1105	Unknown error

Before processing the next SQL statement, all these messages are deleted, and a new list is created.

With the statement `SHOW COUNT(*) WARNINGS`, we can ask for the number of error messages. We get the same result if we ask for the value of the system variable called `WARNING_COUNT`.

The statement `SHOW ERRORS` resembles `SHOW WARNINGS`. The former returns all the errors, warnings, and notes; the latter returns the errors only. And, of course, a system `COLUMN_AUTHS` variable called `ERROR_COUNT` exists.

4.18 DEFINITIONS OF SQL STATEMENTS

This book uses a particular formal notation to indicate the syntax of certain SQL statements. In other words, by using this notation, we give a definition of an SQL statement. These definitions are clearly indicated by enclosing the text in boxes. For example, the following is part of the definition of the `CREATE INDEX` statement:



DEFINITION

```
<create index statement> ::=
    CREATE [ UNIQUE ] INDEX <index name>
    ON <table name> <column list>
```

```
<column list> ::=
    ( <column name> [ , <column name> ]... )
```

If you are not familiar with this notation, we advise that you study it before you continue with the next chapters (see Appendix A).

Because the functionality of certain SQL statements is very extensive, we do not always show the complete definition in one place, but instead extend it step by step. We omit the definitions of the syntactically simple statements. Appendix A includes the complete definitions of all SQL statements.

This page intentionally left blank

Part II

Querying and Updating Data

One statement in particular forms the core of SQL and clearly represents the nonprocedural nature of SQL: the `SELECT` statement. It is the show-piece of SQL and, consequently, MySQL. This statement is used to query data in the tables; the result is always a table. Such a result table can be used as the basis of a report, for example.

This book deals with the `SELECT` statement in Chapters 5–15. Each chapter is devoted to one or two clauses of this statement. Several chapters have been added to explain certain concepts in more detail.

Chapter 16, “The `HANDLER` Statement,” is devoted to the `HANDLER` statement, which offers an alternative method to query data. In a more simple way, rows can be retrieved individually. The features of this statement are much more limited than those of the `SELECT` statement. However, for certain applications, `HANDLER` can be more suitable than `SELECT`.

Chapter 17 describes how to insert, update, and delete data. The features of these statements are strongly based upon those of the `SELECT` statement, which makes the latter so important to master.

With MySQL, data can be loaded from files into the database vice versa: Data can be unloaded to external files. Chapter 18, “Loading and Unloading Data,” describes the statements and features to do so.

The use of XML documents has become increasingly popular. Because of this, the need to store these special documents in tables has increased. Chapter 19, “Working with XML Documents,” describes the functions with which XML documents can be queried and updated.

SELECT Statement: Common Elements

5.1 INTRODUCTION

This first chapter dealing with the SELECT statement describes a number of common elements that are important to many SQL statements and certainly crucial to the SELECT statement. Those who are familiar with programming languages and other database languages will find most of these concepts familiar.

Among others, this chapter covers the following common elements:

- Literal
- Expression
- Column specification
- User variable
- System variable
- Case expression
- Scalar function
- Null value
- Cast expression
- Compound expression
- Row expression
- Table expression
- Aggregation function

5.2 LITERALS AND THEIR DATA TYPES

The previous chapter used literals in many examples of SQL statements. A *literal* is a fixed or unchanging value. Literals are used, for example, in conditions for selecting rows in `SELECT` statements and for specifying the values for a new row in `INSERT` statements; see Figure 5.1.

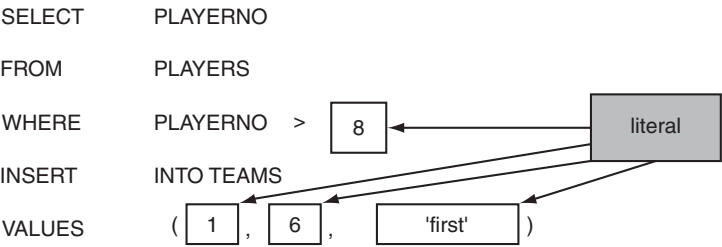



FIGURE 5.1 Literals in SQL statements

Each literal has a particular *data type*, just like a column in a table. The names of the different types of literals are derived from the names of their respective data types as we use them in the `CREATE TABLE` statement.

The literals are divided into several main groups: the numeric, the alphanumeric, the temporal, the Boolean, and the hexadecimal literals. They all have their own properties, idiosyncrasies, and limitations. Here you will find the definitions of all literals followed by the descriptions.

Each literal always has a *data type*; however, no literal exists for each data type. Chapter 20, “Creating Tables, discusses all data types (including the one for which no literal exists). That chapter also describes the `CREATE TABLE` statement in detail.

 **DEFINITION**

```
<literal> ::=
  <numeric literal> |
  <alphanumeric literal> |
  <temporal literal> |
  <boolean literal> |
  <hexadecimal literal>

<numeric literal> ::=
  <integer literal> |
  <decimal literal> |
  <float literal> |
  <bit literal>
```

continues

```

<integer literal> ::= [ + | - ] <whole number>

<decimal literal> ::=
    [ + | - ] <whole number> [ .<whole number> ] |
    [ + | - ] <whole number> .
    [ + | - ] .<whole number>

<float literal> ::=
    <mantissa> { E | e } <exponent>

<bit literal > ::=
    { b | B } ' { 0 | 1 }... '

<alphanumeric literal> ::= <character list>

<temporal literal> ::=
    <date literal> |
    <time literal> |
    <datetime literal> |
    <timestamp literal> |
    <year literal>

<date literal> ::=
    { ' <years> - <months> - <days> ' } |
    { <years> <months> <days> }

<time literal> ::=
    { ' <hours> : <minutes> [ : <seconds>
      [ . <microseconds> ] ] ' } |
    { ' [ <hours> : <minutes> : ] <seconds> ' } |
    { <hours> <minutes> <seconds> } |
    { [ [ <hours> ] <minutes> ] <seconds> }

<datetime literal> ;
<timestamp literal> ::=
    { ' <years> - <months> - <days> <space>
      [ <hours> [ : <minutes> [ : <seconds>
        [ . <micro seconds> ] ] ] ] ' } |
    { <years> <months> <days> <hours> <minutes> <seconds> }

<year literal> ::= <year>

<hexadecimal literal> ::=
    { X | x } <hexadecimal character>... |
    0x <hexadecimal character>...

<hexadecimal character> ::=
    <digit> | A | B | C | D | E | F | a | b | c | d | e | f

```

continues

```

<years>          ;
<micro seconds> ;
<year>           ::= <whole number>

<months>        ;
<days>         ;
<hours>         ;
<minutes>       ;
<seconds>       ::= <digit> [ <digit> ]

<whole number> ::= <digit>...

<boolean literal> ::= TRUE | true | FALSE | false

<mantissa> ::= <decimal literal>

<exponent> ::= <integer literal>

<character list> ::= ' [ <character>... ] '

<character> ::= <digit> | <letter> | <special character> | ''

<special character> ::=
    { \ { 0 | ' | " | b | n | r | t | z | \ | % } } |
    <any other character>

<whole number> ::= <digit>...

```

5.2.1 The Integer Literal

MySQL has several types of numeric literals. The *integer literal* is used frequently. This is a whole number or integer without a decimal point, possibly preceded by a plus or minus sign. Examples are shown here:

```

38
+12
-3404
016

```

The following examples are *not* correct integer literals:

```

342.16
-14E5
jan

```

5.2.2 The Decimal Literal

The second numeric literal is the *decimal literal*. This is a number with or without a decimal point, possibly preceded by a plus or minus sign. Each integer literal is, by definition, a decimal literal. Examples follow:

```
49
18.47
-3400
17.
0.83459
-.47
```

The total number of digits is called the *precision*, and the number of digits after the decimal point is the *scale*. The decimal literal 123.45 has a precision of 5 and a scale of 2. The scale of an integer literal is always 0. The maximum range of a decimal literal is measured by the scale and the precision. The precision must be greater than 0, and the scale must be between 0 and the precision. For example, a decimal with a precision of 8 and a scale of 2 is allowed, but not with a precision of 6 and a scale of 8.

In the sample database of the tennis club, only one column has been defined with this data type, and that is AMOUNT in the PENALTIES table.

5.2.3 Float, Real, and Double Literals

A *float literal* is a decimal literal followed by an *exponent*. *Float* is short for *single precision floating point*. These are examples of float literals:

Float literal	Value
-34E2	-3400
0.16E4	1600
4E-3	0.004
4e-3	0.004

5.2.4 The Alphanumeric Literal

An *alphanumeric literal* is a string of zero or more alphanumeric characters enclosed in quotation marks. This could be double (") or single (') quotation marks. The quotation marks are not considered to be part of the literal; they define the beginning and end of the string. The following characters are permitted in an alphanumeric literal: