

Ryan Stephens  
Ron Plew  
Arie D. Jones

**Fourth Edition**

Sams **Teach Yourself**

**SQL**®

in **24**  
**Hours**

**SAMS**



Ryan Stephens  
Ron Plew  
Arie D. Jones

Sams **Teach Yourself**

**SQL<sup>®</sup>**

in **24**  
**Hours**

**FOURTH EDITION**

From library of Wow! eBook  
[www.wowebook.com](http://www.wowebook.com)

**SAMS**

800 East 96th Street, Indianapolis, Indiana, 46240 USA

## **Sams Teach Yourself SQL® in 24 Hours, Fourth Edition**

Copyright © 2008 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-672-33018-6

ISBN-10: 0-672-33018-0

*Library of Congress Cataloging-in-Publication Data*

Stephens, Ryan K.

*Sams teach yourself SQL in 24 hours / Ryan Stephens, Ron Plew, Arie D.*

Jones. – 4th ed.

*p. cm. – (Sams teach yourself in 24 hours)*

*On t.p. of earlier ed. Ronald R. Plew's name appeared first.*

*Includes indexes*

*ISBN 978-0-672-33018-6 (pbk.)*

*1. SQL (Computer program language) I. Plew, Ronald R. II. Jones,*

*Arie. III. Plew, Ronald R. Sams teach yourself SQL in 24 hours. IV.*

*Title.*

*QA76.73.S67P554 2008*

*005.75'6–dc22*

2008016630

Printed in the United States of America

First Printing May 2008

### **Trademarks**

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

### **Warning and Disclaimer**

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

### **Bulk Sales**

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

**U.S. Corporate and Government Sales**

**1-800-382-3419**

**corpsales@pearsontechgroup.com**

For sales outside of the U.S., please contact

**International Sales**

**international@pearson.com**

### **Associate**

#### **Publisher**

*Mark Taub*

### **Acquisitions Editor**

*Trina MacDonald*

### **Development Editor**

*Michael Thurston*

### **Managing Editor**

*Patrick Kanouse*

### **Project Editor**

*Mandie Frank*

### **Copy Editor**

*Heather Wilkins*

*Editorial Services*

### **Indexer**

*Heather McNeil*

### **Proofreader**

*Matt Purcell*

### **Technical Editor**

*Steve Cvar*

### **Publishing Coordinator**

*Olivia Basegio*

### **Designer**

*Gary Adair*

### **Composition**

*Bronkella Publishing*

# Contents at a Glance

Introduction .....	1
<b>Part I: A SQL Concepts Overview</b>	
<b>HOUR 1</b> Welcome to the World of SQL .....	7
<b>Part II: Building Your Database</b>	
<b>HOUR 2</b> Defining Data Structures .....	27
<b>3</b> Managing Database Objects .....	41
<b>4</b> The Normalization Process .....	61
<b>5</b> Manipulating Data .....	73
<b>6</b> Managing Database Transactions .....	87
<b>Part III: Getting Effective Results from Queries</b>	
<b>HOUR 7</b> Introduction to the Database Query .....	101
<b>8</b> Using Operators to Categorize Data .....	117
<b>9</b> Summarizing Data Results from a Query .....	141
<b>10</b> Sorting and Grouping Data .....	151
<b>11</b> Restructuring the Appearance of Data .....	165
<b>12</b> Understanding Dates and Times .....	185
<b>Part IV: Building Sophisticated Database Queries</b>	
<b>HOUR 13</b> Joining Tables in Queries .....	203
<b>14</b> Using Subqueries to Define Unknown Data .....	221
<b>15</b> Combining Multiple Queries into One .....	235
<b>Part V: SQL Performance Tuning</b>	
<b>HOUR 16</b> Using Indexes to Improve Performance .....	253
<b>17</b> Improving Database Performance .....	265

**Part VI: Using SQL to Manage Users and Security**

**HOURL 18** Managing Database Users ..... 283  
**19** Managing Database Security ..... 297

**Part VII: Summarized Data Structures**

**HOURL 20** Creating and Using Views and Synonyms ..... 313  
**21** Working with the System Catalog ..... 329

**Part VIII: Applying SQL Fundamentals in Today's World**

**HOURL 22** Advanced SQL Topics ..... 343  
**23** Extending SQL to the Enterprise, the Internet, and the Intranet ..... 359  
**24** Extensions to Standard SQL ..... 369

**Part IX: Appendixes**

**A** Common SQL Commands ..... 381  
**B** Using MySQL for Exercises ..... 387  
**C** Answers to Quizzes and Exercises ..... 391  
**D** CREATE TABLE Statements for Book Examples ..... 435  
**E** INSERT Statements for Data in Book Examples ..... 437  
**F** Bonus Exercises ..... 441  
Glossary ..... 447  
Index ..... 451

# Table of Contents

<b>Introduction</b>	<b>1</b>
What This Book Intends to Accomplish .....	1
What We Added to This Edition .....	1
What You Need .....	2
Conventions Used in This Book .....	2
ANSI SQL and Vendor Implementations .....	3
Understanding the Examples and Exercises .....	3
<b>Part I: A SQL Concepts Overview</b>	
<b>HOOR 1: Welcome to the World of SQL</b>	<b>7</b>
SQL Definition and History .....	7
SQL Sessions .....	14
Types of SQL Commands .....	15
The Database Used in This Book .....	17
Summary .....	22
Q&A .....	23
Workshop .....	24
<b>Part II: Building Your Database</b>	
<b>HOOR 2: Defining Data Structures</b>	<b>27</b>
What Is Data? .....	27
Basic Data Types .....	28
Summary .....	36
Q&A .....	37
Workshop .....	37

**Sams Teach Yourself SQL in 24 Hours**

<b>HOOR 3: Managing Database Objects</b>	<b>41</b>
What Are Database Objects? .....	41
What Is a Schema? .....	42
A Table: The Primary Storage for Data .....	44
Integrity Constraints .....	52
Summary .....	56
Q&A .....	57
Workshop .....	58
<b>HOOR 4: The Normalization Process</b>	<b>61</b>
Normalizing a Database .....	61
Denormalizing a Database .....	69
Summary .....	69
Q&A .....	70
Workshop .....	70
<b>HOOR 5: Manipulating Data</b>	<b>73</b>
Overview of Data Manipulation .....	73
Populating Tables with New Data .....	74
Updating Existing Data .....	79
Deleting Data from Tables .....	81
Summary .....	82
Q&A .....	82
Workshop .....	83
<b>HOOR 6: Managing Database Transactions</b>	<b>87</b>
What Is a Transaction? .....	87
Controlling Transactions .....	88
Transactional Control and Database Performance .....	95
Summary .....	95
Q&A .....	96
Workshop .....	96

### **Part III: Getting Effective Results from Queries**

<b>HOUR 7: Introduction to the Database Query</b>	<b>101</b>
What Is a Query? .....	101
Introduction to the SELECT Statement .....	101
Examples of Simple Queries .....	109
Summary .....	113
Q&A .....	113
Workshop .....	114
<b>HOUR 8: Using Operators to Categorize Data</b>	<b>117</b>
What Is an Operator in SQL? .....	117
Comparison Operators .....	118
Logical Operators .....	121
Conjunctive Operators .....	127
Negative Operators .....	130
Arithmetic Operators .....	134
Summary .....	138
Q&A .....	138
Workshop .....	138
<b>HOUR 9: Summarizing Data Results from a Query</b>	<b>141</b>
What Are Aggregate Functions? .....	141
Summary .....	149
Q&A .....	149
Workshop .....	149
<b>HOUR 10: Sorting and Grouping Data</b>	<b>151</b>
Why Group Data? .....	151
The GROUP BY Clause .....	152
GROUP BY Versus ORDER BY .....	156
The HAVING Clause .....	159
Summary .....	160

**Sams Teach Yourself SQL in 24 Hours**

Q&A .....	160
Workshop .....	161
<b>HOURL 11: Restructuring the Appearance of Data</b> .....	<b>165</b>
ANSI Character Functions .....	165
Various Common Character Functions .....	166
Miscellaneous Character Functions .....	175
Mathematical Functions .....	178
Conversion Functions .....	179
Combining Character Functions .....	181
Summary .....	182
Q&A .....	182
Workshop .....	183
<b>HOURL 12: Understanding Dates and Times</b> .....	<b>185</b>
How Is a Date Stored? .....	186
Date Functions .....	187
Date Conversions .....	192
Summary .....	197
Q&A .....	197
Workshop .....	198
 <b>Part IV: Building Sophisticated Database Queries</b>	
<b>HOURL 13: Joining Tables in Queries</b> .....	<b>203</b>
Selecting Data from Multiple Tables .....	203
Types of Joins .....	204
Join Considerations .....	214
Summary .....	218
Q&A .....	218
Workshop .....	219

<b>HOUR 14: Using Subqueries to Define Unknown Data</b>	<b>221</b>
What Is a Subquery? .....	221
Embedded Subqueries .....	227
Correlated Subqueries .....	229
Summary .....	230
Q&A .....	231
Workshop .....	231
<b>HOUR 15: Combining Multiple Queries into One</b>	<b>235</b>
Single Queries Versus Compound Queries .....	235
Compound Query Operators .....	236
Using ORDER BY with a Compound Query .....	242
Using GROUP BY with a Compound Query .....	244
Retrieving Accurate Data .....	246
Summary .....	246
Q&A .....	246
Workshop .....	247
<b>Part V: SQL Performance Tuning</b>	
<b>HOUR 16: Using Indexes to Improve Performance</b>	<b>253</b>
What Is an Index? .....	253
How Do Indexes Work? .....	254
The CREATE INDEX Command .....	255
Types of Indexes .....	255
When Should Indexes Be Considered? .....	258
When Should Indexes Be Avoided? .....	259
Dropping an Index .....	260
Summary .....	261
Q&A .....	261
Workshop .....	262

**Sams Teach Yourself SQL in 24 Hours**

<b>HOOR 17: Improving Database Performance</b>	<b>265</b>
What Is SQL Statement Tuning? .....	265
Database Tuning Versus SQL Statement Tuning .....	266
Formatting Your SQL Statement .....	266
Full Table Scans .....	272
Other Performance Considerations .....	273
Performance Tools .....	276
Summary .....	276
Q&A .....	277
Workshop .....	278

**Part VI: Using SQL to Manage Users and Security**

<b>HOOR 18: Managing Database Users</b>	<b>283</b>
Users Are the Reason .....	284
The Management Process .....	286
Tools Utilized by Database Users .....	293
Summary .....	294
Q&A .....	294
Workshop .....	295
<b>HOOR 19: Managing Database Security</b>	<b>297</b>
What Is Database Security? .....	297
What Are Privileges? .....	298
Controlling User Access .....	302
Controlling Privileges Through Roles .....	305
Summary .....	307
Q&A .....	308
Workshop .....	309







**Sams Teach Yourself SQL in 24 Hours**

<b>APPENDIX D: CREATE TABLE Statements for Book Examples</b>	<b>435</b>
EMPLOYEE_TBL .....	435
EMPLOYEE_PAY_TBL .....	435
CUSTOMER_TBL .....	436
ORDERS_TBL .....	436
PRODUCTS_TBL .....	436
<b>APPENDIX E: INSERT Statements for Book Examples</b>	<b>437</b>
EMPLOYEE_TBL .....	437
EMPLOYEE_PAY_TBL .....	438
CUSTOMER_TBL .....	438
ORDERS_TBL .....	439
PRODUCTS_TBL .....	440
<b>APPENDIX F: Bonus Exercises</b>	<b>441</b>
<b>Glossary</b>	<b>447</b>
<b>Index</b>	<b>451</b>











## What You Need

You might be wondering, what do I need to make this book work for me? Theoretically, you should be able to pick up this book, study the material for the current hour, study the examples, and either write out the exercises or run them on a relational database server. However, it would be to your benefit to have access to a relational database system to which to apply the material in each lesson. The relational database to which you have access is not a major factor because SQL is the standard language for all relational databases. Some database systems that you can use include Oracle, Sybase, Informix, Microsoft SQL Server, Microsoft Access, MySQL, and dBASE.

## Conventions Used in This Book

For the most part, we have tried to keep conventions in this book as simple as possible.

Many new terms are printed in italics.

In the listings, all code that you type in (input) appears in boldface monospace. Output appears in standard monospace. Any code that is serving as a placeholder appears in *italic monospace*.

SQL code and keywords have been placed in uppercase for your convenience and general consistency. For example:

```
SELECT * FROM PRODUCTS_TBL;
```

```
PROD_ID    PROD_DESC                                COST
-----
11235     WITCHES COSTUME                          29.99
222       PLASTIC PUMPKIN 18 INCH                   7.75
13        FALSE PARAFFIN TEETH                      1.1
90        LIGHTED LANTERNS                          14.5
15        ASSORTED COSTUMES                          10
9         CANDY CORN                                1.35
6         PUMPKIN CANDY                             1.45
87        PLASTIC SPIDERS                           1.05
119       ASSORTED MASKS                            4.95
```

9 rows selected.

The following special design features enhance the text:

There are syntax boxes to draw your attention to the syntax of the commands discussed during each hour.

```
SELECT [ ALL | * | DISTINCT COLUMN1, COLUMN2 ]
FROM TABLE [ , TABLE2 ];
```





































































## Q&A

- Q.** *How is it that I can enter numbers such as a person's Social Security number in fields defined as character fields?*
- A.** Numeric values are still alphanumeric, which are allowed in string data types. The process is called an implicit conversion because it is handled automatically by the database system. Typically, the only data stored as numeric values are values used in computations. However, it might be helpful for some to define all numeric fields with a numeric data type to help control the data entered in that field.
- Q.** *I still do not understand the difference between constant-length and varying-length data types. Can you explain?*
- A.** Say you have an individual's last name defined as a constant-length data type with a length of 20 bytes. Suppose the individual's name is Smith. When the data is inserted into the table, 20 bytes are taken, 5 for the name and 15 for the extra spaces (remember that this is a constant-length data type). If you use a varying-length data type with a length of 20 and inserted Smith, only 5 bytes of space are taken. If you then imagine that you are inserting 100,000 rows of data into this system, you could possibly save 1.5 million bytes of data.
- Q.** *Are there limits on the lengths of data types?*
- A.** Yes, there are limits on the lengths of data types, and they do vary among the various implementations.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. You may refer to Appendix C, "Answers to Quizzes and Exercises," for answers.









## What Is a Schema?

A *schema* is a collection of database objects (as far as this hour is concerned—tables) associated with one particular database username. This username is called the *schema owner*, or the owner of the related group of objects. You may have one or multiple schemas in a database. The user is only associated with the schema of the same name and often the terms will be used interchangeably. Basically, any user who creates an object has just created it in her own schema unless she specifically instructs it to be created in another one. So, based on a user's privileges within the database, the user has control over objects that are created, manipulated, and deleted. A schema can consist of a single table and has no limits to the number of objects that it may contain, unless restricted by a specific database implementation.

Say you have been issued a database username and password by the database administrator. Your username is USER1. Suppose you log on to the database and then create a table called EMPLOYEE\_TBL. According to the database, your table's actual name is USER1.EMPLOYEE\_TBL. The schema name for that table is USER1, which is also the owner of that table. You have just created the first table of a schema.

The good thing about schemas is that when you access a table that you own (in your own schema), you do not have to refer to the schema name. For instance, you could refer to your table as either one of the following:

```
EMPLOYEE_TBL  
USER1.EMPLOYEE_TBL
```

The first option is preferred because it requires fewer keystrokes. If another user were to query one of your tables, the user would have to specify the schema, as follows:

```
USER1.EMPLOYEE_TBL
```

In Hour 20, "Creating and Using Views and Synonyms," you learn about the distribution of permissions so that other users can access your tables. You also learn about synonyms, which allow you to give a table another name so you do not have to specify the schema name when accessing a table. Figure 3.1 illustrates two schemas in a relational database.



































































The syntax for inserting values into a limited number of columns in a table is as follows:

```
INSERT INTO TABLE_NAME ('COLUMN1', 'COLUMN2')
VALUES ('VALUE1', 'VALUE2');
```

You use `ORDERS_TBL` and insert values into only specified columns in the following example.

Here is the table structure:

`ORDERS_TBL`

COLUMN NAME	Null?	DATA TYPE
ORD_NUM	NOT NULL	VARCHAR2(10)
CUST_ID	NOT NULL	VARCHAR2(10)
PROD_ID	NOT NULL	VARCHAR2(10)
QTY	NOT NULL	NUMBER(4)
ORD_DATE		DATE

Here is the sample INSERT statement:

```
insert into orders_tbl (ord_num,cust_id,prod_id,qty)
values ('23A16','109','7725',2);
```

1 row created.

You have specified a column list enclosed by parentheses after the table name in the INSERT statement. You have listed all columns into which you want to insert data. `ORD_DATE` is the only excluded column. If you look at the table definition, you can see that `ORD_DATE` does not require data for every record in the table. You know that `ORD_DATE` does not require data because `NOT NULL` is not specified in the table definition. `NOT NULL` tells us that `NULL` values are not allowed in the column. Furthermore, the list of values must appear in the same order as the column list.

The column list in the INSERT statement does not have to reflect the same order of columns as in the definition of the associated table, but the list of values must be in the order of the associated columns in the column list.

## Inserting Data from Another Table

You can insert data into a table based on the results of a query from another table using a combination of the INSERT statement and the SELECT statement. Briefly, a *query* is an inquiry to the database that either expects or does not expect data to be returned. See Hour 7 for more information on queries. A query is a question that the

user asks the database, and the data returned is the answer. In the case of combining the INSERT statement with the SELECT statement, you are able to insert the data retrieved from a query into a table.

The syntax for inserting data from another table is

```
insert into table_name (('column1', 'column2'))
select *|('column1', 'column2'))
from table_name
[where condition(s)];
```

You see three new keywords in this syntax, which are covered here briefly. These keywords are SELECT, FROM, and WHERE. SELECT is the main command used to initiate a query in SQL. FROM is a clause in the query that specifies the names of tables in which the target data should be found. The WHERE clause, also part of the query, is used to place conditions on the query itself. A *condition* is a way of placing criteria on data affected by a SQL statement. An example condition might state: WHERE NAME = 'SMITH'. These three keywords are covered extensively during Hour 7 and Hour 8, “Using Operators to Categorize Data.”

The following example uses a simple query to view all data in the PRODUCTS\_TBL table. SELECT \* tells the database server that you want information on all columns of the table. Because no WHERE clause is used, you will see all records in the table as well.

```
select * from products_tbl;
PROD_ID   PROD_DESC                               COST
-----
11235     WITCHES COSTUME                         29.99
222       PLASTIC PUMPKIN 18 INCH                 7.75
13        FALSE PARAFFIN TEETH                    1.1
90        LIGHTED LANTERNS                        14.5
15        ASSORTED COSTUMES                       10
9         CANDY CORN                              1.35
6         PUMPKIN CANDY                           1.45
87        PLASTIC SPIDERS                         1.05
119       ASSORTED MASKS                          4.95
1234     KEY CHAIN                               5.95
2345     OAK BOOKSHELF                           59.99
```

11 rows selected.

Now, insert values into the PRODUCTS\_TMP table based on the preceding query. You can see that 11 rows are created in the temporary table.

```
insert into products_tmp
select * from products_tbl;
```

11 rows created.

## HOOR 5: Manipulating Data

You must ensure that the columns returned from the SELECT query are in the same order as the columns that you have in your table or INSERT statement. Additionally, double-check that the data from the SELECT query is compatible with the data type of the column that it is inserting into the table. For example, trying to insert a VARCHAR field with 'ABC' into a numeric column would cause your statement to fail.

The following query shows all data in the PRODUCTS\_TMP table that you just inserted:

```
select * from products_tmp;
PROD_ID  PROD_DESC                                COST
-----
11235    WITCHES COSTUME                          29.99
222      PLASTIC PUMPKIN 18 INCH                   7.75
13       FALSE PARAFFIN TEETH                       1.1
90       LIGHTED LANTERNS                          14.5
15       ASSORTED COSTUMES                          10
9        CANDY CORN                                 1.35
6        PUMPKIN CANDY                             1.45
87       PLASTIC SPIDERS                           1.05
119      ASSORTED MASKS                            4.95
1234     KEY CHAIN                                  5.95
2345     OAK BOOKSHELF                             59.99
```

11 rows selected.

## Inserting NULL Values

Inserting a NULL value into a column of a table is a simple matter. You might want to insert a NULL value into a column if the value of the column in question is unknown. For instance, not every person carries a pager, so it would be inaccurate to enter an erroneous pager number—not to mention, you would not be budgeting space. A NULL value can be inserted into a column of a table using the keyword NULL.

The syntax for inserting a NULL value follows:

```
insert into schema.table_name values
('column1', NULL, 'column3');
```

The NULL keyword should be used in the proper sequence of the associated column that exists in the table. That column will not have data in it for that row if you enter NULL. In the syntax, a NULL value is being entered in the place of COLUMN2.

Study the two following examples:

```
insert into orders_tbl (ord_num,cust_id,prod_id,qty,ORD_DATE)
values ('23A16', '109', '7725', 2, NULL);
```

1 row created.

In this example, all columns in which to insert values are listed, which also happen to be every column in the `ORDERS_TBL` table. You insert a `NULL` value for the `ORD_DATE` column, meaning that you either do not know the order date, or there is no order date at this time. Now look at the second example:

```
insert into orders_tbl
values ('23A16', '109', '7725', 2);
```

1 row created.

The second example contains two differences from the first statement, but the results are the same. First, there is not a column list. Remember that a column list is not required if you are inserting data into all columns of a table. Second, instead of inserting the value `NULL` into the `ORD_DATE` column, you simply leave off the last value, which signifies that a `NULL` value should be added. Remember that a `NULL` value signifies an absence of value from a field and is different from an empty string.

## Updating Existing Data

Pre-existing data in a table can be modified using the `UPDATE` command. The `UPDATE` command does not add new records to a table, nor does it remove records—`UPDATE` simply updates existing data. The update is generally used to update one table at a time in a database, but can be used to update multiple columns of a table at the same time. An individual row of data in a table can be updated, or numerous rows of data can be updated in a single statement, depending on what's needed.

### Updating the Value of a Single Column

The most simple form of the `UPDATE` statement is its use to update a single column in a table. Either a single row of data or numerous records can be updated when updating a single column in a table.

The syntax for updating a single column follows:

```
update table_name
set column_name = 'value'
[where condition];
```

The following example updates the `QTY` column in the `ORDERS` table to the new value 1 for the `ORD_NUM` 23A16, which you have specified using the `WHERE` clause:

```
update orders_tbl
set qty = 1
where ord_num = '23A16';
```

1 row updated.

The following example is identical to the previous example, except for the absence of the WHERE clause:

```
update orders_tbl
set qty = 1;
```

11 rows updated.

Notice that in this example, 11 rows of data were updated. You set the QTY to 1, which updated the quantity column in the ORDERS\_TBL table for all rows of data. Is this really what you wanted to do? Perhaps in some cases, but rarely will you issue an UPDATE statement without a WHERE clause. An easy way to check to see whether you are going to be updating the correct dataset or not is to write a SELECT statement for the same table with your WHERE clause that you will be using in the INSERT statement. Then you can physically verify that these are the rows that you want to update.

### **Watch Out!**

Extreme caution must be used when using the UPDATE statement without a WHERE clause. The target column is updated for all rows of data in the table if conditions are not designated using the WHERE clause. In most situations, the use of the WHERE clause with a DML command is appropriate.

## **Updating Multiple Columns in One or More Records**

Next, you see how to update multiple columns with a single UPDATE statement. Study the following syntax:

```
update table_name
set column1 = 'value',
   [column2 = 'value',]
   [column3 = 'value']
[where condition];
```

Notice the use of the SET in this syntax—there is only one SET, but multiple columns. Each column is separated by a comma. You should start to see a trend in SQL. The comma is usually used to separate different types of arguments in SQL statements. In the following code, a comma is used to separate the two columns being updated. Again, the WHERE clause is optional, but usually necessary.

```
update orders_tbl
set qty = 1,
   cust_id = '221'
where ord_num = '23A16';
```

1 row updated.

The SET keyword is used only once for each UPDATE statement. If more than one column is to be updated, a comma is used to separate the columns to be updated.

**By the  
Way**

## Deleting Data from Tables

The DELETE command is used to remove entire rows of data from a table. The DELETE command is not used to remove values from specific columns; a full record, including all columns, is removed. The DELETE statement must be used with caution—as it works all too well.

To delete a single record or selected records from a table, the DELETE statement must be used with the following syntax:

```
delete from table_name
[where condition];
```

```
delete from orders_tbl
where ord_num = '23A16';
```

1 row deleted.

Notice the use of the WHERE clause. The WHERE clause is an essential part of the DELETE statement if you are attempting to remove selected rows of data from a table. You rarely issue a DELETE statement without the use of the WHERE clause. If you do, your results will be similar to the following example:

```
delete from orders_tbl;
```

11 rows deleted.

If the WHERE clause is omitted from the DELETE statement, all rows of data are deleted from the table. As a general rule, always use a WHERE clause with the DELETE statement. Additionally, test your WHERE clause with a SELECT statement first.

Also, remember that the DELETE command might have a permanent impact on the database. Ideally, it should be possible to recover erroneously deleted data via a backup, but in some cases, it might be difficult or even impossible to recover data. If data cannot be recovered, it must be re-entered into the database—trivial if dealing with only one row of data, but not so trivial if dealing with thousands of rows of data. Hence, the importance of the WHERE clause.

**Watch  
Out!**

The temporary table that was populated from the original table earlier in this hour can be very useful for testing the DELETE and UPDATE commands before issuing them against the original table.

## Summary

You have learned the three basic commands in DML: the INSERT, UPDATE, and DELETE statements. As you have seen, data manipulation is a very powerful part of SQL, allowing the database user to populate tables with new data, update existing data, and delete data.

A very important lesson when updating or deleting data from tables in a database is sometimes learned when neglecting the use of the WHERE clause. Remember that the WHERE clause places conditions on a SQL statement—particularly in the case of UPDATE and DELETE operations, when you are specifying specific rows of data that will be affected during a transaction. All target table data rows are affected if the WHERE clause is not used, which could be disastrous to the database. Protect your data and be cautious during data manipulation operations.

## Q&A

- Q.** *With all the warnings about DELETE and UPDATE, I'm a little afraid to use them. If I accidentally update all the records in a table because the WHERE clause was not used, can the changes be reversed?*
- A.** There is no reason to be afraid, because there is not much you can do to the database that cannot be corrected, although considerable time and work might be involved. Hour 6, "Managing Database Transactions," discusses the concepts of transactional control, which allows data manipulation operations to either be finalized or undone.
- Q.** *Is the INSERT statement the only way to enter data into a table?*
- A.** No, but remember that the INSERT statement is ANSI standard. The various implementations have their tools to enter data into tables. For example, Oracle has a utility called SQL\*Loader. Also, many of the various implementations have utilities called IMPORT that can be used to insert data. There are many good books on the market that will expand on these utilities.

# Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. Use the EMPLOYEE\_TBL with the following structure:

column	data type	(not)null	
last_name	varchar2(20)	not null	
first_name	varchar2(20)	not null	
ssn	char(9)	not null	
phone	number(10)	null	
LAST_NAME	FIRST_NAME	SSN	PHONE
SMITH	JOHN	312456788	3174549923
ROBERTS	LISA	232118857	3175452321
SMITH	SUE	443221989	3178398712
PIERCE	BILLY	310239856	3176763990

What would happen if the following statements were run?

**A.**

```
insert into employee_tbl
('JACKSON', 'STEVE', '313546078', '3178523443');
```

**B.**

```
insert into employee_tbl values
('JACKSON', 'STEVE', '313546078', '3178523443');
```

**C.**

```
insert into employee_tbl values
('MILLER', 'DANIEL', '230980012', NULL);
```

**D.**

```
insert into employee_tbl values
('TAYLOR', NULL, '445761212', '3179221331');
```

**E.**

```
delete from employee_tbl;
```

**F.**

```
delete from employee_tbl  
where last_name = 'SMITH';
```

**G.**

```
delete from employee_tbl  
where last_name = 'SMITH'  
and first_name = 'JOHN';
```

**H.**

```
update employee_tbl  
set last_name = 'CONRAD';
```

**I.**

```
update employee_tbl  
set last_name = 'CONRAD'  
where last_name = 'SMITH';
```

**J.**

```
update employee_tbl  
set last_name = 'CONRAD',  
first_name = 'LARRY';
```

**K.**

```
update employee_tbl  
set last_name = 'CONRAD'  
first_name = 'LARRY'  
where ssn = '313546078';
```

## Exercises

1. Go to Appendix E, “INSERT Statements for Data in Book Examples.” Invoke MySQL as you have done in previous exercises.

Now you need to insert the data into the tables that you created in Hour 3, “Managing Database Objects.” There are two ways to do this. The first method is to type each INSERT statement that is found in Appendix E at the `mysql>` command prompt. This method is recommended if you have the time to do so. The second method is to download the file `tysql24_data.sql` from the website for this book and execute the file from the `mysql>` command prompt.

The syntax to execute `tysql24_data.sql` at the command prompt is as follows:

```
source tysql24_data.sql
```

If you downloaded the file `tysql24_data.sql` to the `mysql` folder on your computer, the syntax to execute this file would be as follows:

```
source c:\mysql\tysql24_data.sql
```

After you have executed the file `tysql24_data.sql`, your tables will be populated with data and you can proceed with the exercises in the rest of this book. If you executed the file `tysql24_data.sql`, you do not have to manually type the `INSERT` statements at the `mysql>` command prompt.

**2.** Use the `PRODUCTS_TBL` for the next exercise.

**A.** Add the following products to the product table:

PROD_ID	PROD_DESC	COST
301	FIREMAN COSTUME	24.99
302	POLICEMAN COSTUME	24.99
303	KIDDIE GRAB BAG	4.99

Write DML to accomplish the following:

- B.** Correct the cost of the two costumes added. The cost should be the same as the witch's costume.
- C.** Now we have decided to cut our product line, starting with the new products. Remove the three products you just added.

*This page intentionally left blank*

## HOUR 6

# Managing Database Transactions

In this hour, you learn the concepts behind the management of database transactions.

---

### ***The highlights of this hour include:***

- ▶ The definition of a transaction
- ▶ The commands used to control transactions
- ▶ The syntax and examples of transaction commands
- ▶ When to use transactional commands
- ▶ The consequences of poor transactional control

## **What Is a Transaction?**

A *transaction* is a unit of work that is performed against a database. Transactions are units or sequences of work accomplished in a logical order, whether in a manual fashion by a user or automatically by some sort of a database program. In a relational database using SQL, transactions are accomplished using the Data Manipulation Language (DML) commands that were discussed during Hour 5, “Manipulating Data” (INSERT, UPDATE, and DELETE). A transaction is the propagation of one or more changes to the database. For instance, you are performing a transaction if you perform an UPDATE statement on a table to change an individual’s name.

A transaction can either be one DML statement or a group of statements. When managing transactions, each designated transaction (group of DML statements) must be successful as one entity or none of them will be successful.

The following list describes the nature of transactions:

- ▶ All transactions have a beginning and an end.
- ▶ A transaction can be saved or undone.
- ▶ If a transaction fails in the middle, no part of the transaction can be saved to the database.

**By the  
Way**

Starting or executing transactions is implementation specific. You must check your particular implementation for how to begin transactions.

## Controlling Transactions

*Transactional control* is the capability to manage various transactions that might occur within a relational database management system. When you speak of transactions, you are referring to the INSERT, UPDATE, and DELETE commands, which were covered during the previous hour.

When a transaction is executed and completes successfully, the target table is not immediately changed, although it might appear so according to the output. When a transaction successfully completes, transactional control commands are used to finalize the transaction, either saving the changes made by the transaction to the database or reversing the changes made by the transaction.

Three commands are used to control transactions:

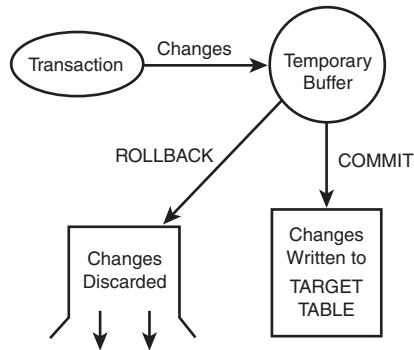
- ▶ COMMIT
- ▶ ROLLBACK
- ▶ SAVEPOINT

Each of these is discussed in detail in the following sections.

**By the  
Way**

Transactional control commands are only used with the DML commands INSERT, UPDATE, and DELETE. For example, you do not issue a COMMIT statement after creating a table. When the table is created, it is automatically committed to the database. Likewise, you cannot issue a ROLLBACK statement to replenish a table that was just dropped.

When a transaction has completed, the transactional information is stored either in an allocated area or in a temporary rollback area in the database. All changes are held in this temporary rollback area until a transactional control command is issued. When a transactional control command is issued, changes are either made to the database or discarded; then, the temporary rollback area is emptied. Figure 6.1 illustrates how changes are applied to a relational database.



**FIGURE 6.1**  
Rollback area.

## The COMMIT Command

The COMMIT command is the transactional command used to save changes invoked by a transaction to the database. The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.

The syntax for this command is

```
commit [ work ];
```

The keyword COMMIT is the only mandatory part of the syntax, along with the character or command used to terminate a statement according to each implementation. WORK is a keyword that is completely optional; its only purpose is to make the command more user-friendly.

In the following example, you begin by selecting all data from the PRODUCT\_TMP table:

```
SELECT * FROM PRODUCTS_TMP;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35

## HOOR 6: Managing Database Transactions

6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

Next, you delete all records from the table where the product cost is less than \$14.00.

```
DELETE FROM PRODUCTS_TMP
WHERE COST < 14;
```

8 rows deleted.

A COMMIT statement is issued to save the changes to the database, completing the transaction.

```
COMMIT;
```

Commit complete.

### Watch Out!

Frequent COMMIT statements in large loads or unloads of the database are highly recommended; however, too many COMMIT statements cause the job to take a lot of extra time to complete. Remember that all changes are sent to the temporary rollback area first. If this temporary rollback area runs out of space and cannot store information about changes made to the database, the database will probably halt, disallowing further transactional activity.

### By the Way

In some implementations, transactions are committed without issuing the COMMIT command—instead, merely signing out of the database causes a commit to occur. However, in some implementations, such as MySQL, after you perform a SET TRANSACTION command, the auto-commit functionality will not resume until it has received a COMMIT or ROLLBACK statement.

## The ROLLBACK Command

The ROLLBACK command is the transactional control command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued.

The syntax for the ROLLBACK command is as follows:

```
rollback [ work ];
```

Once again, as in the COMMIT statement, the WORK keyword is an optional part of the ROLLBACK syntax.

In the following example, you begin by selecting all records from the PRODUCTS\_TMP table since the previous deletion of 14 records:

```
SELECT * FROM PRODUCTS_TMP;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

3 rows selected.

Next, you update the table, changing the product cost to \$39.99 for the product identification number 11235:

```
update products_tmp  
set cost = 39.99  
where prod_id = '11235';
```

1 row updated.

If you perform a quick query on the table, the change appears to have occurred:

```
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	39.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

3 rows selected.

Now, issue the ROLLBACK statement to undo the last change:

```
rollback;
```

Rollback complete.

Finally, verify that the change was not committed to the database:

```
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

3 rows selected

## The SAVEPOINT Command

A *savepoint* is a point in a transaction where you can roll the transaction back to this point without rolling back the entire transaction.

The syntax for the SAVEPOINT command is

```
savepoint savepoint_name
```

This command serves only to create a savepoint among transactional statements. The ROLLBACK command is used to undo a group of transactions. The savepoint is a way of managing transactions by breaking large numbers of transactions into smaller, more manageable groups.

The savepoint name must be unique to the associated group of transactions. However, the savepoint can have the same name as a table or other object. Refer to specific implementation documentation for more details on naming conventions. Otherwise, savepoint names are a matter of personal preference and are used only by the database application developer to manage groups of transactions.

## The ROLLBACK TO SAVEPOINT Command

The syntax for rolling back to a savepoint is as follows:

```
ROLLBACK TO SAVEPOINT_NAME;
```

In this example, you are going to delete the remaining three records from the PRODUCTS\_TMP table. You want to issue a SAVEPOINT command before each delete, so you can issue a ROLLBACK command to any savepoint at any time to return the appropriate data to its original state:

```
savepoint sp1;  
Savepoint created.  
delete from products_tmp where prod_id = '11235';  
1 row deleted.  
savepoint sp2;  
Savepoint created.  
delete from products_tmp where prod_id = '90';  
1 row deleted.  
savepoint sp3;  
Savepoint created.  
delete from products_tmp where prod_id = '2345';  
1 row deleted.
```

Now that the three deletions have taken place, let's say you have changed your mind and decided to issue a ROLLBACK command to the savepoint that you identified as SP2. Because SP2 was created after the first deletion, the last two deletions are undone:

```
rollback to sp2;  
Rollback complete.
```

Notice that only the first deletion took place because you rolled back to SP2:

```
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

```
2 rows selected.
```

Remember, the ROLLBACK command by itself will roll back to the last COMMIT or ROLLBACK statement. You have not yet issued a COMMIT, so all deletions are undone, as in the following example:

```
rollback;  
Rollback complete.  
select * from products_tmp;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
90	LIGHTED LANTERNS	14.5
2345	OAK BOOKSHELF	59.99

3 rows selected.

## The RELEASE SAVEPOINT Command

The `RELEASE SAVEPOINT` command is used to remove a savepoint that you have created. After a savepoint has been released, you can no longer use the `ROLLBACK` command to undo transactions performed since the savepoint. You might want to issue a `RELEASE SAVEPOINT` command to avoid the accidental rollback to a savepoint that is no longer needed.

```
RELEASE SAVEPOINT savepoint_name;
```

## The SET TRANSACTION Command

The `SET TRANSACTION` command can be used to initiate a database transaction. This command is used to specify characteristics for the transaction that follows. For example, you can specify a transaction to be read only or read write. For example:

```
SET TRANSACTION READ WRITE;  
SET TRANSACTION READ ONLY;
```

`READ WRITE` is used for transactions that are allowed to query and manipulate data in the database. `READ ONLY` is used for transactions that require query-only access. `READ ONLY` is useful for report generation and for increasing the speed at which transactions are accomplished. If a transaction is `READ WRITE`, the database must create locks on database objects to maintain data integrity in case multiple transactions are happening concurrently. If a transaction is `READ ONLY`, no locks are established by the database, thereby improving transaction performance.

Other characteristics can be set for a transaction that are out of the scope of this book. MySQL supports this syntax for setting an isolation level for the transaction but in slightly different syntax. For more information, see the documentation for your implementation of SQL.

## Transactional Control and Database Performance

Poor transactional control can hurt database performance and even bring the database to a halt. Repeated poor database performance might be due to a lack of transactional control during large inserts, updates, or deletes. Large batch processes also cause temporary storage for rollback information to grow until either a `COMMIT` or `ROLLBACK` command is issued.

When a `COMMIT` is issued, rollback transactional information is written to the target table and the rollback information in temporary storage is cleared. When a `ROLLBACK` is issued, no changes are made to the database and the rollback information in the temporary storage is cleared. If neither a `COMMIT` nor `ROLLBACK` is issued, the temporary storage for rollback information continues to grow until there is no more space left, thus forcing the database to stop all processes until space is freed. Although space usage is ultimately controlled by the database administrator (DBA), a lack of transactional control can still cause database processing to stop, sometimes forcing the DBA to take action that might consist of killing running user processes.

### Summary

During this hour, you learned the preliminary concepts of transactional management through the use of three transactional control commands: `COMMIT`, `ROLLBACK`, and `SAVEPOINT`. `COMMIT` is used to save a transaction to the database. `ROLLBACK` is used to undo a transaction that was performed. `SAVEPOINT` is used to break a transaction or transactions into groups, allowing you to roll back to specific logical points in transaction processing.

Remember that you should frequently use the `COMMIT` and `ROLLBACK` commands when running large transactional jobs to keep space free in the database. Also, keep in mind that these transactional commands are used only with the three DML commands (`INSERT`, `UPDATE`, and `DELETE`).

## Q&A

**Q. *Is it necessary to issue a commit after every INSERT statement?***

**A.** No, absolutely not. If you were inserting a few hundred thousand rows into a table, a COMMIT would be recommended every 5,000–10,000 rows, depending on the size of the temporary rollback area (seek the advice of your database administrator). Remember that the database might freeze up or not function properly when the rollback area fills up.

**Q. *How does the ROLLBACK command undo a transaction?***

**A.** The ROLLBACK command clears all changes from the rollback area.

**Q. *If I issue a transaction and 99% of the transaction completes but the other 1% errs, will I be able to redo only the error part?***

**A.** No, the entire transaction must succeed; otherwise, data integrity is compromised.

**Q. *A transaction is permanent after I issue a COMMIT, but can't I change data with an UPDATE command?***

**A.** The word *permanent* used in this matter means that it is now a part of the database. The UPDATE statement can always be used to make modifications or corrections to the data.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. True or false: If you have committed several transactions, have several more transactions that have not been committed, and issue a ROLLBACK command, all your transactions for the same session are undone.
2. True or false: A SAVEPOINT command actually saves transactions after a specified amount of transactions have executed.
3. Briefly describe the purpose of each one of the following commands: COMMIT, ROLLBACK, and SAVEPOINT.

## Exercises

1. Take the following transactions and create a SAVEPOINT command after the first three transactions. Then place a ROLLBACK statement to your savepoint at the end. Try to determine what the CUSTOMER\_TBL will look like after you are done.

```
INSERT INTO CUSTOMER_TBL VALUES (615, 'FRED WOLF', '109 MEMORY  
LANE', 'PLAINFIELD', 'IN', 46113, '3175555555', NULL);  
INSERT INTO CUSTOMER_TBL VALUES (559, 'RITA THOMPSON', '125  
PEACHTREE', 'INDIANAPOLIS', 'IN', 46248, '3171111111', NULL);  
INSERT INTO CUSTOMER_TBL VALUES (715, 'BOB DIGGLER', '1102 HUNTINGTON  
ST', 'SHELBY', 'IN', 41234, '3172222222', NULL);  
UPDATE CUSTOMER_TBL SET CUST_NAME='FRED WOLF' WHERE CUST_ID='559';  
UPDATE CUSTOMER_TBL SET CUST_ADDRESS='APT C 4556 WATERWAY' WHERE  
CUST_ID='615';  
UPDATE CUSTOMER_TBL SET CUST_CITY='CHICAGO' WHERE CUST_ID='715';
```

2. Take the following group of transactions and create a savepoint after the first three transactions.

Then place a COMMIT statement at the end, followed by a ROLLBACK statement to your savepoint. What do you think should happen?

```
UPDATE CUSTOMER_TBL SET CUST_NAME='FRED WOLF' WHERE CUST_ID='559';  
UPDATE CUSTOMER_TBL SET CUST_ADDRESS='APT C 4556 WATERWAY' WHERE  
CUST_ID='615';  
UPDATE CUSTOMER_TBL SET CUST_CITY='CHICAGO' WHERE CUST_ID='715';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='615';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='559';  
DELETE FROM CUSTOMER_TBL WHERE CUST_ID='615';
```

*This page intentionally left blank*

## PART III

# Getting Effective Results from Queries

<b>HOUR 7</b>	Introduction to the Database Query	<b>101</b>
<b>HOUR 8</b>	Using Operators to Categorize Data	<b>117</b>
<b>HOUR 9</b>	Summarizing Data Results from a Query	<b>141</b>
<b>HOUR 10</b>	Sorting and Grouping Data	<b>151</b>
<b>HOUR 11</b>	Restructuring the Appearance of Data	<b>165</b>
<b>HOUR 12</b>	Understanding Dates and Times	<b>185</b>

*This page intentionally left blank*

## HOUR 7

# Introduction to the Database Query

In this seventh hour, you will learn about database queries, which involve the use of the SELECT statement. The SELECT statement is the most frequently used of all SQL commands after a database's establishment. The SELECT statement allows you to view data that is stored in the database.

---

### ***The highlights of this hour include:***

- ▶ What a database query is
- ▶ How to use the SELECT statement
- ▶ Adding conditions to queries using the WHERE clause
- ▶ Using column aliases
- ▶ Selecting data from another user's table

## What Is a Query?

A *query* is an inquiry into the database using the SELECT statement. A query is used to extract data from the database in a readable format according to the user's request. For instance, if you have an employee table, you might issue a SQL statement that returns the employee who is paid the most. This request to the database for usable employee information is a typical query that can be performed in a relational database.

## Introduction to the SELECT Statement

The SELECT statement, the command that represents Data Query Language (DQL) in SQL, is the basic statement used to construct database queries. The SELECT statement is not a

standalone statement, which means that one or more additional clauses (elements) are required for a syntactically correct query. In addition to the required clauses, there are optional clauses that increase the overall functionality of the SELECT statement. The SELECT statement is by far one of the most powerful statements in SQL. The FROM clause is a mandatory clause and must always be used in conjunction with the SELECT statement.

There are four keywords, or *clauses*, that are valuable parts of a SELECT statement. These keywords are as follows:

- ▶ SELECT
- ▶ FROM
- ▶ WHERE
- ▶ ORDER BY

Each of these keywords is covered in detail during the following sections.

## The SELECT Statement

The SELECT statement is used in conjunction with the FROM clause to extract data from the database in an organized, readable format. The SELECT part of the query is for selecting the data you want to see according to the columns in which they are stored in a table.

The syntax for a simple SELECT statement is as follows:

```
SELECT [ * | ALL | DISTINCT COLUMN1, COLUMN2 ]  
FROM TABLE1 [ , TABLE2 ];
```

The SELECT keyword in a query is followed by a list of columns that you want displayed as part of the query output. The asterisk (\*) is used to denote that all columns in a table should be displayed as part of the output. Check your particular implementation for its usage. The ALL option is used to display all values for a column, including duplicates. The DISTINCT option is used to suppress duplicate rows from being displayed in the output. The ALL option is considered an inferred option, meaning that it is considered the default and therefore does not necessarily need to be used in the SELECT statement. The FROM keyword is followed by a list of one or more tables from which you want to select data. Notice that the columns following the SELECT clause are separated by commas, as is the table list following the FROM clause.

Commas are used to separate arguments in a list in SQL statements. *Arguments* are values that are either required or optional to the syntax of a SQL statement or command. Some common lists include lists of columns in a query, lists of tables to be selected from in a query, values to be inserted into a table, and values grouped as a condition in a query's WHERE clause.

Explore the basic capabilities of the SELECT statement by studying the following examples. First, perform a simple query from the PRODUCTS\_TBL table:

```
SELECT * FROM PRODUCTS_TBL;
```

PROD_ID	PROD_DESC	COST
-----		
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

The asterisk represents all columns in the table, which, as you can see, are displayed in the form PROD\_ID, PROD\_DESC, and COST. Each column in the output is displayed in the order that it appears in the table. There are 11 records in this table, identified by the feedback 11 rows selected. This feedback differs among implementations; for example, another feedback for the same query would be 11 rows affected.

Now select data from another table, CANDY\_TBL. Create this table in the image of the PRODUCTS\_TBL table for the following examples. List the column name after the SELECT keyword to display only one column in the table:

```
SELECT PROD_DESC FROM CANDY_TBL;
```

```
PROD_DESC
-----
CANDY CORN
CANDY CORN
HERSHEYS KISS
SMARTIES
```

4 rows selected.

Four records exist in the CANDY\_TBL table. The next statement uses the ALL option to show you that the ALL is optional and redundant. There is never a need to specify ALL; it is a default option.

```
SELECT ALL PROD_DESC
FROM CANDY_TBL;
```

```
PROD_DESC
-----
CANDY CORN
CANDY CORN
HERSHEYS KISS
SMARTIES
```

4 rows selected.

The DISTINCT option is used in the following statement to suppress the display of duplicate records. Notice that the value CANDY CORN is only printed once in this example.

```
SELECT DISTINCT PROD_DESC
FROM CANDY_TBL;
```

```
PROD_DESC
-----
CANDY CORN
HERSHEYS KISS
SMARTIES
```

3 rows selected.

DISTINCT and ALL can also be used with parentheses enclosing the associated column. The use of parentheses is often used in SQL—as well as many other languages—to improve readability.

```
SELECT DISTINCT(PROD_DESC)
FROM CANDY_TBL;
```

```
PROD_DESC
-----
CANDY CORN
HERSHEYS KISS
SMARTIES
```

3 rows selected.

## **The FROM Clause**

The FROM clause must be used in conjunction with the SELECT statement. It is a required element for any query. The FROM clause's purpose is to tell the database what table(s) to access to retrieve the desired data for the query. The FROM clause may contain one or more tables. The FROM clause must always list at least one table.

The syntax for the FROM clause is as follows:

```
from table1 [ , table2 ]
```

## The WHERE Clause

A *condition* is part of a query that is used to display selective information as specified by the user. The value of a condition is either TRUE or FALSE, thereby limiting the data received from the query. The WHERE clause is used to place conditions on a query by eliminating rows that would normally be returned by a query without conditions.

There can be more than one condition in the WHERE clause. If there is more than one condition, they are connected by the AND and OR operators, which are discussed during Hour 8, “Using Operators to Categorize Data.” As you also learn during the next hour, several conditional operators exist that can be used to specify conditions in a query. This hour only deals with a single condition for each query.

An *operator* is a character or keyword in SQL that is used to combine elements in a SQL statement.

The syntax for the WHERE clause is as follows:

```
select [ all | * | distinct column1, column2 ]
from table1 [ , table2 ]
where [ condition1 | expression1 ]
[ and|OR condition2 | expression2 ]
```

The following is a simple SELECT statement without conditions specified by the WHERE clause:

```
SELECT *
FROM PRODUCTS_TBL;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

Now add a condition for the same query.

```
SELECT * FROM PRODUCTS_TBL
WHERE COST < 5;
```

PROD_ID	PROD_DESC	COST
13	FALSE PARAFFIN TEETH	1.1
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95

5 rows selected.

The only records displayed are those that cost less than \$5.

In the following query, you want to display the product description and cost that matches the product identification 119.

```
SELECT PROD_DESC, COST
FROM PRODUCTS_TBL
WHERE PROD_ID = '119';
```

PROD_DESC	COST
ASSORTED MASKS	4.95

1 row selected.

## The ORDER BY Clause

You usually want your output to have some kind of order. Data can be sorted by using the `ORDER BY` clause. The `ORDER BY` clause arranges the results of a query in a listing format you specify. The default ordering of the `ORDER BY` clause is an *ascending order*; the sort displays in the order A–Z if it's sorting output names alphabetically. A *descending order* for alphabetical output would be displayed in the order Z–A. Ascending order for output for numeric values between 1 and 9 would be displayed 1–9; descending order is displayed as 9–1.

SQL sorts are ASCII, character-based sorts. The numeric values 0–9 would be sorted as character values and sorted before the characters A–Z. Because numeric values are treated like characters during a sort, the following list of numeric values would be sorted in the following order: 1, 12, 2, 255, 3.

The syntax for the ORDER BY clause is as follows:

```
select [ all | * | distinct column1, column2 ]
from table1 [ , table2 ]
where [ condition1 | expression1 ]
[ and|OR condition2 | expression2 ]
ORDER BY column1|integer [ ASC|DESC ]
```

Begin your exploration of the ORDER BY clause with an extension of one of the previous statements. You will order the product description in ascending order, or alphabetical order. Note the use of the ASC option. ASC can be specified after any column in the ORDER BY clause.

```
SELECT PROD_DESC, PROD_ID, COST
FROM PRODUCTS_TBL
WHERE COST < 20
ORDER BY PROD_DESC ASC;
```

PROD_DESC	PROD_ID	COST
ASSORTED COSTUMES	15	10
ASSORTED MASKS	119	4.95
CANDY CORN	9	1.35
FALSE PARAFFIN TEETH	13	1.1
LIGHTED LANTERNS	90	14.5
PLASTIC PUMPKIN 18 INCH	222	7.75
PLASTIC SPIDERS	87	1.05
PUMPKIN CANDY	6	1.45

8 rows selected.

Because ascending order for output is the default, ASC does not have to be specified.

**By the  
Way**

You can use DESC, as in the following statement, if you want the same output to be sorted in reverse alphabetical order.

```
SELECT PROD_DESC, PROD_ID, COST
FROM PRODUCTS_TBL
WHERE COST < 20
ORDER BY PROD_DESC DESC;
```

PROD_DESC	PROD_ID	COST
PUMPKIN CANDY	6	1.45
PLASTIC SPIDERS	87	1.05
PLASTIC PUMPKIN 18 INCH	222	7.75
LIGHTED LANTERNS	90	14.5
FALSE PARAFFIN TEETH	13	1.1
CANDY CORN	9	1.35
ASSORTED MASKS	119	4.95
ASSORTED COSTUMES	15	10

8 rows selected.

Shortcuts do exist in SQL. A column listed in the ORDER BY clause can be abbreviated with an integer. The *integer* is a substitution for the actual column name (an alias for the purpose of the sort operation), identifying the position of the column after the SELECT keyword.

An example of using an integer as an identifier in the ORDER BY clause follows:

```
SELECT PROD_DESC, PROD_ID, COST
FROM PRODUCTS_TBL
WHERE COST < 20
ORDER BY 1;
```

PROD_DESC	PROD_ID	COST
ASSORTED COSTUMES	15	10
ASSORTED MASKS	119	4.95
CANDY CORN	9	1.35
FALSE PARAFFIN TEETH	13	1.1
LIGHTED LANTERNS	90	14.5
PLASTIC PUMPKIN 18 INCH	222	7.75
PLASTIC SPIDERS	87	1.05
PUMPKIN CANDY	6	1.45

8 rows selected.

In this query, the integer 1 represents the column PROD\_DESC. The integer 2 represents the PROD\_ID column, 3 represents the COST column, and so on.

You can order by multiple columns in a query, using either the column name itself or the associated number of the column in the SELECT:

```
ORDER BY 1,2,3
```

Columns in an ORDER BY clause are not required to appear in the same order as the associated columns following the SELECT, as shown by the following example:

```
ORDER BY 1,3,2
```

The order in which the columns are specified within the ORDER BY clause will be the manner in which the ordering process is done. So the statement below would first order by the PROD\_DESC column and then by the COST column.

```
ORDER BY PROD_DESC,COST
```

## Case Sensitivity

Case sensitivity is a very important concept to understand when coding with SQL. Typically, SQL commands and keywords are not case-sensitive, which allows you to enter your commands and keywords in either uppercase or lowercase—whatever you

prefer. The case may also be mixed (both uppercase and lowercase for a single word or statement). See Hour 5, “Manipulating Data,” on case sensitivity.

Case sensitivity is, however, a factor when dealing with data in SQL. In most situations, data seems to be stored exclusively in uppercase in a relational database to provide data consistency.

For instance, your data would not be consistent if you arbitrarily entered your data using random case:

```
SMITH
Smith
smith
```

If the last name was stored as `smith` and you issued a query as follows, no rows would be returned:

```
SELECT *
FROM EMPLOYEE_TBL
WHERE LAST_NAME = 'SMITH';
```

You must use the same case in your query as the case the data is stored in when referencing data in the database. When entering data, consult the rules set forth by your company for the appropriate case to be used. The way data is stored varies widely among organizations.

***By the  
Way***

## Examples of Simple Queries

This section provides several examples of queries based on the concepts that have been discussed. The hour begins with the simplest query you can issue and builds upon the initial query progressively. You use the `EMPLOYEE_TBL` table.

Select all records from a table and display all columns:

```
SELECT * FROM EMPLOYEE_TBL;
```

Select all records from a table and display a specified column:

```
SELECT EMP_ID
FROM EMPLOYEE_TBL;
```

Select all records from a table and display a specified column. You can enter code on one line or use a carriage return as desired:

```
SELECT EMP_ID FROM EMPLOYEE_TBL;
```

Select all records from a table and display multiple columns separated by commas:

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL;
```

Display data for a given condition:

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '333333333';
```

Display data for a given condition and sort the output:

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE CITY = 'INDIANAPOLIS'  
ORDER BY EMP_ID;
```

Display data for a given condition and sort the output on multiple columns, one column sorted in reverse order:

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE CITY = 'INDIANAPOLIS'  
ORDER BY EMP_ID, LAST_NAME DESC;
```

Display data for a given condition and sort the output using an integer in the place of the spelled-out column name:

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE CITY = 'INDIANAPOLIS'  
ORDER BY 1;
```

Display data for a given condition and sort the output by multiple columns using integers. The order of the columns in the sort is different than their corresponding order after the SELECT keyword:

```
SELECT EMP_ID, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE CITY = 'INDIANAPOLIS'  
ORDER BY 2, 1;
```

When selecting all rows of data from a large table, the results could return a substantial amount of data.

**By the  
Way**

## Counting the Records in a Table

A simple query can be issued on a table to get a quick count of the number of records in the table or on the number of values for a column in the table. A count is accomplished by the function `COUNT`. Although functions are not discussed until later in this book, this function should be introduced here because it is often a part of one of the simplest queries that you can create.

The syntax of the `COUNT` function is as follows:

```
SELECT COUNT(*)  
FROM TABLE_NAME;
```

The `COUNT` function is used with parentheses, which are used to enclose the target column to count or the asterisk to count all rows of data in the table.

Counting the number of records in the `PRODUCTS_TBL` table:

```
SELECT COUNT(*) FROM PRODUCTS_TBL;
```

```
COUNT(*)  
-----  
          9
```

1 row selected.

Counting the number of values for `PROD_ID` in the `PRODUCTS_TBL` table:

```
SELECT COUNT(PROD_ID) FROM PRODUCTS_TBL;
```

```
COUNT(PROD_ID)  
-----  
          9
```

1 row selected.

Interesting note: Counting the number of values for a column is the same as counting the number of records in a table, if the column being counted is `NOT NULL` (a required column). However, `COUNT(*)` is typically used for counting the number of rows for a table.

**By the  
Way**

## Selecting Data from Another User's Table

Permission must be granted to a user to access another user's table. If no permission has been granted, access is not allowed. You can select data from another user's table after access has been granted (the GRANT command is discussed in Hour 20, "Creating and Using Views and Synonyms"). To access another user's table in a SELECT statement, you must precede the table name with the schema name or the username that owns (created) the table, as in the following example:

```
SELECT EMP_ID
FROM SCHEMA.EMPLOYEE_TBL;
```

**By the  
Way**

If a synonym exists in the database for the table to which you desire access, you do not have to specify the schema name for the table. *Synonyms* are alternate names for tables, which are discussed in Hour 21, "Working with the System Catalog."

## Using Column Aliases

*Column aliases* are used to temporarily rename a table's columns for the purpose of a particular query. The following syntax illustrates the use of column aliases:

```
SELECT COLUMN_NAME ALIAS_NAME
FROM TABLE_NAME;
```

The following example displays the product description twice, giving the second column an alias named PRODUCT. Notice the column headers in the output.

```
select prod_desc,
       prod_desc product
from products_tbl;
```

PROD_DESC	PRODUCT
-----	
WITCHES COSTUME	WITCHES COSTUME
PLASTIC PUMPKIN 18 INCH	PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH	FALSE PARAFFIN TEETH
LIGHTED LANTERNS	LIGHTED LANTERNS
ASSORTED COSTUMES	ASSORTED COSTUMES
CANDY CORN	CANDY CORN
PUMPKIN CANDY	PUMPKIN CANDY
PLASTIC SPIDERS	PLASTIC SPIDERS
ASSORTED MASKS	ASSORTED MASKS
KEY CHAIN	KEY CHAIN
OAK BOOKSHELF	OAK BOOKSHELF

11 rows selected.

Column aliases can be used to customize names for column headers and can also be used to reference a column with a shorter name in some SQL implementations.

When a column is renamed in a SELECT statement, the name is not a permanent change. The change is only for that particular SELECT statement.

**By the  
Way**

## Summary

You have been introduced to the database query, a means for obtaining useful information from a relational database. The SELECT statement, which is known as the Data Query Language (DQL) command, is used to create queries in SQL. The FROM clause must be included with every SELECT statement. You have learned how to place a condition on a query using the WHERE clause and how to sort data using the ORDER BY clause. You have learned the fundamentals of writing queries, and, after a few exercises, you should be prepared to learn more about queries during the next hour.

## Q&A

**Q.** *Why won't the SELECT clause work without the FROM clause?*

**A.** The SELECT clause merely tells the database what data you want to see. The FROM clause tells the database where to get the data.

**Q.** *When I use the ORDER BY clause and choose the option descending, what does that really do to the data?*

**A.** Say that you use the ORDER BY clause and have selected last\_name from the EMPLOYEE\_TBL. If you used the descending option, the order would start with the letter Z and finish with the letter A. Now, let's say that you have used the ORDER BY clause and have selected the salary from the EMPLOYEE\_PAY\_TBL. If you used the descending option, the order would start with the largest salary down to the lowest salary.

**Q.** *What advantage is there to renaming columns?*

**A.** The new column name could fit the description of the returned data more closely for a particular report.

**Q.** *What would be the ordering of the following statement:*

```
SELECT PROD_DESC,PROD_ID,COST FROM PRODUCTS_TBL  
ORDER BY 3,1
```

**A.** The query would be ordered by the `COST` column, and then by the `PROD_DESC` column. Because no ordering preference was specified, they would both be in ascending order.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. Name the required parts for any `SELECT` statement.
2. In the `WHERE` clause, are single quotation marks required for all the data?
3. Under what part of the SQL language does the `SELECT` statement (database query) fall?
4. Can multiple conditions be used in the `WHERE` clause?
5. What is the purpose of the `DISTINCT` option?
6. Is the `ALL` option required?
7. How are numeric characters treated when ordering based upon a character field?

## Exercises

1. Invoke MySQL on your computer. Using your learnsql database, enter the following SELECT statements at the mysql> command prompt. Determine whether the syntax is correct. If the syntax is incorrect, make corrections to the code as necessary. We are using the EMPLOYEE\_TBL here.

**A.**

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME,  
FROM EMPLOYEE_TBL;
```

**B.**

```
SELECT EMP_ID, LAST_NAME  
ORDER BY EMPLOYEE_TBL  
FROM EMPLOYEE_TBL;
```

**C.**

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY EMP_ID;
```

**D.**

```
SELECT EMP_ID SSN, LAST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY 1;
```

**E.**

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '213764555'  
ORDER BY 3, 1, 2;
```

2. Does the following SELECT statement work?

```
SELECT LAST_NAME, FIRST_NAME, PHONE  
FROM EMPLOYEE_TBL  
WHERE EMP_ID = '333333333';
```

3. Write a SELECT statement that returns the name and cost of each product from the PRODUCTS\_TBL. Which product is the most expensive?
4. Write a query that generates a list of all customers and their telephone numbers.

*This page intentionally left blank*

## HOUR 8

# Using Operators to Categorize Data

Operators are used in conjunction with the SELECT command to place extended criteria on data that is returned by a query. Various operators are available to the SQL user that support all data querying needs.

---

### ***The highlights of this hour include:***

- ▶ What is an operator?
- ▶ An overview of operators in SQL
- ▶ How are operators used singularly?
- ▶ How are operators used in combinations?

## **What Is an Operator in SQL?**

An operator is a reserved word or a character used primarily in a SQL statement's WHERE clause to perform operation(s), such as comparisons and arithmetic operations. *Operators* are used to specify conditions in a SQL statement and to serve as conjunctions for multiple conditions in a statement.

The operators discussed during this hour are

- ▶ Comparison operators
- ▶ Logical operators
- ▶ Operators used to negate conditions
- ▶ Arithmetic operators

## Comparison Operators

*Comparison operators* are used to test single values in a SQL statement. The comparison operators discussed consist of =, <>, <, and >.

These operators are used to test

- ▶ Equality
- ▶ Non-equality
- ▶ Less-than values
- ▶ Greater-than values

Examples and the meanings of comparison operators are covered in the following sections.

### Equality

The *equal operator* compares single values to one another in a SQL statement. The equal sign (=) symbolizes equality. When testing for equality, the compared values must match exactly or no data is returned. If two values are equal during a comparison for equality, the returned value for the comparison is TRUE; the returned value is FALSE if equality is not found. This Boolean value (TRUE/FALSE) is used to determine whether data is returned according to the condition.

The = operator can be used by itself or combined with other operators. Remember from the previous chapter that character data comparisons are case sensitive.

The following example shows that salary is equal to 20000:

```
WHERE SALARY = '20000'
```

The following query returns all rows of data where the PROD\_ID is equal to 2345:

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID = '2345';
```

PROD_ID	PROD_DESC	COST
2345	OAK BOOKSHELF	59.99

1 row selected.

## Non-Equality

For every equality, there are multiple non-equalities. In SQL, the operator used to measure non-equality is `<>` (the less than sign combined with the greater than sign). The condition returns `TRUE` if the condition finds non-equality; `FALSE` is returned if equality is found.

Another option comparable to `<>` is `!=`. Many of the major implementations have adopted `!=` to represent not-equal. Check your particular implementation for the usage.

**By the  
Way**

The following example shows that salary is not equal to `20000`:

```
WHERE SALARY <> '20000'
```

The following example shows all of the product information from the products table that do not have the product id of 2345:

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID <> '2345';
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

## Less Than, Greater Than

The symbols `<` (less than) and `>` (greater than) can be used by themselves or in combination with each other or other operators.

The following examples show that salary is less than or greater than to `20000`:

```
WHERE SALARY < '20000'
WHERE SALARY > '20000'
```

## HOOR 8: Using Operators to Categorize Data

In the first example, anything less than and not equal to 20000 returns TRUE. Any value of 20000 or more returns FALSE. Greater than works the opposite of less than.

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > 20;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
2345	OAK BOOKSHELF	59.99

2 rows selected.

In the next example, notice that the value 24.99 was not included in the query's result set. The less than operator is not inclusive.

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST < 24.99;
```

PROD_ID	PROD_DESC	COST
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95

9 rows selected.

## Combinations of Comparison Operators

The equal operator can be combined with the less than and greater than operators.

The following example shows that salary is less than or equal to 20000:

```
WHERE SALARY <= '20000'
```

The next example shows that salary is greater than or equal to 20000:

```
WHERE SALARY >= '20000'
```

Less than or equal to 20000 includes 20000 and all values less than 20000. Any value in that range returns TRUE; any value greater than 20000 returns FALSE. Greater than or equal to also includes the value 20000 in this case and works the same as the <= operator.

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST <= 24.99;
```

PROD_ID	PROD_DESC	COST
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95

9 rows selected.

## Logical Operators

*Logical operators* are those operators that use SQL keywords to make comparisons instead of symbols. The logical operators covered in the following subsections are

- ▶ IS NULL
- ▶ BETWEEN
- ▶ IN
- ▶ LIKE
- ▶ EXISTS
- ▶ UNIQUE
- ▶ ALL and ANY

### IS NULL

The NULL operator is used to compare a value with a NULL value. For example, you might look for employees who do not have a pager by searching for NULL values in the PAGER column of the EMPLOYEE\_TBL table.

The following example compares a value to a NULL value; here, salary has no value:

```
WHERE SALARY IS NULL
```

The following example does not find a NULL value because salary has a value containing the letters *N-U-L-L*:

```
WHERE SALARY = NULL
```

## HOOR 8: Using Operators to Categorize Data

The following example demonstrates finding all of the employees from the employee table who do not have a pager:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL
WHERE PAGER IS NULL;
```

```
EMP_ID   LAST_NAM FIRST_NA PAGER
-----
311549902 STEPHENS TINA
442346889 PLEW      LINDA
220984332 WALLACE  MARIAH
443679012 SPURGEON TIFFANY
```

4 rows selected.

Understand that the literal word *null* is different than a NULL value. Examine the following example:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL
WHERE PAGER = NULL;
```

no rows selected.

### BETWEEN

The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value. The minimum and maximum values are included as part of the conditional set.

The following example shows that salary must fall between 20000 and 30000, including the values 20000 and 30000:

```
WHERE SALARY BETWEEN '20000' AND '30000'
```

The following example shows all of the products that cost between \$5.95 and \$14.50:

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST BETWEEN 5.95 AND 14.5;
```

```
PROD_ID   PROD_DESC                                COST
-----
222       PLASTIC PUMPKIN 18 INCH                  7.75
90        LIGHTED LANTERNS                          14.5
15        ASSORTED COSTUMES                         10
1234     KEY CHAIN                                 5.95
```

4 rows selected.

Notice that the values 5.95 and 14.5 are included in the output.

BETWEEN is inclusive and therefore includes the minimum and maximum values in the query results.

**By the  
Way**

## IN

The IN operator is used to compare a value to a list of literal values that have been specified. For TRUE to be returned, the compared value must match at least one of the values in the list.

The following example shows that salary must match one of the values 20000, 30000, or 40000:

```
WHERE SALARY IN('20000', '30000', '40000')
```

The following example show using the IN operator to match all of the products that have a product id within a certain range of values:

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID IN ('13', '9', '87', '119');
```

PROD_ID	PROD_DESC	COST
119	ASSORTED MASKS	4.95
87	PLASTIC SPIDERS	1.05
9	CANDY CORN	1.35
13	FALSE PARAFFIN TEETH	1.1

4 rows selected.

Using the IN operator can achieve the same results as using the OR operator and can return the results more quickly.

## LIKE

The LIKE operator is used to compare a value to similar values using wildcard operators. There are two wildcards used in conjunction with the LIKE operator:

- ▶ The percent sign (%)
- ▶ The underscore (\_)

The percent sign represents zero, one, or multiple characters. The underscore represents a single number or character. The symbols can be used in combinations.

To find any values that start with 200:

```
WHERE SALARY LIKE '200%
```

## HOOR 8: Using Operators to Categorize Data

To find any values that have 200 in any position:

```
WHERE SALARY LIKE '%200%'
```

To find any values that have 00 in the second and third positions:

```
WHERE SALARY LIKE '_00%'
```

To find any values that start with 2 and are at least three characters in length:

```
WHERE SALARY LIKE '2_%%'
```

To find any values that end with 2:

```
WHERE SALARY LIKE '%2'
```

To find any values that have a 2 in the second position and end with a 3:

```
WHERE SALARY LIKE '_2%3'
```

To find any values in a five-digit number that start with 2 and end with 3:

```
WHERE SALARY LIKE '2___3'
```

The following example shows all product descriptions that end with the letter S in uppercase:

```
SELECT PROD_DESC  
FROM PRODUCTS_TBL  
WHERE PROD_DESC LIKE '%S';
```

```
PROD_DESC  
-----  
LIGHTED LANTERNS  
ASSORTED COSTUMES  
PLASTIC SPIDERS  
ASSORTED MASKS
```

4 rows selected.

The following example shows all product descriptions whose second character is the letter S in uppercase:

```
SELECT PROD_DESC  
FROM PRODUCTS_TBL  
WHERE PROD_DESC LIKE '_S%';
```

```
PROD_DESC  
-----  
ASSORTED COSTUMES  
ASSORTED MASKS
```

2 rows selected.

## EXISTS

The EXISTS operator is used to search for the presence of a row in a specified table that meets certain criteria.

The following example searches to see whether the EMP\_ID 333333333 is in EMPLOYEE\_TBL:

```
WHERE EXISTS (SELECT EMP_ID FROM EMPLOYEE_TBL WHERE EMPLOYEE_ID = '333333333')
```

The following example is a form of a subquery, which is further discussed during Hour 14, “Using Subqueries to Define Unknown Data:”

```
SELECT COST
FROM PRODUCTS_TBL
WHERE EXISTS ( SELECT COST
                FROM PRODUCTS_TBL
                WHERE COST > 100 );
```

No rows selected.

-----

There were no rows selected because no records existed where the cost was greater than 100.

Consider the following example:

```
SELECT COST
FROM PRODUCTS_TBL
WHERE EXISTS ( SELECT COST
                FROM PRODUCTS_TBL
                WHERE COST < 100 );
```

COST

-----

```
29.99
 7.75
  1.1
14.5
  10
 1.35
 1.45
 1.05
 4.95
 5.95
59.99
```

11 rows selected.

The cost was displayed for records in the table because records existed where the product cost was less than 100.

## ALL, SOME, and ANY Operators

The ALL operator is used to compare a value to all values in another value set.

The following example tests salary to see whether it is greater than all salaries of the employees living in Indianapolis:

```
WHERE SALARY > ALL SALARY (SELECT FROM EMPLOYEE_TBL WHERE CITY = 'INDIANAPOLIS')
```

The following example shows how the ALL operator is used in conjunction with subquery:

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > ALL ( SELECT COST
                  FROM PRODUCTS_TBL
                  WHERE COST < 10 );
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
2345	OAK BOOKSHELF	59.99

4 rows selected.

In this output, five records had a cost greater than the cost of all records having a cost less than 10.

The ANY operator is used to compare a value to any applicable value in the list according to the condition. SOME is an alias for ANY, so they can be used interchangeably.

The following example tests salary to see whether it is greater than any of the salaries of employees living in Indianapolis:

```
WHERE SALARY > ANY (SELECT SALARY FROM EMPLOYEE_TBL WHERE CITY = 'INDIANAPOLIS')
```

The following example shows the use of the ANY operator used in conjunction with a subquery:

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > ANY ( SELECT COST
                  FROM PRODUCTS_TBL
                  WHERE COST < 10 );
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10

9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

10 rows selected.

In this output, more records were returned than when using ALL because the cost only had to be greater than any of the costs that were less than 10. The one record that was not displayed had a cost of 1.05, which was not greater than any of the values less than 10 (which was, in fact, 1.05). It should also be noted that ANY is not a synonym for IN because the IN operator can take an expression list of the form shown below, while ANY cannot:

```
IN (<Item#1>,<Item#2>,<Item#3>)
```

Additionally, the negation of IN, discussed in the section “Negative Operators,” would be NOT IN, and its alias would be <>ALL instead of <>ANY.

## Conjunctive Operators

What if you want to use multiple conditions to narrow data in a SQL statement? You must be able to combine the conditions, and you would do this with conjunctive operators. These operators are

- ▶ AND
- ▶ OR

*Conjunctive operators* provide a means to make multiple comparisons with different operators in the same SQL statement. The following sections describe each operator's behavior.

### AND

The AND operator allows the existence of multiple conditions in a SQL statement's WHERE clause. For an action to be taken by the SQL statement, whether it be a transaction or query, all conditions separated by the AND must be TRUE.

The following example shows that the EMPLOYEE\_ID must match 33333333 and the salary must equal 20000:

```
WHERE EMPLOYEE_ID = '33333333' AND SALARY = '20000'
```

## HOOR 8: Using Operators to Categorize Data

The following example shows the use of the AND operator to find the products with a cost between two limiting values:

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > 10
      AND COST < 30;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
90	LIGHTED LANTERNS	14.5

2 rows selected.

In this output, the value for cost had to be both greater than 10 and less than 30 for data to be retrieved.

This statement retrieves no data because each row of data has only one product identification:

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID = '7725'
      AND PROD_ID = '2345';
```

no rows selected

## OR

The OR operator is used to combine multiple conditions in a SQL statement's WHERE clause. For an action to be taken by the SQL statement, whether it is a transaction or query, at least one of the conditions that are separated by OR must be TRUE.

The following example shows that salary must match either 20000 or 30000:

```
WHERE SALARY = '20000' OR SALARY = '30000'
```

Each of the comparison and logical operators can be used singularly or in combination with each other.

The following example show the use of the OR operator to limit a query on the products table:

```
SELECT *
FROM PRODUCTS_TBL
WHERE PROD_ID = '90'
   OR PROD_ID = '2345';
```

PROD_ID	PROD_DESC	COST
2345	OAK BOOKSHELF	59.99
90	LIGHTED LANTERNS	14.5

2 rows selected.

In this output, either one of the conditions had to be TRUE for data to be retrieved. Two records that met either one or the other condition were found.

When using multiple conditions and operators in a SQL statement, you might find that it improves overall readability if parentheses are used to separate statements into logical groups. However, be aware that the misuse of parentheses could adversely affect your output results.

***Did you  
Know?***

In the next example, notice the use of the AND and two OR operators. In addition, notice the logical placement of the parentheses to make the statement more readable.

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > 10
   AND ( PROD_ID = '222'
       OR PROD_ID = '90'
       OR PROD_ID = '11235' );
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
90	LIGHTED LANTERNS	14.5

2 rows selected.

The cost in this output had to be greater than 10, and the product identification had to be any one of the three listed. A row was not returned for PROD\_ID 222 because the cost for this identification was not greater than 10. Parentheses are not used just to make your code more readable but also to ensure that logical grouping of conjunctive operators are evaluated properly. By default, operators are parsed from left to right in the order that they are listed. For example, you want to return all the

products in a table whose cost is greater than 5 and whose PRODUCT\_ID is in the range of values 222, 90, 11235, and 13. Try the following query to see the result set it returns:

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > 5
      AND (PROD_ID = '222'
          OR PROD_ID = '90'
          OR PROD_ID = '11235'
          OR PROD_ID = '13');
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
90	LIGHTED LANTERNS	14.50

3 rows in set

If you remove the parentheses, you can see how the result is much different:

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST > 5
      AND PROD_ID = '222'
      OR PROD_ID = '90'
      OR PROD_ID = '11235'
      OR PROD_ID = '13';
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
13	FALSE PARAFFIN TEETH	1.10
222	PLASTIC PUMPKIN 18 INCH	7.75
90	LIGHTED LANTERNS	14.50

3 rows in set

FALSE PARAFFIN TEETH gets returned now because this SQL query asks to return a PROD\_ID equal to 222 and COST greater than 5 or any rows with PROD\_ID equal to 90, 11235, or 13. Use parentheses properly within your WHERE clause to ensure that you are returning the correct logical result set.

## Negative Operators

Of all the conditions tested by the logical operators discussed here, there is a way to negate each one of these operators to change the condition's viewpoint.

The NOT operator reverses the meaning of the logical operator with which it is used. The NOT can be used with operators to form the following methods:

- ▶ Not Equal
- ▶ NOT BETWEEN
- ▶ NOT IN
- ▶ NOT LIKE
- ▶ IS NOT NULL
- ▶ NOT EXISTS
- ▶ NOT UNIQUE

Each method is discussed in the following sections. First, let's look at how to test for inequality.

## Not Equal

You have learned how to test for inequality using the `<>` operator. Inequality is worth mentioning in this section because to test for it, you are actually negating the equality operator. The following is a second method for testing inequality available in some SQL implementations:

The following examples show that salary is not equal to `20000`:

```
WHERE SALARY <> '20000'  
WHERE SALARY != '20000'
```

In the second example, you can see that the exclamation mark is used to negate the equality comparison. The use of the exclamation mark is allowed in addition to the standard operator for inequality `<>` in some implementations.

Check your particular implementation for the use of the exclamation mark to negate the inequality operator. The other operators mentioned are most always the same if compared between different SQL implementations.

**By the  
Way**

## NOT BETWEEN

The `BETWEEN` operator is negated as follows:

```
WHERE Salary NOT BETWEEN '20000' AND '30000'
```

**HOOR 8: Using Operators to Categorize Data**

The value for salary cannot fall between 20000 and 30000 or include the values 20000 and 30000. Let's see how this works on PRODUCTS\_TBL:

```
SELECT *
FROM PRODUCTS_TBL
WHERE COST NOT BETWEEN 5.95 AND 14.5;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
13	FALSE PARAFFIN TEETH	1.1
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
2345	OAK BOOKSHELF	59.99

7 rows selected.

**By the  
Way**

Remember that BETWEEN is inclusive; therefore, in the previous example, any rows that equal 5.95 or 14.50 are not included in the query results.

**NOT IN**

The IN operator is negated as NOT IN. All salaries in the following example that are not in the listed values, if any, are returned:

```
WHERE SALARY NOT IN ('20000', '30000', '40000')
```

The following example demonstrates using the negation of the IN operator:

```
SELECT *
FROM PRODUCTS_TBL
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
6	PUMPKIN CANDY	1.45
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

7 rows selected.

In this output, records were not displayed for the listed identifications after the NOT IN operator.

## NOT LIKE

The LIKE, or wildcard, operator is negated as NOT LIKE. When NOT LIKE is used, only values that are not similar are returned.

To find any values that do not start with 200:

```
WHERE SALARY NOT LIKE '200%'
```

To find any values that do not have 200 in any position:

```
WHERE SALARY NOT LIKE '%200%'
```

To find any values that do not have 00 starting in the second position:

```
WHERE SALARY NOT LIKE '_00%'
```

To find values that do not start with 2 and have a length of three or greater:

```
WHERE SALARY NOT LIKE '2_%_ %'
```

The following example demonstrates using the NOT LIKE operator to display a list of values:

```
SELECT PROD_DESC  
FROM PRODUCTS_TBL  
WHERE PROD_DESC NOT LIKE 'L%';
```

```
PROD_DESC  
-----  
WITCHES COSTUME  
PLASTIC PUMPKIN 18 INCH  
FALSE PARAFFIN TEETH  
ASSORTED COSTUMES  
CANDY CORN  
PUMPKIN CANDY  
PLASTIC SPIDERS  
ASSORTED MASKS  
KEY CHAIN  
OAK BOOKSHELF
```

10 rows selected.

In this output, the product descriptions starting with the letter *L* were not displayed.

## IS NOT NULL

The IS NULL operator is negated as IS NOT NULL to test for values that are not NULL. The following example only returns NOT NULL rows:

```
WHERE SALARY IS NOT NULL
```

## HOOR 8: Using Operators to Categorize Data

The following example demonstrates using the IS NOT NULL operator to retrieve a list of employees whose page number is not NULL.

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL
WHERE PAGER IS NOT NULL;
```

```
EMP_ID    LAST_NAM FIRST_NA PAGER
-----
213764555 GLASS     BRANDON 3175709980
313782439 GLASS     JACOB   8887345678
```

2 rows selected.

### NOT EXISTS

EXISTS is negated as NOT EXISTS.

The following example searches to see whether the EMP\_ID 3333333333 is not in EMPLOYEE\_TBL:

```
WHERE NOT EXISTS (SELECT EMP_ID FROM EMPLOYEE_TBL WHERE EMP_ID = '333333333')
```

The following example demonstrates the use of the NOT EXISTS operator in conjunction with a subquery:

```
SELECT MAX(COST)
FROM PRODUCTS_TBL
WHERE NOT EXISTS ( SELECT COST
                   FROM PRODUCTS_TBL
                   WHERE COST > 100 );
```

```
MAX(COST)
-----
      59.99
```

The maximum cost for the table is displayed in this output because no records contained a cost greater than 100.

## Arithmetic Operators

*Arithmetic operators* are used to perform mathematical functions in SQL—the same as in most other languages. The four conventional operators for mathematical functions are

- ▶ + (addition)
- ▶ - (subtraction)

- ▶ \* (multiplication)
- ▶ / (division)

## Addition

Addition is performed through the use of the plus (+) symbol.

The following example adds the SALARY column with the BONUS column for a total for each row of data:

```
SELECT SALARY + BONUS FROM EMPLOYEE_PAY_TBL;
```

This example returns all rows that are greater than the total of the SALARY and BONUS columns:

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY + BONUS > '40000';
```

## Subtraction

Subtraction is performed using the minus (-) symbol.

The following example subtracts the BONUS column from the SALARY column for the difference:

```
SELECT SALARY - BONUS FROM EMPLOYEE_PAY_TBL;
```

This example returns all rows where the SALARY minus the BONUS is greater than 40000:

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY - BONUS > '40000';
```

## Multiplication

Multiplication is performed by using the asterisk (\*) symbol.

The following example multiplies the SALARY column by 10:

```
SELECT SALARY * 10 FROM EMPLOYEE_PAY_TBL;
```

The next example returns all rows where the product of the SALARY multiplied by 10 is greater than 40000.

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY * 10 > '40000';
```

The pay rate in the following example is multiplied by 1.1, which increases the current pay rate by 10%:

```
SELECT EMP_ID, PAY_RATE, PAY_RATE * 1.1
FROM EMPLOYEE_PAY_TBL
WHERE PAY_RATE IS NOT NULL;
```

EMP_ID	PAY_RATE	PAY_RATE*1.1
442346889	14.75	16.225
220984332	11	12.1
443679012	15	16.5

3 rows selected.

## Division

Division is performed through the use of the slash (/) symbol.

The following example divides the SALARY column by 10:

```
SELECT SALARY / 10 FROM EMPLOYEE_PAY_TBL;
```

This example returns all rows that are greater than 40000:

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE SALARY > '40000';
```

This example returns all rows where the salary divided by 10 is greater than 40000:

```
SELECT SALARY FROM EMPLOYEE_PAY_TBL WHERE (SALARY / 10) > '40000';
```

## Arithmetic Operator Combinations

The arithmetic operators can be used in combinations with one another. Remember the rules of precedence in basic mathematics. Multiplication and division operations are performed first, and then addition and subtraction operations. The only way the user has control over the order of the mathematical operations is through the use of parentheses. Parentheses surrounding an expression cause that expression to be evaluated as a block.

*Precedence* is the order in which expressions are resolved in a mathematical expression or with embedded functions in SQL.

Expression	Result
1 + 1 * 5	6
(1 + 1) * 5	10
10 - 4 / 2 + 1	9
(10 - 4) / (2 + 1)	2

In the following examples, notice that the placement of parentheses in an expression does not affect the outcome if only multiplication and division are involved. Precedence is not a factor in these cases. Although it might not appear to make sense, it is possible that some implementations of SQL do not follow the ANSI standard in cases like this; however, this is unlikely.

Expression	Result
$4 * 6 / 2$	12
$(4 * 6) / 2$	12
$4 * (6 / 3)$	12

The following are some more examples:

```
SELECT SALARY * 10 + 1000
FROM EMPLOYEE_PAY_TBL
WHERE SALARY > 20000;
```

```
SELECT SALARY / 52 + BONUS
FROM EMPLOYEE_PAY_TBL;
```

```
SELECT (SALARY - 1000 + BONUS) / 52 * 1.1
FROM EMPLOYEE_PAY_TBL;
```

The following is a rather wild example:

```
SELECT SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY < BONUS * 3 + 10 / 2 - 50;
```

Because parentheses are not used, mathematical precedence takes effect, altering the value for BONUS tremendously for the condition.

When combining arithmetic operators, remember to consider the rules of precedence. The absence of parentheses in a statement could render inaccurate results. Although the syntax of a SQL statement is correct, a logical error might result.

**Watch  
Out!**

## Summary

You have been introduced to various operators available in SQL. You have learned the hows and whys of operators. You have seen examples of operators being used by themselves and in various combinations with one another, using the conjunctive-type operators AND and OR. You have learned the basic arithmetic functions: addition, subtraction, multiplication, and division. Comparison operators are used to test equality, inequality, less than values, and greater than values. Logical operators include BETWEEN, IN, LIKE, EXISTS, ANY, and ALL. You are already experiencing how elements are added to SQL statements to further specify conditions and better control the processing and retrieving capabilities provided with SQL.

## Q&A

**Q.** *Can I have more than one AND in the WHERE clause?*

**A.** Yes. In fact, all the operators can be used multiple times. An example would be

```
SELECT SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY > 20000
AND BONUS BETWEEN 1000 AND 3000
AND POSITION = 'VICE PRESIDENT'
```

**Q.** *What happens if I use single quotation marks around a NUMBER data type in a WHERE clause?*

**A.** Your query still processes. Quotation marks are not necessary for NUMBER fields.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. True or false: Both conditions when using the OR operator must be TRUE.
2. True or false: All specified values must match when using the IN operator.
3. True or false: The AND operator can be used in the SELECT and the WHERE clauses.
4. True or false: The ANY operator can accept an expression list.
5. What is the logical negation of the IN operator?
6. What is the logical negation of the ANY and ALL operators?
7. What, if anything, is wrong with the following SELECT statements?

**A.**

```
SELECT SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY BETWEEN 20000, 30000
```

**B.**

```
SELECT SALARY + DATE_HIRE
FROM EMPLOYEE_PAY_TBL
```

**C.**

```
SELECT SALARY, BONUS
FROM EMPLOYEE_PAY_TBL
WHERE DATE_HIRE BETWEEN 1999-09-22
AND 1999-11-23
AND POSITION = 'SALES'
OR POSITION = 'MARKETING'
AND EMPLOYEE_ID LIKE '%55%
```

## Exercises

1. Using the following CUSTOMER\_TBL:

```
DESCRIBE CUSTOMER_TBL;
```

Name	Null?	Type
CUST_ID	NOT NULL	VARCHAR (10)
CUST_NAME	NOT NULL	VARCHAR (30)
CUST_ADDRESS	NOT NULL	VARCHAR (20)
CUST_CITY	NOT NULL	VARCHAR (12)
CUST_STATE	NOT NULL	VARCHAR (2)
CUST_ZIP	NOT NULL	VARCHAR (5)
CUST_PHONE	NOT NULL	VARCHAR (10)
CUST_FAX		VARCHAR (10)

Write a `SELECT` statement that returns customer IDs and customer names (alpha order) for customers who live in Indiana, Ohio, Michigan, and Illinois, and whose names begin with the letters *A* or *B*.

2. Using the following `PRODUCTS_TBL`:

**DESCRIBE PRODUCTS\_TBL**

Name	Null?	Type
-----		
PROD_ID	NOT NULL	VARCHAR (10)
PROD_DESC	NOT NULL	VARCHAR (25)
COST	NOT NULL	DECIMAL (6,2)

Write a `SELECT` statement that returns the product ID, product description, and the product cost. Limit the product cost to range from \$1.00 and \$12.50.

3. Assuming that you used the `BETWEEN` operator in exercise 2, rewrite your SQL statement to achieve the same results using different operators. If you did not use the `BETWEEN` operator, do so now.
4. Write a `SELECT` statement that returns products that are either less than 1.00 or greater than 12.50. There are two ways to achieve the same results.
5. Write a `SELECT` statement that returns the following information from `PRODUCTS_TBL`: product description, product cost, and 5% sales tax for each product. List the products in order from most to least expensive.
6. Write a `SELECT` statement that returns the following information from `PRODUCTS_TBL`: product description, product cost, 5% sales tax for each product, and total cost with sales tax. List the products in order from most to least expensive. There are two ways to achieve the same results. Try both.

## HOUR 9

# Summarizing Data Results from a Query

In this hour, you learn about SQL's aggregate functions. You can perform a variety of useful functions with aggregate functions.

---

### ***The highlights of this hour include:***

- ▶ What functions are
- ▶ How functions are used
- ▶ When to use functions
- ▶ Using aggregate functions
- ▶ Summarizing data with aggregate functions
- ▶ Results from using functions

## What Are Aggregate Functions?

Functions are keywords in SQL used to manipulate values within columns for output purposes. A *function* is a command normally used in conjunction with a column name or expression that processes the incoming data to produce a result. SQL contains several types of functions. This hour covers aggregate functions. An *aggregate function* is used to provide summarization information for an SQL statement, such as counts, totals, and averages.

The basic set of aggregate functions discussed in this hour are

- ▶ COUNT
- ▶ SUM

## HOOR 9: Summarizing Data Results from a Query

- ▶ MAX
- ▶ MIN
- ▶ AVG

The following queries show the data used for most of this hour's examples:

```
SELECT *
FROM PRODUCTS_TBL;
```

PROD_ID	PROD_DESC	COST
11235	WITCHES COSTUME	29.99
222	PLASTIC PUMPKIN 18 INCH	7.75
13	FALSE PARAFFIN TEETH	1.1
90	LIGHTED LANTERNS	14.5
15	ASSORTED COSTUMES	10
9	CANDY CORN	1.35
6	PUMPKIN CANDY	1.45
87	PLASTIC SPIDERS	1.05
119	ASSORTED MASKS	4.95
1234	KEY CHAIN	5.95
2345	OAK BOOKSHELF	59.99

11 rows selected.

The following query lists the employee information from the EMPLOYEE\_TBL table. Note that some of the employees do not have pager numbers assigned.

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, PAGER
FROM EMPLOYEE_TBL;
```

EMP_ID	LAST_NAM	FIRST_NA	PAGER
311549902	STEPHENS	TINA	
442346889	PLEW	LINDA	
213764555	GLASS	BRANDON	3175709980
313782439	GLASS	JACOB	8887345678
220984332	WALLACE	MARIAH	
443679012	SPURGEON	TIFFANY	

6 rows selected.

## The COUNT Function

The COUNT function is used to count rows or values of a column that do not contain a NULL value. When used within a query, the COUNT function returns a numeric value. The COUNT function may also be used with the DISTINCT command to only count the distinct rows of a dataset. ALL (opposite of DISTINCT) is the default; it is not necessary to include ALL in the syntax. Duplicate rows are counted if DISTINCT

is not specified. One other option with the COUNT function is to use COUNT with an asterisk. COUNT(\*) counts all the rows of a table including duplicates, whether a NULL value is contained in a column or not.

The syntax for the COUNT function is as follows:

```
COUNT [ ( * ) | ( DISTINCT | ALL ) ] ( COLUMN NAME )
```

The DISTINCT command cannot be used with COUNT(\*), only with COUNT(*column\_name*).

**By the  
Way**

This example counts all employee IDs:

```
SELECT COUNT(EMPLOYEE_ID) FROM EMPLOYEE_PAY_ID
```

This example counts only the distinct rows:

```
SELECT COUNT(DISTINCT SALARY) FROM EMPLOYEE_PAY_TBL
```

This example counts all rows for SALARY:

```
SELECT COUNT(ALL SALARY) FROM EMPLOYEE_PAY_TBL
```

This final example counts all rows of the EMPLOYEE table:

```
SELECT COUNT(*) FROM EMPLOYEE_TBL
```

COUNT(\*) is used in the following example to get a count of all records in the EMPLOYEE\_TBL table. There are six employees.

```
SELECT COUNT(*)  
FROM EMPLOYEE_TBL;
```

```
COUNT(*)  
-----  
6
```

COUNT(EMP\_ID) is used in the next example to get a count of all the employee identification IDs that exist in the table. The returned count is the same as the last query because all employees have an identification number.

```
SELECT COUNT(EMP_ID)  
FROM EMPLOYEE_TBL;
```

```
COUNT(EMP_ID)  
-----  
6
```

## HOOR 9: Summarizing Data Results from a Query

COUNT (PAGER) is used in the following example to get a count of all of the employee records that have a pager number. Only two employees had pager numbers.

```
SELECT COUNT(PAGER)
FROM EMPLOYEE_TBL;
```

```
COUNT (PAGER)
-----
2
```

The ORDERS\_TBL table is shown next:

```
SELECT *
FROM ORDERS_TBL;
```

ORD_NUM	CUST_ID	PROD_ID	QTY	ORD_DATE_
56A901	232	11235	1	22-OCT-99
56A917	12	907	100	30-SEP-99
32A132	43	222	25	10-OCT-99
16C17	090	222	2	17-OCT-99
18D778	287	90	10	17-OCT-99
23E934	432	13	20	15-OCT-99
90C461	560	1234	2	

7 rows selected.

This example obtains a count of all distinct product identifications in the ORDERS\_TBL table.

```
SELECT COUNT(DISTINCT PROD_ID )
FROM ORDERS_TBL;
```

```
COUNT(DISTINCT PROD_ID )
-----
6
```

The PROD\_ID 222 has two entries in the table, thus reducing the distinct values from 7 to 6.

Because the COUNT function counts the rows, data types do not play a part. The rows can contain columns with any data type.

## The SUM Function

The SUM function is used to return a total on the values of a column for a group of rows. The SUM function can also be used in conjunction with DISTINCT. When SUM is used with DISTINCT, only the distinct rows are totaled, which might not have much purpose. Your total is not accurate in that case because rows of data are omitted.

The syntax for the SUM function is as follows:

```
SUM ([ DISTINCT ] COLUMN NAME)
```

The value of an argument must be numeric to use the SUM function. The SUM function cannot be used on columns having a data type other than numeric, such as character or date.

**By the  
Way**

This example totals the salaries:

```
SELECT SUM(SALARY) FROM EMPLOYEE_PAY_TBL
```

This example totals the distinct salaries:

```
SELECT SUM(DISTINCT SALARY) FROM EMPLOYEE_PAY_TBL
```

In the following query, the *sum*, or total amount, of all cost values is being retrieved from the PRODUCTS\_TBL table:

```
SELECT SUM(COST)
FROM PRODUCTS_TBL;
```

```
SUM(COST)
-----
163.07
```

Observe how the DISTINCT command in the following example skews the results from above, this is why it is rarely useful.

```
SELECT SUM(DISTINCT COST)
FROM PRODUCTS_TBL;
```

```
SUM(COST)
-----
72.14
```

The following query demonstrates that although some aggregate functions require numeric data, this is only limited to the type of data. Here the PAGER column of the EMPLOYEE\_TBL table is used to show that the implicit conversion of the CHAR data to a numeric type is supported:

```
SELECT SUM(PAGER)
FROM EMPLOYEE_TBL;
```

```
SUM(PAGER)
-----
12063055658
```

When you use a type of data that cannot be implicitly converted to a numeric type, such as the `LAST_NAME` column, it will return a result of 0.

```
SELECT SUM(LAST_NAME)
FROM EMPLOYEE_TBL;
```

```
SUM(LAST_NAME)
-----
0
```

## The AVG Function

The `AVG` function is used to find the average value for a given group of rows. When used with the `DISTINCT` command, the `AVG` function returns the average of the distinct rows. The syntax for the `AVG` function is as follows:

```
AVG ([ DISTINCT ] COLUMN NAME)
```

**By the  
Way**

The value of the argument must be numeric for the `AVG` function to work.

This example returns the average salary:

```
SELECT AVG(SALARY) FROM EMPLOYEE_PAY_TBL
```

This example returns the distinct average salary:

```
SELECT AVG(DISTINCT SALARY) EMPLOYEE_PAY_TBL
```

The average value for all values in the `PRODUCTS_TBL` table's `COST` column is being retrieved in the following example:

```
SELECT AVG(COST)
FROM PRODUCTS_TBL;
```

```
AVG(COST)
-----
13.5891667
```

**By the  
Way**

In some implementations, the results of your query might be truncated to the precision of the data type.

The next example uses two aggregate functions in the same query. Because some employees are paid hourly and others paid a salary, you want to retrieve the average value for both PAY\_RATE and SALARY.

```
SELECT AVG(PAY_RATE), AVG(SALARY)
FROM EMPLOYEE_PAY_TBL;
```

```
AVG(PAY_RATE)      AVG(SALARY)
-----
13.5833333        30000
```

## The MAX Function

The MAX function is used to return the maximum value from the values of a column in a group of rows. NULL values are ignored when using the MAX function. The DISTINCT command is an option. However, because the maximum value for all the rows is the same as the distinct maximum value, DISTINCT is useless.

The syntax for the MAX function is

```
MAX([ DISTINCT ] COLUMN NAME)
```

This example returns the highest salary:

```
SELECT MAX(SALARY) FROM EMPLOYEE_PAY_TBL
```

This example returns the highest distinct salary:

```
SELECT MAX(DISTINCT SALARY) FROM EMPLOYEE_PAY_TBL
```

The following example returns the maximum value for the COST column in the PRODUCTS\_TBL table:

```
SELECT MAX(COST)
FROM PRODUCTS_TBL;
```

```
MAX(COST)
-----29.99
```

```
SELECT MAX(DISTINCT COST)
FROM PRODUCTS_TBL;
```

```
MAX(COST)
29.99
```

## The MIN Function

The MIN function returns the minimum value of a column for a group of rows. NULL values are ignored when using the MIN function. The DISTINCT command is an

## HOOR 9: Summarizing Data Results from a Query

option. However, because the minimum value for all rows is the same as the minimum value for distinct rows, `DISTINCT` is useless.

The syntax for the `MIN` function is

```
MIN([ DISTINCT ] COLUMN NAME)
```

This example returns the lowest salary:

```
SELECT MIN(SALARY) FROM EMPLOYEE_PAY_TBL
```

This example returns the lowest distinct salary:

```
SELECT MIN(DISTINCT SALARY) FROM EMPLOYEE_PAY_TBL
```

The following example returns the minimum value for the `COST` column in the `PRODUCTS_TBL` table:

```
SELECT MIN(COST)
FROM PRODUCTS_TBL;
```

```
MIN(COST)
-----
      1.05
```

```
SELECT MIN(DISTINCT COST)
FROM PRODUCTS_TBL;
```

```
MIN(COST)
-----
      1.05
```

### Watch Out!

One very important thing to keep in mind when using aggregate functions with the `DISTINCT` command is that your query might not return the desired results. The purpose of aggregate functions is to return summarized data based on all rows of data in a table.

The final example combines aggregate functions with the use of arithmetic operators:

```
SELECT COUNT(ORD_NUM), SUM(QTY),
       SUM(QTY) / COUNT(ORD_NUM) AVG_QTY
FROM ORDERS_TBL;
```

```
COUNT(ORD_NUM)  SUM(QTY)  AVG_QTY
-----
      7          160      22.857143
```

You have performed a count on all order numbers, figured the sum of all quantities ordered, and, by dividing the two figures, have derived the average quantity of an item per order. You also created a column alias for the computation—`AVG_QTY`.

## Summary

Aggregate functions can be very useful and are quite simple to use. You have learned how to count values in columns, count rows of data in a table, get the maximum and minimum values for a column, figure the sum of the values in a column, and figure the average value for values in a column. Remember that NULL values are not considered when using aggregate functions, except when using the COUNT function in the format COUNT (\*).

Aggregate functions are the first functions in SQL that you have learned, but more follow. Aggregate functions can also be used for group values, which are discussed during the next hour. As you learn about other functions, you see that the syntaxes of most functions are similar to one another and that their concepts of use are relatively easy to understand.

## Q&A

**Q.** *Why are NULL values ignored when using the MAX or MIN function?*

**A.** A NULL value means that nothing is there.

**Q.** *Why don't data types matter when using the COUNT function?*

**A.** The COUNT function only counts rows.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, "Answers to Quizzes and Exercises," for answers.

## Quiz

1. True or false: The AVG function returns an average of all rows from a select column, including any NULL values.
2. True or false: The SUM function is used to add column totals.

**HOOR 9: Summarizing Data Results from a Query**

3. True or false: The COUNT(\*) function counts all rows in a table.
4. Will the following SELECT statements work? If not, what will fix the statements?

**A.**

```
SELECT COUNT *  
FROM EMPLOYEE_PAY_TBL;
```

**B.**

```
SELECT COUNT(EMPLOYEE_ID), SALARY  
FROM EMPLOYEE_PAY_TBL;
```

**C.**

```
SELECT MIN(BONUS), MAX(SALARY)  
FROM EMPLOYEE_PAY_TBL  
WHERE SALARY > 20000;
```

**D.**

```
SELECT COUNT(DISTINCT PROD_ID) FROM PRODUCTS_TBL;
```

**E.**

```
SELECT AVG(LAST_NAME) FROM EMPLOYEE_TBL;
```

**F.**

```
SELECT AVG(PAGER) FROM EMPLOYEE_TBL;
```

## Exercises

1. Use EMPLOYEE\_PAY\_TBL to construct SQL statements to solve the following exercises:
  - A. What is the average salary?
  - B. What is the maximum bonus?
  - C. What are the total salaries?
  - D. What is the minimum pay rate?
  - E. How many rows are in the table?
2. How many employees do we have whose last names begin with a G?
3. If every product cost \$10.00, what would be the total dollar amount for all orders?

## HOOR 10

# Sorting and Grouping Data

You have learned how to query the database and return data in an organized fashion. You have also learned how to sort data from a query. During this hour, you learn how to break returned data from a query into groups for improved readability.

---

### ***The highlights of this hour include:***

- ▶ Why you would want to group data
- ▶ The `GROUP BY` clause
- ▶ Group value functions
- ▶ The how and why of group functions
- ▶ Grouping by columns
- ▶ `GROUP BY` versus `ORDER BY`
- ▶ The `HAVING` clause

## Why Group Data?

Grouping data is the process of combining columns with duplicate values in a logical order. For example, a database might contain information about employees; many employees live in different cities, while some employees live in the same city. You might want to execute a query that shows employee information for each particular city. You are grouping employee information by city, and a summarized report is created.

Suppose that you wanted to figure the average salary paid to employees according to each city. You would do this by using the aggregate function `AVG` on the `SALARY` column, as you learned last hour, and by using the `GROUP BY` clause to group the output by city.

Grouping data is accomplished through the use of the `GROUP BY` clause of a `SELECT` statement (query). Last hour, you learned how to use aggregate functions. During this lesson,

you will see how aggregate functions are used in conjunction with the `GROUP BY` clause to display results more effectively.

## The `GROUP BY` Clause

The `GROUP BY` clause is used in collaboration with the `SELECT` statement to arrange identical data into groups. The `GROUP BY` clause follows the `WHERE` clause in a `SELECT` statement and precedes the `ORDER BY` clause.

The position of the `GROUP BY` clause in a query is as follows:

```
SELECT
FROM
WHERE
GROUP BY
ORDER BY
```

The `GROUP BY` clause must follow the conditions in the `WHERE` clause and must precede the `ORDER BY` clause if one is used.

The following is the `SELECT` statement's syntax, including the `GROUP BY` clause:

```
SELECT COLUMN1, COLUMN2
FROM TABLE1, TABLE2
WHERE CONDITIONS
GROUP BY COLUMN1, COLUMN2
ORDER BY COLUMN1, COLUMN2
```

The following sections give examples and explanations of the `GROUP BY` clause's use in a variety of situations.

## Group Functions

Typical group functions—those that are used with the `GROUP BY` clause to arrange data in groups—include `AVG`, `MAX`, `MIN`, `SUM`, and `COUNT`. These are the aggregate functions that you learned about during Hour 9, “Summarizing Data Results from a Query.” Remember that the aggregate functions were used for single values in Hour 9; now, you use the aggregate functions for group values.

## Grouping Selected Data

Grouping data is a simple process. The selected columns (the column list following the `SELECT` keyword in a query) are the columns that can be referenced in the `GROUP BY` clause. If a column is not found in the `SELECT` statement, it cannot be used in the `GROUP BY` clause. This is logical if you think about it—how can you group data on a report if the data is not displayed?

If the column name has been qualified, the qualified name must go into the GROUP BY clause. The column name can also be represented by a number, which is discussed later in Representing Column Names with Numbers. When grouping the data, the order of columns grouped does not have to match the column order in the SELECT clause.

## Creating Groups and Using Aggregate Functions

The SELECT clause has conditions that must be met when using GROUP BY. Specifically, whatever columns are selected must appear in the GROUP BY clause, except for any aggregate values. The columns in the GROUP BY clause do not necessarily have to be in the same order as they appear in the SELECT clause. Should the columns in the SELECT clause be qualified, the qualified names of the columns must be used in the GROUP BY clause. Some examples of syntax for the GROUP BY clause are shown next.

The following SQL statement selects the EMP\_ID and the CITY from the EMPLOYEE\_TBL and groups the data returned by CITY and then EMP\_ID:

```
SELECT EMP_ID, CITY
FROM EMPLOYEE_TBL
GROUP BY CITY, EMP_ID;
```

Note the order of the columns selected, versus the order of the columns in the GROUP BY clause.

**By the  
Way**

This SQL statement returns the EMP\_ID and the total of the SALARY column. Then it groups the results by both the salaries and employee IDs:

```
SELECT EMP_ID, SUM(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY SALARY, EMP_ID;
```

This SQL statement returns the total of all the salaries from the EMPLOYEE\_PAY\_TBL:

```
SELECT SUM(SALARY) AS TOTAL_SALARY
FROM EMPLOYEE_PAY_TBL;
```

```
TOTAL_SALARY
90000.00
```

```
1 row selected
```

## HOURL 10: Sorting and Grouping Data

This SQL statement returns the totals for the different groups of salaries:

```
SELECT SUM(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY SALARY;
```

```
SUM(SALARY)
(null)
20000.00
30000.00
40000.00
```

4 rows selected

Practical examples using real data follow. In this first example, you can see that there are three distinct cities in the EMPLOYEE\_TBL table:

```
SELECT CITY
FROM EMPLOYEE_TBL;
```

```
CITY
-----
GREENWOOD
INDIANAPOLIS
WHITELAND
INDIANAPOLIS
INDIANAPOLIS
INDIANAPOLIS
```

6 rows selected.

In the following example, you select the city and a count of all records for each city. You see a count on each of the three distinct cities because you are using a GROUP BY clause:

```
SELECT CITY, COUNT(*)
FROM EMPLOYEE_TBL
GROUP BY CITY;
```

```
CITY                COUNT(*)
-----
GREENWOOD           1
INDIANAPOLIS       4
WHITELAND           1
```

3 rows selected.

The following is a query from a temporary table created based on EMPLOYEE\_TBL and EMPLOYEE\_PAY\_TBL. You will soon learn how to join two tables for a query:

```
SELECT *
FROM EMP_PAY_TMP;
```

CITY	LAST_NAM	FIRST_NA	PAY_RATE	SALARY
GREENWOOD	STEPHENS	TINA		30000
INDIANAPOLIS	PLEW	LINDA	14.75	
WHITELAND	GLASS	BRANDON		40000
INDIANAPOLIS	GLASS	JACOB		20000
INDIANAPOLIS	WALLACE	MARIAH	11	
INDIANAPOLIS	SPURGEON	TIFFANY	15	

6 rows selected.

In the following example, you retrieve the average pay rate and salary on each distinct city using the aggregate function AVG. There is no average pay rate for GREENWOOD or WHITELAND because no employees living in those cities are paid hourly:

```
SELECT CITY, AVG(PAY_RATE), AVG(SALARY)
FROM EMP_PAY_TMP
GROUP BY CITY;
```

CITY	AVG(PAY_RATE)	AVG(SALARY)
GREENWOOD		30000
INDIANAPOLIS	13.5833333	20000
WHITELAND		40000

3 rows selected.

In the next example, you combine the use of multiple components in a query to return grouped data. You still want to see the average pay rate and salary, but only for INDIANAPOLIS and WHITELAND. You group the data by CITY—you have no choice because you are using aggregate functions on the other columns. Lastly, you want to order the report by 2 and then 3, which is the average pay rate and then average salary, respectively. Study the following details and output:

```
SELECT CITY, AVG(PAY_RATE), AVG(SALARY)
FROM EMP_PAY_TMP
WHERE CITY IN ('INDIANAPOLIS', 'WHITELAND')
GROUP BY CITY
ORDER BY 2,3;
```

CITY	AVG(PAY_RATE)	AVG(SALARY)
INDIANAPOLIS	13.5833333	20000
WHITELAND		40000

## HOOR 10: Sorting and Grouping Data

Values are sorted before NULL values; therefore, the record for INDIANAPOLIS is displayed first. GREENWOOD is not selected, but if it was, its record would have been displayed before the WHITELAND record because the average salary for GREENWOOD is \$30,000 (the second sort in the ORDER BY clause was on average salary).

The last example in this section shows the use of the MAX and MIN aggregate functions with the GROUP BY clause:

```
SELECT CITY, MAX(PAY_RATE), MIN(SALARY)
FROM EMP_PAY_TMP
GROUP BY CITY;
```

CITY	MAX(PAY_RATE)	MIN(SALARY)
-----	-----	-----
GREENWOOD		30000
INDIANAPOLIS	15	20000
WHITELAND		40000

3 rows selected.

## Representing Column Names with Numbers

Like the ORDER BY clause, the GROUP BY clause can be ordered by using an integer to represent the column name. The following is an example of representing column names with numbers:

```
SELECT YEAR (DATE_HIRE) as YEAR_HIRED, SUM(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY 1;
```

YEAR_HIRED	SUM(SALARY)
-----	-----
1989	40000.00
1990	
1991	
1994	30000.00
1996	
1997	20000.00

6 rows selected.

This SQL statement returns the SUM of the employee salaries grouped by the year in which the employees were hired. The GROUP BY clause is performed on the entire result set. The order for the groupings is 1, representing EMP\_ID.

## GROUP BY **Versus** ORDER BY

You should understand that the GROUP BY clause works the same as the ORDER BY clause in that both are used to sort data. The ORDER BY clause is specifically used to

sort data from a query. The GROUP BY clause also sorts data from a query to properly group the data. Therefore, the GROUP BY clause can be used to sort data the same as the ORDER BY clause.

There are some differences and disadvantages of using GROUP BY for sorting operations:

- ▶ All non-aggregate columns selected must be listed in the GROUP BY clause.
- ▶ The GROUP BY clause is generally not necessary unless using aggregate functions.

An example of performing sort operations utilizing the GROUP BY clause in place of the ORDER BY clause is shown next:

```
SELECT LAST_NAME, FIRST_NAME, CITY
FROM EMPLOYEE_TBL
GROUP BY LAST_NAME;
```

```
SELECT LAST_NAME, CITY
          *
```

```
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

In this example, an error was received from the database server stating that FIRST\_NAME is not a GROUP BY expression. Remember that all columns and expressions in the SELECT statement must be listed in the GROUP BY clause, with the exception of aggregate columns (those columns targeted by an aggregate function).

Different SQL implementations will return errors in different formats.

**By the  
Way**

In the next example, the previous problem is solved by adding all the expressions in the SELECT statement to the GROUP BY clause:

```
SELECT LAST_NAME, FIRST_NAME, CITY
FROM EMPLOYEE_TBL
GROUP BY LAST_NAME, FIRST_NAME, CITY;
```

LAST_NAME	FIRST_NAME	CITY
GLASS	BRANDON	WHITELAND
GLASS	JACOB	INDIANAPOLIS
PLEW	LINDA	INDIANAPOLIS
SPURGEON	TIFFANY	INDIANAPOLIS
STEPHENS	TINA	GREENWOOD
WALLACE	MARIAH	INDIANAPOLIS

6 rows selected.

## HOURL 10: Sorting and Grouping Data

In this example, the same columns were selected from the same table, but all columns in the `GROUP BY` clause are listed as they appeared after the `SELECT` keyword. The results were ordered by `LAST_NAME` first, `FIRST_NAME` second, and `CITY` third. These results could have been accomplished easier with the `ORDER BY` clause; however, it might help you better understand how the `GROUP BY` clause works if you can visualize how it must first sort data to group data results.

The following example shows a `SELECT` statement from `EMPLOYEE_TBL` and uses the `GROUP BY` clause to order by `CITY`:

```
SELECT CITY, LAST_NAME
FROM EMPLOYEE_TBL
GROUP BY CITY, LAST_NAME;
```

CITY	LAST_NAME
GREENWOOD	STEPHENS
INDIANAPOLIS	GLASS
INDIANAPOLIS	PLEW
INDIANAPOLIS	SPURGEON
INDIANAPOLIS	WALLACE
WHITELAND	GLASS

6 rows selected.

Notice the order of data in the previous results, as well as the `LAST_NAME` of the individual for each `CITY`. In the following example, all employee records in the `EMPLOYEE_TBL` table are now counted, and the results are grouped by `CITY`, but ordered by the count on each city first:

```
SELECT CITY, COUNT(*)
FROM EMPLOYEE_TBL
GROUP BY CITY
ORDER BY 2,1;
```

CITY	COUNT(*)
GREENWOOD	1
WHITELAND	1
INDIANAPOLIS	4

Notice the order of the results. The results were first sorted by the count on each city (1–4), and then by city. The count for the first two cities in the output is 1. Because the count is the same, which is the first expression in the `ORDER BY` clause, the city is then sorted; `GREENWOOD` is placed before `WHITELAND`.

Although `GROUP BY` and `ORDER BY` perform a similar function, there is one major difference. The `GROUP BY` clause is designed to group identical data, whereas the `ORDER BY` clause is designed merely to put data into a specific order. `GROUP BY` and `ORDER BY` can be used in the same `SELECT` statement, but must follow a specific

order. The GROUP BY clause is always placed before the ORDER BY clause in the SELECT statement.

The GROUP BY clause can be used in the CREATE VIEW statement to sort data, but the ORDER BY clause is not allowed in the CREATE VIEW statement. The CREATE VIEW statement is discussed in depth in Hour 20, “Creating and Using Views and Synonyms.”

***Did you  
Know?***

## The HAVING Clause

The HAVING clause, when used in conjunction with the GROUP BY clause in a SELECT statement, tells GROUP BY which groups to include in the output. HAVING is to GROUP BY as WHERE is to SELECT. In other words, the WHERE clause places conditions on the selected columns, and the HAVING clause places conditions on groups created by the GROUP BY clause. Therefore, when you use the HAVING clause, you are effectively including or excluding, as the case might be, whole groups of data from the query results.

The following is the position of the HAVING clause in a query:

```
SELECT
FROM
WHERE
GROUP BY
HAVING
ORDER BY
```

The HAVING clause must follow the GROUP BY clause in a query and must also precede the ORDER BY clause if used.

The following is the syntax of the SELECT statement, including the HAVING clause:

```
SELECT COLUMN1, COLUMN2
FROM TABLE1, TABLE2
WHERE CONDITIONS
GROUP BY COLUMN1, COLUMN2
HAVING CONDITIONS
ORDER BY COLUMN1, COLUMN2
```

In the following example, you select the average pay rate and salary for all cities except GREENWOOD. You group the output by CITY, but only want to display those

## HOOR 10: Sorting and Grouping Data

groups (cities) that have an average salary greater than \$20,000. You sort the results by average salary for each city:

```
SELECT CITY, AVG(PAY_RATE), AVG(SALARY)
FROM EMP_PAY_TMP
WHERE CITY <> 'GREENWOOD'
GROUP BY CITY
HAVING AVG(SALARY) > 20000
ORDER BY 3;
```

CITY	AVG(PAY_RATE)	AVG(SALARY)
-----	-----	-----
WHITELAND		40000

1 row selected.

Why was only one row returned by this query?

- ▶ The city GREENWOOD was eliminated from the WHERE clause.
- ▶ INDIANAPOLIS was deducted from the output because the average salary was 20000, which is not greater than 20000.

## Summary

You have learned how to group the results of a query using the GROUP BY clause. The GROUP BY clause is primarily used with aggregate SQL functions, such as SUM, AVG, MAX, MIN, and COUNT. The nature of GROUP BY is like that of ORDER BY in that both sort query results. The GROUP BY clause must sort data to group results logically, but can also be used exclusively to sort data, although an ORDER BY clause is much simpler for this purpose.

The HAVING clause, an extension to the GROUP BY clause, is used to place conditions on the established groups of a query. The WHERE clause is used to place conditions on a query's SELECT clause. During the next hour, you learn a new arsenal of functions that allow you to further manipulate query results.

## Q&A

- Q.** *Is using the GROUP BY clause mandatory when using the ORDER BY clause in a SELECT statement?*
- A.** No. Using the GROUP BY clause is strictly optional, but it can be very useful when used with ORDER BY.

**Q. What is a group value?**

**A.** Take the CITY column from the EMPLOYEE\_TBL. If you select the employee's name and city, and then group the output by city, all the cities that are identical are arranged together.

**Q. Must a column appear in the SELECT statement to use a GROUP BY clause on it?**

**A.** Yes, a column must be in the SELECT statement to use a GROUP BY clause on it.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, "Answers to Quizzes and Exercises," for answers.

## Quiz

**1.** Will the following SQL statements work?

**A.**

```
SELECT SUM(SALARY), EMP_ID
FROM EMPLOYEE_PAY_TBL
GROUP BY 1 and 2;
```

**B.**

```
SELECT EMP_ID, MAX(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY SALARY, EMP_ID;
```

**C.**

```
SELECT EMP_ID, COUNT(SALARY)
FROM EMPLOYEE_PAY_TBL
ORDER BY EMP_ID
GROUP BY SALARY;
```

**D.**

```
SELECT YEAR(DATE_HIRE) AS YEAR_HIRED, SUM(SALARY)
FROM EMPLOYEE_PAY_TBL
GROUP BY 1
HAVING SUM(SALARY)>20000;
```

2. True or false: You must also use the `GROUP BY` clause when using the `HAVING` clause.
3. True or false: The following SQL statement returns a total of the salaries by groups:  

```
SELECT SUM(SALARY)
FROM EMPLOYEE_PAY_TBL;
```
4. True or false: The columns selected must appear in the `GROUP BY` clause in the same order.
5. True or false: The `HAVING` clause tells the `GROUP BY` which groups to include.

## Exercises

1. Invoke `mysql.exe` on your computer, and then type `use learnsql;` at the `mysql>` prompt.
2. Enter the following query at the `mysql>` prompt to show all cities in `EMPLOYEE_TBL`:  

```
SELECT CITY
FROM EMPLOYEE_TBL;
```
3. Now, enter the following query and compare the results to the query in exercise 2:  

```
SELECT CITY, COUNT(*)
FROM EMPLOYEE_TBL
GROUP BY CITY;
```
4. The `HAVING` clause works like the `WHERE` clause in that it allows the user to specify conditions on data returned. The `WHERE` clause is the main filter on the query and the `HAVING` clause is the filter used after groups of data have been established using the `GROUP BY` clause. Enter the following query to see how the `HAVING` clause works:  

```
SELECT CITY, COUNT(*)
FROM EMPLOYEE_TBL
GROUP BY CITY
HAVING COUNT(*) > 1;
```
5. Modify the query in exercise 3 to order the results in descending order, from highest count to lowest.

- 6.** Write a query to list the average pay rate by position from the EMPLOYEE\_PAY\_TBL table.
- 7.** Write a query to list the average salary by position from the EMPLOYEE\_PAY\_TBL table.
- 8.** Write a query to list the average salary by position from the EMPLOYEE\_PAY\_TBL table where the average salary is greater than 20000.

*This page intentionally left blank*

## HOUR 11

# Restructuring the Appearance of Data

During this hour, you learn how to restructure the appearance of output results using a wide array of functions, some ANSI standard functions, and other functions based on the standard and several variations used by some major SQL implementations.

---

### ***This hour's highlights include:***

- ▶ Introduction to character functions
- ▶ How and when to use character functions
- ▶ Examples of ANSI SQL functions
- ▶ Examples of common implementation-specific functions
- ▶ Overview of conversion functions
- ▶ How and when to use conversion functions

## **ANSI Character Functions**

*Character functions* are functions used to represent strings in SQL in formats alternate to how they are stored in the table. The first part of this hour discusses the concepts for character functions as covered by ANSI. The second part of this hour shows real-world examples using functions that are specific to various SQL implementations. ANSI functions discussed in this hour include concatenation, substring, TRANSLATE, REPLACE, UPPER, and LOWER.

## Concatenation

*Concatenation* is the process of combining two separate strings into one string. For example, you might want to concatenate an individual's first and last names into a single string for the complete name.

JOHN concatenated with SMITH produces JOHN SMITH.

## Substring

The concept of *substring* is the capability to extract part of a string, or a “sub” of the string. For example, the following values are substrings of JOHNSON:

- ▶ J
- ▶ JOHN
- ▶ JO
- ▶ ON
- ▶ SON

## TRANSLATE

The TRANSLATE function is used to translate a string, character by character, into another string. There are normally three arguments with the TRANSLATE function: the string to be converted, a list of the characters to convert, and a list of the substitution characters. Implementation examples are shown in the next part of this hour.

## Various Common Character Functions

Character functions are used mainly to compare, join, search, and extract a segment of a string or a value in a column. Several character functions are available to the SQL programmer.

The following sections illustrate the application of ANSI concepts in some of the leading implementations of SQL, such as Oracle, Sybase, SQLBase, Informix, and SQL Server.

The ANSI concepts discussed in this book are just that—concepts. Standards provided by ANSI are simply guidelines for how the use of SQL in a relational database should be implemented. With that thought, keep in mind that the specific functions discussed in this hour are not necessarily the exact functions that you might use in your particular implementation. Yes, the concepts are the same, and the way the functions work are generally the same, but function names and actual syntax might differ.

## Concatenation

Concatenation, along with most other functions, is represented slightly differently among various implementations. The following examples show the use of concatenation in Oracle and SQL Server.

Let's say you want to concatenate JOHN and SON to produce JOHNSON. In Oracle, your code would look like this:

```
SELECT 'JOHN' || 'SON'
```

In SQL Server, your code would appear as follows:

```
SELECT 'JOHN' + 'SON'
```

In MySQL, your code would look like

```
SELECT CONCAT('JOHN' , 'SON')
```

Now for an overview of the syntaxes. The syntax for Oracle is

```
COLUMN_NAME || [ ' ' || ] COLUMN_NAME [ COLUMN_NAME ]
```

The syntax for SQL Server is

```
COLUMN_NAME + [ ' ' + ] COLUMN_NAME [ COLUMN_NAME ]
```

The syntax for MySQL is

```
CONCAT(COLUMN_NAME , [ ' ' , ] COLUMN_NAME [ COLUMN_NAME ])
```

This SQL Server statement concatenates the values for city and state into one value:

```
SELECT CITY + STATE FROM EMPLOYEE_TBL;
```

This Oracle statement concatenates the values for city and state into one value, placing a comma between the values for city and state:

```
SELECT CITY || ', ' || STATE FROM EMPLOYEE_TBL;
```

This SQL Server statement concatenates the values for city and state into one value, placing a space between the two original values.

```
SELECT CITY + ' ' + STATE FROM EMPLOYEE_TBL;
```

This SQL Server statement concatenates the last name with the first name and inserts a comma between the two original values.

```
SELECT LAST_NAME || ', ' || FIRST_NAME NAME
FROM EMPLOYEE_TBL;
```

```
NAME
-----
STEPHENS, TINA
PLEW, LINDA
GLASS, BRANDON
GLASS, JACOB
WALLACE, MARIAH
SPURGEON, TIFFANY
```

6 rows selected.

**By the  
Way**

Notice the use of single quotation marks and a comma in the preceding SQL statement. Most characters and symbols are allowed if enclosed by single quotation marks. Some implementations might use double quotation marks for literal string values.

## TRANSLATE

The TRANSLATE function searches a string of characters and checks for a specific character, makes note of the position found, searches the replacement string at the same position, and then replaces that character with the new value. The syntax is

```
TRANSLATE(CHARACTER SET, VALUE1, VALUE2)
```

This SQL statement substitutes every occurrence of I in the string with A, every occurrence of N with B, and replaces all occurrences of D with C.

```
SELECT TRANSLATE (CITY, 'IND', 'ABC' FROM EMPLOYEE_TBL) CITY_TRANSLATION
```

The following example illustrates the use of TRANSLATE with real data:

```
SELECT CITY, TRANSLATE(CITY, 'IND', 'ABC')
FROM EMPLOYEE_TBL;
```

```
CITY          CITY_TRANSLATION
-----
GREENWOOD     GREEBWOOC
INDIANAPOLIS ABCAABAPOLAS
WHITELAND     WHATELABC
INDIANAPOLIS ABCAABAPOLAS
INDIANAPOLIS ABCAABAPOLAS
INDIANAPOLIS ABCAABAPOLAS
```

6 rows selected.

Notice in this example that all occurrences of I were replaced with A, N with B, and D with C. In the city INDIANAPOLIS, IND was replaced with ABC, but in GREENWOOD, D was replaced with C. Also notice how the value WHITELAND was translated.

## REPLACE

The REPLACE function is used to replace every occurrence of a character(s) with a specified character(s). The use of this function is similar to the TRANSLATE function, except only one specific character or string is replaced within another string. The syntax is

```
REPLACE('VALUE', 'VALUE', [ NULL ] 'VALUE')
```

This statement returns all the first names and changes any occurrence of T to a B:

```
SELECT REPLACE(FIRST__'T', 'B') FROM EMPLOYEE_TBL
```

This statement returns all of the cities in the employee table and the same cities with each I replaced with a Z:

```
SELECT CITY, REPLACE(CITY, 'I', 'Z')
FROM EMPLOYEE_TBL;
```

CITY	REPLACE(CITY)
GREENWOOD	GREENWOOD
INDIANAPOLIS	ZNDZANAPOLZS
WHITELAND	WHZTELAND
INDIANAPOLIS	ZNDZANAPOLZS
INDIANAPOLIS	ZNDZANAPOLZS
INDIANAPOLIS	ZNDZANAPOLZS

6 rows selected.

## UPPER

Most implementations have a way to control the case of data by using functions. The UPPER function is used to convert lowercase letters to uppercase letters for a specific string.

The syntax is as follows:

```
UPPER(character string)
```

This SQL statement converts all characters in the column to uppercase:

```
SELECT UPPER(CITY)
FROM EMPLOYEE_TBL;
```

```
UPPER(CITY)
-----
GREENWOOD
INDIANAPOLIS
WHITELAND
INDIANAPOLIS
INDIANAPOLIS
INDIANAPOLIS
```

6 rows selected.

## LOWER

The converse of the UPPER function, the LOWER function is used to convert uppercase letters to lowercase letters for a specific string.

The syntax is as follows:

```
LOWER(character string)
```

This SQL statement converts all characters in the column to lowercase:

```
SELECT LOWER(CITY)
FROM EMPLOYEE_TBL;
```

```
LOWER(CITY)
-----
greenwood
indianapolis
whiteLand
indianapolis
indianapolis
indianapolis
```

6 rows selected.

## SUBSTR

Taking an expression's substring is common in most implementations of SQL, but the function name might differ, as shown in the following Oracle and SQL Server examples.

The syntax for Oracle is

```
SUBSTR(COLUMN NAME, STARTING POSITION, LENGTH)
```

The syntax for SQL Server is

```
SUBSTRING(COLUMN NAME, STARTING POSITION, LENGTH)
```

The only difference between the two implementations is the spelling of the function name.

This SQL statement returns the first three characters of EMP\_ID:

```
SELECT SUBSTRING(EMP_ID,1,3) FROM EMPLOYEE_TBL
```

This SQL statement returns the fourth and fifth characters of EMP\_ID:

```
SELECT SUBSTRING(EMP_ID,4,2) FROM EMPLOYEE_TBL
```

This SQL statement returns the sixth through the ninth characters of EMP\_ID:

```
SELECT SUBSTRING(EMP_ID,6,4) FROM EMPLOYEE_TBL
```

The following is an example that is compatible with Microsoft SQL Server and MySQL:

```
SELECT EMP_ID, SUBSTRING(EMP_ID,1,3)  
FROM EMPLOYEE_TBL;
```

```
EMP_ID    SUB  
-----  
311549902 311  
442346889 442  
213764555 213  
313782439 313  
220984332 220  
443679012 443
```

6 rows affected.

The following SQL statement is what you would use for Oracle.

```
SELECT EMP_ID, SUBSTR(EMP_ID,1,3)  
FROM EMPLOYEE_TBL;
```

```
EMP_ID    SUB  
-----  
311549902 311  
442346889 442  
213764555 213  
313782439 313  
220984332 220  
443679012 443
```

6 rows selected.

Notice the difference between the feedback of the two queries. The first example returns the feedback 6 rows affected and the second returns 6 rows selected. You will see differences such as this between the various implementations.

### INSTR

The INSTR function is used to search a string of characters for a specific set of characters and report the position of those characters. The syntax is as follows:

```
INSTR(COLUMN_NAME, 'SET',
[ START POSITION [ , OCCURRENCE ] ] );
```

This SQL statement returns the position of the first occurrence of the letter *I* for each state in EMPLOYEE\_TBL:

```
SELECT INSTR(STATE, 'I', 1, 1) FROM EMPLOYEE_TBL;
```

This SQL statement looks for the first occurrence of the letter *A* in the PROD\_DESC column:

```
SELECT PROD_DESC,
       INSTR(PROD_DESC, 'A', 1, 1)
FROM PRODUCTS_TBL;
```

PROD_DESC	INSTR (PROD_DESC, 'A', 1, 1)
WITCHES COSTUME	0
PLASTIC PUMPKIN 18 INCH	3
FALSE PARAFFIN TEETH	2
LIGHTED LANTERNS	10
ASSORTED COSTUMES	1
CANDY CORN	2
PUMPKIN CANDY	10
PLASTIC SPIDERS	3
ASSORTED MASKS	1
KEY CHAIN	7
OAK BOOKSHELF	2

11 rows selected.

Notice that if the searched character *A* was not found in a string, the value 0 was returned for the position.

## LTRIM

The LTRIM function is another way of clipping part of a string. This function and SUBSTRING are in the same family. LTRIM is used to trim characters from the left of a string. The syntax is

```
LTRIM(CHARACTER STRING [ , 'set' ])
```

This SQL statement trims the characters LES from the left of all names that are LESLIE.

```
SELECT LTRIM(FIRST_NAME, 'LES') FROM CUSTOMER_TBL WHERE FIRST_NAME = 'LESLIE';
```

This SQL statement returns the positions and also the returns the position with the word 'SALES' trimmed from the left side of the character string:

```
SELECT POSITION, LTRIM(POSITION, 'SALES')  
FROM EMPLOYEE_PAY_TBL;
```

POSITION	LTRIM(POSITION,
-----	-----
MARKETING	MARKETING
TEAM LEADER	TEAM LEADER
SALES MANAGER	MANAGER
SALESMAN	MAN
SHIPPER	HIPPER
SHIPPER	HIPPER

6 rows selected.

The S in SHIPPER was trimmed off, even though SHIPPER does not contain the string SALES. The first four characters of SALES were ignored. The searched characters must appear in the same order of the search string and must be on the far left of the string. In other words, LTRIM will trim off all characters to the left of the last occurrence in the search string.

## RTRIM

Like LTRIM, the RTRIM function is used to trim characters, but this time from the right of a string. The syntax is

```
RTRIM(CHARACTER STRING [ , 'set' ])
```

This SQL statement returns the first name BRANDON and trims the ON, leaving BRAND as a result:

```
SELECT RTRIM(FIRST_NAME, 'ON') FROM EMPLOYEE_TBL WHERE FIRST_NAME = 'BRANDON';
```

## HOOR 11: Restructuring the Appearance of Data

This SQL statement returns a list of the positions in the PAY\_TBL as well as the positions with the letters 'ER' trimmed from the right of the character string:

```
SELECT POSITION, RTRIM(POSITION, 'ER')
FROM EMPLOYEE_PAY_TBL;
```

POSITION	RTRIM(POSITION,
MARKETING	MARKETING
TEAM LEADER	TEAM LEAD
SALES MANAGER	SALES MANAG
SALESMAN	SALESMAN
SHIPPER	SHIPP
SHIPPER	SHIPP

6 rows selected.

The string ER was trimmed from the right of all applicable strings.

## DECODE

The DECODE function is not ANSI—at least not at the time of this writing—but its use is shown here because of its great power. This function is used in SQLBase, Oracle, and possibly other implementations. DECODE is used to search a string for a value or string, and if the string is found, an alternative string is displayed as part of the query results.

The syntax is

```
DECODE(COLUMN NAME, 'SEARCH1', 'RETURN1', [ 'SEARCH2', 'RETURN2', 'DEFAULT
VALUE' ])
```

This query searches the value of all last names in EMPLOYEE\_TBL; if the value SMITH is found, JONES is displayed in its place. Any other names are displayed as OTHER, which is called the default value.

```
SELECT DECODE(LAST_NAME, 'SMITH', 'JONES', 'OTHER') FROM EMPLOYEE_TBL;
```

In the following example, DECODE is used on the values for CITY in EMPLOYEE\_TBL:

```
SELECT CITY,
      DECODE(CITY, 'INDIANAPOLIS', 'INDY',
              'GREENWOOD', 'GREEN', 'OTHER')
FROM EMPLOYEE_TBL;
```

CITY	DECOD
GREENWOOD	GREEN
INDIANAPOLIS	INDY
WHITELAND	OTHER
INDIANAPOLIS	INDY
INDIANAPOLIS	INDY
INDIANAPOLIS	INDY

6 rows selected.

The output shows the value INDIANAPOLIS displayed as INDY, GREENWOOD displayed as GREEN, and all other cities displayed as OTHER.

## Miscellaneous Character Functions

The following sections show a few other character functions worth mentioning. Once again, these are functions that are fairly common among major implementations.

### LENGTH

The LENGTH function is a common function used to find the length of a string, number, date, or expression in bytes. The syntax is

```
LENGTH(CHARACTER STRING)
```

This SQL statement returns the product description and also its corresponding length:

```
SELECT PROD_DESC, LENGTH(PROD_DESC)
FROM PRODUCTS_TBL;
```

PROD_DESC	LENGTH(PROD_DESC)
-----	-----
WITCHES COSTUME	15
PLASTIC PUMPKIN 18 INCH	23
FALSE PARAFFIN TEETH	19
LIGHTED LANTERNS	16
ASSORTED COSTUMES	17
CANDY CORN	10
PUMPKIN CANDY	13
PLASTIC SPIDERS	15
ASSORTED MASKS	14
KEY CHAIN	9
OAK BOOKSHELF	13

11 rows selected.

### IFNULL (NULL Value Checker)

The IFNULL function is used to return data from one expression if another expression is NULL. IFNULL can be used with most data types; however, the value and the substitute must be the same data type. The syntax is

```
IFNULL('VALUE', 'SUBSTITUTION')
```

## HOOR 11: Restructuring the Appearance of Data

This SQL statement finds NULL values and substitutes 999999999 for any NULL values:

```
SELECT PAGER, IFNULL(PAGER,999999999)
FROM EMPLOYEE_TBL;
```

PAGER	IFNULL (PAGER,
-----	-----
	9999999999
	9999999999
3175709980	3175709980
8887345678	8887345678
	9999999999
	9999999999

6 rows selected.

Only NULL values were represented as 9999999999.

## COALESCE

The COALESCE function is similar to the IFNULL function in that it is used to specifically replace NULL values within the result set. The COALESCE function, however, can accept a whole set of values and checks each one in order until it finds a non-NULL result. If a non-NULL result is not present, COALESCE returns a NULL value.

The following example demonstrates the COALESCE function by giving us the first non-NULL value of BONUS, SALARY, and PAY\_RATE:

```
SELECT EMP_ID, COALESCE(BONUS,SALARY,PAY_RATE)
FROM EMPLOYEE_PAY_TBL;
```

EMP_ID	COALESCE (BONUS,SALARY,PAY_RATE)
-----	-----
213764555	2000.00
220984332	11.00
311549902	40000.00
313782439	1000.00
442346889	14.75
443679012	15.00

6 rows selected.

## LPAD

LPAD (left pad) is used to add characters or spaces to the left of a string. The syntax is

```
LPAD (CHARACTER SET)
```

The following example pads periods to the left of each product description, totaling 30 characters between the actual value and padded periods:

```
SELECT LPAD(PROD_DESC,30, '.') PRODUCT
FROM PRODUCTS_TBL;
```

```
PRODUCT
-----
.....WITCHES COSTUME
.....PLASTIC PUMPKIN 18 INCH
.....FALSE PARAFFIN TEETH
.....LIGHTED LANTERNS
.....ASSORTED COSTUMES
.....CANDY CORN
.....PUMPKIN CANDY
.....PLASTIC SPIDERS
.....ASSORTED MASKS
.....KEY CHAIN
.....OAK BOOKSHELF
```

11 rows selected.

## RPAD

The RPAD (right pad) is used to add characters or spaces to the right of a string. The syntax is

```
RPAD(CHARACTER SET)
```

The following example pads periods to the right of each product description, totaling 30 characters between the actual value and padded periods:

```
SELECT RPAD(PROD_DESC,30, '.') PRODUCT
FROM PRODUCTS_TBL;
```

```
PRODUCT
-----
WITCHES COSTUME.....
PLASTIC PUMPKIN 18 INCH.....
FALSE PARAFFIN TEETH.....
LIGHTED LANTERNS.....
ASSORTED COSTUMES.....
CANDY CORN.....
PUMPKIN CANDY.....
PLASTIC SPIDERS.....
ASSORTED MASKS.....
KEY CHAIN.....
OAK BOOKSHELF.....
```

11 rows selected.

## ASCII

The ASCII function is used to return the American Standard Code for Information Interchange (ASCII) representation of the leftmost character of a string. The syntax is

ASCII(*CHARACTER SET*)

The following are some examples:

- ▶ ASCII('A') returns 65
- ▶ ASCII('B') returns 66
- ▶ ASCII('C') returns 67
- ▶ ASCII('a') returns 97

For more information, you may refer to the ASCII chart located at [www.asciitable.com](http://www.asciitable.com).

## Mathematical Functions

Mathematical functions are fairly standard across implementations. *Mathematical functions* allow you to manipulate numeric values in a database according to mathematical rules.

The most common functions include the following:

Absolute value	(ABS)
Rounding	(ROUND)
Square root	(SQRT)
Sign values	(SIGN)
Power	(POWER)
Ceiling and floor values	(CEIL, FLOOR)
Exponential values	(EXP)
SIN, COS, TAN	

The general syntax of most mathematical functions is

FUNCTION(*EXPRESSION*)

## Conversion Functions

*Conversion functions* are used to convert a data type into another data type. For example, there might be times when you want to convert character data into numeric data. You might have data that is normally stored in character format, but occasionally you want to convert the character format to numeric for the purpose of making calculations. Mathematical functions and computations are not allowed on data that is represented in character format.

The following are general types of data conversions:

- ▶ Character to numeric
- ▶ Numeric to character
- ▶ Character to date
- ▶ Date to character

The first two types of conversions are discussed in this hour. The remaining conversion types are discussed during Hour 12, “Understanding Dates and Times,” where date and time storage is discussed in more detail.

Some implementations might implicitly convert data types when necessary. This means that the system will make the conversion for you when changing between data types. In these cases, the use of conversion functions is unnecessary. Please check your implementations documentation to see which types of implicit conversions are supported.

**By the  
Way**

### Converting Character Strings to Numbers

You should notice two things regarding the differences between numeric data types and character string data types:

- ▶ Arithmetic expressions and functions can be used on numeric values.
- ▶ Numeric values are right-justified, whereas character string data types are left-justified in the output results.

When a character string is converted to a numeric value, the value takes on the two attributes just mentioned.

Some implementations might not have functions to convert character strings to numbers, whereas some will have such conversion functions. In either case, consult your implementation documentation for specific syntax and rules for conversions.

### **By the Way**

For a character string to be converted to a number, the characters must typically be 0 through 9. The addition symbol, minus symbol, and period can also be used to represent positive numbers, negative numbers, and decimals. For example, the string STEVE cannot be converted to a number, whereas an individual's Social Security number could be stored as a character string, but could easily be converted to a numeric value via use of a conversion function.

The following is an example of a numeric conversion using an Oracle conversion function:

```
SELECT EMP_ID, TO_NUMBER(EMP_ID)
FROM EMPLOYEE_TBL;
```

EMP_ID	TO_NUMBER(EMP_ID)
-----	-----
311549902	311549902
442346889	442346889
213764555	213764555
313782439	313782439
220984332	220984332
443679012	443679012

6 rows selected.

The employee identification is right-justified following the conversion.

### **Did you Know?**

The data's justification is the simplest way to identify a column's data type.

## **Converting Numbers to Character Strings**

Converting numeric values to character strings is precisely the opposite of converting characters to numbers.

The following is an example of converting a numeric value to a character string using a Transact-SQL conversion function for Microsoft SQL Server:

```
SELECT PAY = PAY_RATE, NEW_PAY = STR(PAY_RATE)
FROM EMPLOYEE_PAY_TBL
WHERE PAY_RATE IS NOT NULL;
```

PAY	NEW_PAY
-----	-----
17.5	17.5
14.75	14.75

18.25	18.25
12.8	12.8
11	11
15	15

6 rows affected.

The following is the same example using an Oracle conversion function:

```
SELECT PAY_RATE, TO_CHAR(PAY_RATE)
FROM EMPLOYEE_PAY_TBL
WHERE PAY_RATE IS NOT NULL;
```

PAY_RATE	TO_CHAR(PAY_RATE)
17.5	17.5
14.75	14.75
18.25	18.25
12.8	12.8
11	11
15	15

6 rows selected.

## Combining Character Functions

Most functions can be combined in a SQL statement. SQL would be far too limited if function combinations were not allowed. The following example combines two functions in the query (concatenation with substring). By pulling the EMP\_ID column apart into three pieces, you can concatenate those pieces with dashes to render a readable Social Security number. This example uses the CONCAT function to combine the strings for output:

```
SELECT concat(LAST_NAME, ' ', FIRST_NAME) NAME,
       CONCAT(SUBSTR(EMP_ID,1,3), '-',
             SUBSTR(EMP_ID,4,2), '-',
             SUBSTR(EMP_ID,6,4)) AS ID
FROM EMPLOYEE_TBL;
```

NAME	ID
STEPHENS, TINA	311-54-9902
PLEW, LINDA	442-34-6889
GLASS, BRANDON	213-76-4555
GLASS, JACOB	313-78-2439
WALLACE, MARIAH	220-98-4332
SPURGEON, TIFFANY	443-67-9012

6 rows selected.

This example uses the LENGTH function and the arithmetic operator (+) to add the length of the first name to the length of the last name for each column; the SUM function then finds the total length of all first and last names.

```
SELECT SUM(LENGTH(LAST_NAME) + LENGTH(FIRST_NAME)) TOTAL
FROM EMPLOYEE_TBL;
```

```
      TOTAL
-----
71

1 row selected.
```

### By the Way

When embedding functions within functions in a SQL statement, remember that the innermost function is resolved first, and then each function is subsequently resolved from the inside out.

## Summary

You have been introduced to various functions used in a SQL statement—usually a query—to modify or enhance the way output is represented. Those functions include character, mathematical, and conversion functions. It is very important to realize that the ANSI standard is a guideline for how SQL should be implemented by vendors, but does not dictate the exact syntax or necessarily place limits on vendors' innovations. Most vendors have standard functions and conform to the ANSI concepts, but each vendor has its own specific list of available functions. The function name might differ and the syntax might differ, but the concepts with all functions are the same.

## Q&A

**Q.** *Are all the functions in the ANSI standard?*

**A.** No, not all functions are exactly ANSI SQL. Functions, like data types, are often implementation dependent. Most implementations contain supersets of the ANSI functions; many have a wide range of functions with extended capability, whereas other implementations seem to be somewhat limited. Several examples of functions from selected implementations are included in this hour. However, because so many implementations use similar functions (although they might slightly differ), check your particular implementation for available functions and their usage.

**Q.** *Is the data actually changed in the database when using functions?*

**A.** No. Data is not changed in the database when using functions. Functions are typically used in queries to manipulate the output's appearance.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, "Answers to Quizzes and Exercises," for answers.

### Quiz

**1.** Match the descriptions with the possible functions.

Descriptions	Functions
a. Used to select a portion of a character string	
b. Used to trim characters from either the right or left of a string	RPAD
c. Used to change all letters to lowercase	LPAD
d. Used to find the length of a string	RTRIM
e. Used to combine strings	UPPER
	LTRIM
	LENGTH
	LOWER
	SUBSTR

**2.** True or false: Using functions in a SELECT statement to restructure the appearance of data in output will also affect the way the data is stored in the database.

**3.** True or false: The outermost function is always resolved first when functions are embedded within other functions in a query.

## Exercises

1. Type the following code at the `mysql>` prompt to concatenate each employee's last name and first name:

```
SELECT CONCAT(LAST_NAME, ' ', FIRST_NAME)
FROM EMPLOYEE_TBL;
```

2. Type the following code to print each employee's concatenated name and their area code:

```
SELECT CONCAT(LAST_NAME, ' ', FIRST_NAME), SUBSTRING(PHONE, 1, 3)
FROM EMPLOYEE_TBL;
```

3. Write a SQL statement that lists employee emails. Email is not a stored column. The email for each employee should be as follows:

```
FIRST.LAST@PERPTECH.COM
```

For example, John Smith's email would be JOHN.SMITH@PERPTECH.COM.

4. Write a SQL statement that lists employee emails. Email is not a stored column. The email for each employee should be as follows:

```
FIRSTINITIAL.LAST@PERPTECH.COM
```

For example, John Smith's email would be JSMITH@PERPTECH.COM.

5. Write a SQL statement that lists each employee's name, employee ID, and phone number in the following formats:

- ▶ This name should be displayed as SMITH, JOHN
- ▶ The employee id should be displayed as 999-99-9999
- ▶ The phone number should be displayed as (999)999-9999

## HOUR 12

# Understanding Dates and Times

In this hour, you will learn about the nature of dates and time in SQL. Not only does this hour discuss the DATETIME data type in more detail, but you will also see how some implementations use dates, how to extract the date and time in a desired format, and some of the common rules.

---

### ***The highlights of this hour include***

- ▶ Understanding dates and time
- ▶ How date and time are stored
- ▶ Typical date and time formats
- ▶ How to use date functions
- ▶ How to use date conversions

As you know by now, there are many different SQL implementations. This book shows the ANSI standard and the most common nonstandard functions, commands, and operators. MySQL is used for the examples. Even in MySQL, the date can be stored in different formats. You must check your particular implementation for the date storage. No matter how it is stored, your implementation should have functions that convert date formats.

***By the  
Way***

## How Is a Date Stored?

Each implementation has a default storage format for the date and time. This default storage often varies among different implementations, as do other data types for each implementation. The following sections begin by reviewing the standard format of the DATETIME data type and its elements. Then you see the data types for date and time in some popular implementations of SQL, including Oracle, Sybase, and Microsoft SQL Server.

### Standard Data Types for Date and Time

There are three standard SQL data types for date and time (DATETIME) storage:

- ▶ **DATE**—Stores date literals. DATE is formatted as YYYY-MM-DD and ranges from 0001-01-01 to 9999-12-31.
- ▶ **TIME**—Stores time literals. TIME is formatted as HH:MI:SS.nn... and ranges from 00:00:00... to 23:59:61.999....
- ▶ **TIMESTAMP**—Stores date and time literals. TIMESTAMP is formatted as YYYY-MM-DD HH:MI:SS.nn... and ranges from 0001-01-01 00:00:00... to 9999-12-31 23:59:61.999....

### DATETIME Elements

DATETIME elements are those elements pertaining to date and time that are included as part of a DATETIME definition. The following is a list of the constrained DATETIME elements and a valid range of values for each element:

<b>DATETIME Element</b>	<b>Valid Ranges</b>
YEAR	0001 to 9999
MONTH	01 to 12
DAY	01 to 31
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00.000... to 61.999...

Each of these elements, except for the last, is self-explanatory; they are elements of time that we deal with on a daily basis. Seconds can be represented as a decimal, allowing the expression of tenths of a second, hundredths of a second, milliseconds,

and so on. You might question the fact that a minute can contain more than 60 seconds. According to the ANSI standard, this 61.999 seconds is due to the possible insertion or omission of a leap second in a minute, which in itself is a rare occurrence. Refer to your implementation on the allowed values because date and time storage may vary widely.

Date variances such as leap seconds and leap years are handled internally by the database if the data is stored in a DATETIME data type.

**By the  
Way**

## Implementation-Specific Data Types

As with other data types, each implementation provides its own representation and syntax. This section shows how three products (Oracle, Sybase, and MySQL) have been implemented with date and time.

Product	Data Type	Use
Oracle	DATE	Stores both date and time information
Sybase	DATETIME	Stores both date and time information
	SMALLDATETIME	Stores both date and time information, but includes a smaller date range than DATETIME
MySQL	DATETIME	Stores both date and time information
	TIMESTAMP	Stores both date and time information
	DATE	Stores a date value
	TIME	Stores a time value
	YEAR	One byte type that represents the year

Each implementation has its own specific data type(s) for date and time information. However, most implementations comply with the ANSI standard in the fact that all elements of the date and time are included in their associated data types. The way the date is internally stored is implementation dependent.

**By the  
Way**

## Date Functions

Date functions are available in SQL depending on the options with each specific implementation. *Date functions*, similar to character string functions, are used to manipulate the representation of date and time data. Available date functions are

often used to format the output of dates and time in an appealing format, compare date values with one another, compute intervals between dates, and so on.

## The Current Date

You might have already raised the question: How do I get the current date from the database? The need to retrieve the current date from the database might originate from several situations, but the current date is normally returned either to compare it to a stored date or to return the value of the current date as some sort of time-stamp.

The current date is ultimately stored on the host computer for the database and is called the *system date*. The database, which interfaces with the appropriate operating system, has the capability to retrieve the system date for its own purpose or to resolve database requests, such as queries.

Take a look at a couple of methods of attaining the system date based on commands from two different implementations.

Sybase uses a function called `GETDATE()` to return the system date. This function is used in a query as follows. The output is what would return if today's current date were New Year's Eve for 1999.

```
SELECT GETDATE()
```

```
Dec 31, 1999
```

### **By the Way**

Most options discussed in this book for Sybase's and Microsoft's implementations are applicable to both implementations because both use SQL Server for their database server. Both implementations also use an extension to standard SQL known as Transact-SQL.

MySQL uses the `NOW` function to retrieve the current date and time. `NOW` is called a *pseudocolumn* because it acts as any other column in a table and can be selected from any table in the database, although it is not actually part of the table's definition.

The following MySQL statement returns the output if today were New Year's Eve before 2002:

```
SELECT NOW ();
```

```
31-DEC-01 13:41:45
```

## Time Zones

The use of time zones might be a factor when dealing with date and time information. For instance, a time of 6:00 p.m. in central United States does not equate to the same time in Australia, although the actual point in time is the same. Some of us who live within the daylight saving time zone are used to adjusting our clocks twice a year. If time zones are considerations when maintaining data in your case, you might find it necessary to consider time zones and perform time conversions, if available with your SQL implementation.

The following are some common time zones and their abbreviations:

Abbreviation	Definition
AST, ADT	Atlantic standard, daylight time
BST, BDT	Bering standard, daylight time
CST, CDT	Central standard, daylight time
EST, EDT	Eastern standard, daylight time
GMT	Greenwich mean time
HST, HDT	Alaska/Hawaii standard, daylight time
MST, MDT	Mountain standard, daylight time
NST	Newfoundland standard, daylight time
PST, PDT	Pacific standard, daylight time
YST, YDT	Yukon standard, daylight time

The following table shows examples of time zone differences based on a given time:

Time Zone	Time
AST	June 12th, 2002 at 1:15 PM
BST	June 12th, 2002 at 6:15 AM
CST	June 12th, 2002 at 11:15 AM
EST	June 12th, 2002 at 12:15 PM
GMT	June 12th, 2002 at 5:15 PM
HST	June 12th, 2002 at 7:15 AM
MST	June 12th, 2002 at 10:15 AM
NST	June 12th, 2002 at 1:45 PM
PST	June 12th, 2002 at 9:15 AM
YST	June 12th, 2002 at 8:15 AM

Some implementations have functions that allow you to deal with different time zones. However, not all implementations may support the use of time zones. Be sure to verify the use of time zones in your particular implementation, as well as the need to deal with them in the case of your database.

## Adding Time to Dates

Days, months, and other parts of time can be added to dates for the purpose of comparing dates to one another or to provide more specific conditions in the WHERE clause of a query.

Intervals can be used to add periods of time to a DATETIME value. As defined by the standard, intervals are used to manipulate the value of a DATETIME value, as in the following examples:

```
DATE '1999-12-31' + INTERVAL '1' DAY
```

```
'2000-01-01'
```

```
DATE '1999-12-31' + INTERVAL '1' MONTH
```

```
'2000-01-31'
```

The following is an example using the SQL Server function DATEADD:

```
SELECT DATEADD(MONTH, 1, DATE_HIRE)
FROM EMPLOYEE_PAY_TBL;
```

```
DATE_HIRE  ADD_MONTH
-----
23-MAY-89  23-JUN-89
17-JUN-90  17-JUL-90
14-AUG-94  14-SEP-94
28-JUN-97  28-JUL-97
22-JUL-96  22-AUG-96
14-JAN-91  14-FEB-91
```

6 rows affected.

The following example uses the Oracle function ADD\_MONTHS:

```
SELECT DATE_HIRE, ADD_MONTHS(DATE_HIRE,1)
FROM EMPLOYEE_PAY_TBL;
```

```
DATE_HIRE          ADD_MONTH
-----
23-MAY-89          23-JUN-89
17-JUN-90          17-JUL-90
14-AUG-94          14-SEP-94
28-JUN-97          28-JUL-97
22-JUL-96          22-AUG-96
14-JAN-91          14-FEB-91
```

6 rows selected.

To add one day to a date in Oracle, use the following:

```
SELECT DATE_HIRE, DATE_HIRE + 1
FROM EMPLOYEE_PAY_TBL
WHERE EMP_ID = '311549902';
```

DATE_HIRE	DATE_HIRE
-----	-----
23-MAY-89	24-MAY-89

1 row selected.

If you wanted to do the same query in MySQL, you would use the ANSI standard INTERVAL command, as follows. Otherwise, MySQL would convert the date to an integer and try to perform the operation.

```
SELECT DATE_HIRE, DATE_ADD(DATE_HIRE, INTERVAL 1 DAY), DATE_HIRE + 1
FROM EMPLOYEE_PAY_TBL
WHERE EMP_ID = '311549902';
```

DATE_HIRE	DATE_ADD	DATE_HIRE+1
-----	-----	-----
23-MAY-89	24-MAY-89	19890524

1 row selected.

Notice that these examples in MySQL, SQL Server, and Oracle, although they differ syntactically from the ANSI examples, derive their results based on the same concept as described by the SQL standard.

## Comparing Date and Time Periods

OVERLAPS is a powerful standard SQL conditional operator for DATETIME values. The OVERLAPS operator is used to compare two timeframes and return the Boolean value TRUE or FALSE, depending on whether the two timeframes overlap. The following comparison returns the value TRUE:

```
(TIME '01:00:00' , TIME '05:59:00')
OVERLAPS
(TIME '05:00:00' , TIME '07:00:00')
```

The following comparison returns the value FALSE:

```
(TIME '01:00:00' , TIME '05:59:00')
OVERLAPS
(TIME '06:00:00' , TIME '07:00:00')
```

Unfortunately, MySQL does not implement the OVERLAPS function in terms of DATETIME data types.

## Miscellaneous Date Functions

The following list shows some powerful date functions that exist in the implementations for SQL Server, Oracle, and MySQL.

Product	Date Function	Use
SQL Server	DATEPART	Returns the integer value of a DATEPART for a date
	DATENAME	Returns the text value of a DATEPART for a date
	GETDATE()	Returns the system date
	DATEDIFF	Returns the difference between two dates for specified date parts, such as days, minutes, and seconds
Oracle	NEXT_DAY	Returns the next day of the week as specified (for example, FRIDAY) since a given date
	MONTHS_BETWEEN	Returns the number of months between two given dates
MySQL	DAYNAME(date)	Displays day of week
	DAYOFMONTH(date)	Displays day of month
	DAYOFWEEK(date)	Displays day of week
	DAYOFYEAR(date)	Displays day of year

## Date Conversions

The conversion of dates can take place for any number of reasons. Conversions are mainly used to alter the data type of values defined as a DATETIME value or any other valid data type of a particular implementation.

Typical reasons for date conversions are as follows:

- ▶ To compare date values of different data types
- ▶ To format a date value as a character string
- ▶ To convert a character string into a date format

The ANSI CAST operator is used to convert data types into other data types. The basic syntax is as follows:

```
CAST ( EXPRESSION AS NEW_DATA_TYPE )
```

Specific syntax examples of some implementations are illustrated in the following subsections, covering

- ▶ The representation of parts of a DATETIME value
- ▶ Conversions of dates to character strings
- ▶ Conversions of character strings to dates

## Date Pictures

A *date picture* is composed of formatting elements used to extract date and time information from the database in a desired format. Date pictures might not be available in all SQL implementations.

Without the use of a date picture and some type of conversion function, the date and time information is retrieved from the database in a default format, such as

```
1999-12-31  
31-DEC-99  
1999-12-31 23:59:01.11  
...
```

What if you wanted the date displayed as the following?

```
December 31, 1997
```

You would have to convert the date from a DATETIME format into a character string format. This is accomplished by implementation-specific functions for this very purpose, further illustrated in the following sections.

The following table displays some of the common date parts used in various implementations. This will aid you in using the date picture in the following sections to extract the proper datetime information from the database.

**TABLE 12.1 Common Date Parts**

<b>Product</b>	<b>Syntax</b>	<b>Date Part</b>
Sybase	yy	Year
	qq	Quarter
	mm	Month
	dy	Day of year
	wk	Week
	dw	Weekday
	hh	Hour
	mi	Minute
	ss	Second
	ms	Millisecond
Oracle	AD	Anno Domini
	AM	Ante meridian
	BC	Before Christ
	CC	Century
	D	Number of the day in the week
	DD	Number of the day in the month
	DDD	Number of the day in the year
	DAY	The day spelled out (MONDAY)
	Day	The day spelled out (Monday)
	day	The day spelled out (monday)
	DY	The three-letter abbreviation of the day (MON)
	Dy	The three-letter abbreviation of the day (Mon)
	dy	The three-letter abbreviation of the day (mon)
	HH	Hour of the day
	HH12	Hour of the day
	HH24	Hour of the day for a 24-hour clock
	J	Julian days since 12-31-4713 B.C.
	MI	Minute of the hour
	MM	The number of the month
	MON	The three-letter abbreviation of the month (JAN)
	Mon	The three-letter abbreviation of the month (Jan)
	mon	The three-letter abbreviation of the month (jan)
MONTH	The month spelled out (JANUARY)	

<b>Product</b>	<b>Syntax</b>	<b>Date Part</b>
<b>Oracle (continued)</b>		
	Month	The month spelled out (January)
	month	The month spelled out (january)
	PM	Post meridian
	Q	The number of the quarter
	RM	The Roman numeral for the month
	RR	The two digits of the year
	SS	The second of a minute
	SSSSS	The seconds since midnight
	SYYYY	The signed year; if B.C. 500, B.C. = -500
	W	The number of the week in a month
	WW	The number of the week in a year
	Y	The last digit of the year
	YY	The last two digits of the year
	YYY	The last three digits of the year
	YYYY	The year
	YEAR	The year spelled out (NINETEEN-NINETY-NINE)
	Year	The year spelled out (Nineteen-Ninety-Nine)
	year	The year spelled out (nineteen-ninety-nine)
MySQL	SECOND	Seconds
	MINUTE	Minutes
	HOUR	Hours
	DAY	Days
	MONTH	Months
	YEAR	Years
	MINUTE_SECOND	Minutes and seconds
	HOUR_MINUTE	Hours and minutes
	DAY_HOUR	Days and hours
	YEAR_MONTH	Years and months
	HOUR_SECOND	Hours, minutes, and seconds
	DAY_MINUTE	Days and minutes
	DAY_SECOND	Days and seconds

These are some of the most common date parts for MySQL. Other date parts might be available depending on the version of MySQL.

## Converting Dates to Character Strings

DATETIME values are converted to character strings to alter the appearance of output from a query. A conversion function is used to achieve this. Two examples of converting date and time data into a character string as designated by a query follow.

The first using SQL Server:

```
SELECT DATE_HIRE = DATENAME(MONTH, DATE_HIRE)
FROM EMPLOYEE_PAY_TBL;
```

```
DATE_HIRE
-----
May
June
August
June
July
Jan
```

6 rows affected.

The second example is an Oracle date conversion using the TO\_CHAR function:

```
SELECT DATE_HIRE, TO_CHAR(DATE_HIRE, 'Month dd, yyyy') HIRE
FROM EMPLOYEE_PAY_TBL;
```

DATE_HIRE	HIRE
-----	-----
23-MAY-89	May 23, 1989
17-JUN-90	June 17, 1990
14-AUG-94	August 14, 1994
28-JUN-97	June 28, 1997
22-JUL-96	July 22, 1996
14-JAN-91	January 14, 1991

6 rows selected.

## Converting Character Strings to Dates

The following example illustrates a method from a MySQL implementation of converting a character string into a date format. When the conversion is complete, the data can be stored in a column defined as having some form of a DATETIME data type.

```
SELECT STR_TO_DATE('02/25/1998 12:00:00 AM', '%m/%d/%Y %h:%i:%s %p') AS  
FORMAT_DATE  
FROM EMPLOYEE_PAY_TBL;
```

```
FORMAT_DATE
```

```
-----  
01 -JAN -99  
01 -JAN -99  
01 -JAN -99  
01 -JAN -99  
01 -JAN -99  
01 -JAN -99
```

```
6 rows selected.
```

You might be wondering why six rows were selected from this query when only one date value was provided. The reason is because the conversion of the literal string was selected from the `EMPLOYEE_PAY_TBL`, which has six rows of data. Hence, the conversion of the literal string was selected against each record in the table.

## Summary

You have an understanding of `DATETIME` values based on the fact that ANSI has provided a standard. However, as with many SQL elements, most implementations have deviated from the exact functions and syntax of standard SQL commands, although the concepts always remain the same as far as the basic representation and manipulation of date and time information. Last hour, you saw how functions varied depending on each implementation. This hour, you have seen some of the differences between date and time data types, functions, and operators. Keep in mind that not all examples discussed in this hour will work with your particular implementation, but the concepts of dates and times are the same and should be applicable to any implementation.

## Q&A

- Q.** *Why do implementations choose to deviate from a single standard set of data types and functions?*
- A.** Implementations differ as far as the representation of data types and functions mainly because of the way each vendor has chosen to internally store data and provide the most efficient means of data retrieval. However, all implementations should provide the same means for the storage of date and time values based on the required elements prescribed by ANSI, such as the year, month, day, hour, minute, second, and so on.

- Q.** *What if I want to store date and time information differently than what is available in my implementation?*
- A.** Dates can be stored in nearly any type of format if you choose to define the column for a date as a variable length character. The main thing to remember is that when comparing date values to one another, you are usually required to first convert the character string representation of the date to a valid DATETIME format for your implementation—that is, if appropriate conversion functions are available.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. From where is the system date and time normally derived?
2. List the standard internal elements of a DATETIME value.
3. What could be a major factor concerning the representation and comparison of date and time values if your company is an international organization?
4. Can a character string date value be compared to a date value defined as a valid DATETIME data type?
5. What would you use in MySQL to get the current date and time?

## Exercises

1. Type the following SQL code into the `mysql>` prompt to display the current date from the MySQL server:  

```
SELECT CURRENT_DATE;
```

2. Type the following SQL code into the `mysql>` prompt to display each employee's hire date:

```
SELECT EMP_ID, DATE_HIRE
FROM EMPLOYEE_PAY_TBL;
```

3. In MySQL, dates can be displayed in various formats using the `EXTRACT` function in conjunction with the MySQL date pictures. Type the following code to display the year that each employee was hired:

```
SELECT EMP_ID, EXTRACT(YEAR FROM DATE_HIRE)
FROM EMPLOYEE_PAY_TBL;
```

4. Type the following code to display the current date along with the date that each employee was hired:

```
SELECT EMP_ID, DATE_HIRE, CURRENT_DATE
FROM EMPLOYEE_PAY_TBL;
```

5. On what day of the week was each employee hired?
6. What is today's Julian date (day of year)?
7. Type the following SQL code into the `mysql>` prompt to compare the results of casting the current date and time to different data types:

```
SELECT NOW()
FROM EMPLOYEE_PAY_TBL;
```

```
SELECT CAST(NOW() AS DATE)
FROM EMPLOYEE_PAY_TBL;
```

```
SELECT CAST(NOW() AS TIME)
FROM EMPLOYEE_PAY_TBL;
```

*This page intentionally left blank*

## PART IV

# Building Sophisticated Database Queries

<b>HOUR 13</b>	Joining Tables in Queries	<b>203</b>
<b>HOUR 14</b>	Using Subqueries to Define Unknown Data	<b>221</b>
<b>HOUR 15</b>	Combining Multiple Queries into One	<b>235</b>

*This page intentionally left blank*

## HOOR 13

# Joining Tables in Queries

To this point, all database queries you have executed have extracted data from a single table. During this hour, you learn how to join tables in a query so data can be retrieved from multiple tables.

---

### ***The highlights of this hour include***

- ▶ An introduction to the table joins
- ▶ The different types of joins
- ▶ How and when joins are used
- ▶ Numerous practical examples of table joins
- ▶ The effects of improperly joined tables
- ▶ Renaming tables in a query using an alias

## **Selecting Data from Multiple Tables**

Having the capability to select data from multiple tables is one of SQL's most powerful features. Without this capability, the entire relational database concept would not be feasible. Single-table queries are sometimes quite informative, but in the real world, the most practical queries are those whose data is acquired from multiple tables within the database.

As you witnessed in Hour 4, "The Normalization Process," a relational database is broken up into smaller, more manageable tables for simplicity and the sake of overall management ease. As tables are divided into smaller tables, the related tables are created with common columns—*primary keys* and *foreign keys*. These keys are used to join related tables to one another.

You might ask why you should normalize tables if, in the end, you are only going to rejoin the tables to retrieve the data you want. You rarely select all data from all tables, so it is better to pick and choose according to the needs of each individual query. Although performance might suffer slightly due to a normalized database, overall coding and maintenance are much simpler. Remember that you generally normalize the database to reduce redundancy and increase data integrity. Your over-reaching task as a database administrator is to ensure the safeguarding of data.

## Types of Joins

A *join* combines two or more tables to retrieve data from multiple tables. Although different implementations have many ways of joining tables, you concentrate on the most common joins in this lesson. The types of joins that you learn are

- ▶ Equijoins or inner joins
- ▶ Natural joins
- ▶ Non-equijoins
- ▶ Outer joins
- ▶ Self joins

## Component Locations of a Join Condition

As you have learned from previous hours, the `SELECT` and `FROM` clauses are both required SQL statement elements; the `WHERE` clause is a required element of a SQL statement when joining tables. The tables being joined are listed in the `FROM` clause. The join is performed in the `WHERE` clause. Several operators can be used to join tables, such as `=`, `<`, `>`, `<>`, `<=`, `>=`, `!=`, `BETWEEN`, `LIKE`, and `NOT`. However, the most common operator is the equal symbol.

## Joins of Equality

Perhaps the most used and important of the joins is the equijoin, also referred to as an inner join. The *equijoin* joins two tables with a common column in which each is usually the primary key.

The syntax for an equijoin is

```
SELECT TABLE1.COLUMN1, TABLE2.COLUMN2...
FROM TABLE1, TABLE2 [, TABLE3 ]
WHERE TABLE1.COLUMN_NAME = TABLE2.COLUMN_NAME
[ AND TABLE1.COLUMN_NAME = TABLE3.COLUMN_NAME ]
```

Look at the following example:

```
SELECT EMPLOYEE_TBL.EMP_ID,  
       EMPLOYEE_PAY_TBL.DATE_HIRE  
FROM EMPLOYEE_TBL,  
     EMPLOYEE_PAY_TBL  
WHERE EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;
```

Take note of the sample SQL statements. Indentation is used in the SQL statements to improve overall readability. Indentation is not required, but is recommended.

**By the  
Way**

This SQL statement returns the employee identification and the employee's date of hire. The employee identification is selected from the EMPLOYEE\_TBL (although it exists in both tables, you must specify one table), and the hire date is selected from the EMPLOYEE\_PAY\_TBL. Because the employee identification exists in both tables, both columns must be justified with the table name. By justifying the columns with the table names, you tell the database server where to get the data.

Data in the following example is selected from the EMPLOYEE\_TBL and EMPLOYEE\_PAY\_TBL tables because desired data resides in each of the two tables. An equality join is used.

```
SELECT EMPLOYEE_TBL.EMP_ID, EMPLOYEE_TBL.LAST_NAME,  
       EMPLOYEE_PAY_TBL.POSITION  
FROM EMPLOYEE_TBL, EMPLOYEE_PAY_TBL  
WHERE EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;
```

```
EMP_ID    LAST_NAM POSITION  
-----  
311549902 STEPHENS MARKETING  
442346889 PLEW      TEAM LEADER  
213764555 GLASS    SALES MANAGER  
313782439 GLASS    SALESMAN  
220984332 WALLACE  SHIPPER  
443679012 SPURGEON SHIPPER
```

6 rows selected.

Notice that each column in the SELECT clause is preceded by the associated table name in order to identify each column. This is called *qualifying columns* in a query. Qualifying columns is only necessary for columns that exist in more than one table referenced by a query. You usually qualify all columns for consistency and to avoid any questions when debugging or modifying SQL code.

Additionally, the SQL syntax provides for a more readable version of the previous syntax by introducing the JOIN syntax. The JOIN syntax is as follows:

```
SELECT TABLE1.COLUMN1, TABLE2.COLUMN2...
FROM TABLE1
INNER JOIN TABLE2 ON TABLE1.COLUMN_NAME = TABLE2.COLUMN_NAME
```

As you can see, the join operator is removed from the WHERE clause and instead replaced with the JOIN syntax. The table being joined is added after the JOIN syntax and then the JOIN operators are placed after the ON qualifier. In the following example, the previous query for employee identification and hire date is rewritten to use the JOIN syntax:

```
SELECT EMPLOYEE_TBL.EMP_ID,
       EMPLOYEE_PAY_TBL.DATE_HIRE
FROM EMPLOYEE_TBL
INNER JOIN EMPLOYEE_PAY_TBL
ON EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;
```

You will notice that this query returns the same set of data as the previous version, even though the syntax is different. So you may use either version of the syntax without fear of coming up with different results.

## Natural Joins

A *natural join* is nearly the same as the equijoin; however, the natural join differs from the equijoin by eliminating duplicate columns in the joining columns. The JOIN condition is the same, but the columns selected differ. The syntax is as follows:

```
SELECT TABLE1.*, TABLE2.COLUMN_NAME
       [ TABLE3.COLUMN_NAME ]
FROM TABLE1, TABLE2 [ TABLE3 ]
WHERE TABLE1.COLUMN_NAME = TABLE2.COLUMN_NAME
       [ AND TABLE1.COLUMN_NAME = TABLE3.COLUMN_NAME ]
```

Look at the following example:

```
SELECT EMPLOYEE_TBL.*, EMPLOYEE_PAY_TBL.SALARY
FROM EMPLOYEE_TBL,
     EMPLOYEE_PAY_TBL
WHERE EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;
```

This SQL statement returns all columns from EMPLOYEE\_TBL and SALARY from the EMPLOYEE\_PAY\_TBL. The EMP\_ID is in both tables, but is retrieved only from the EMPLOYEE\_TBL because both contain the same information and do not need to be selected.

Alternatively, you may use a `NATURAL JOIN` syntax similar to the `INNER JOIN` syntax described in the previous section. The syntax is very similar:

```
SELECT TABLE1.*, TABLE2.COLUMN_NAME
FROM TABLE1
NATURAL JOIN TABLE2 ON TABLE1.COLUMN_NAME = TABLE2.COLUMN_NAME
```

Look at the following example using this syntax:

```
SELECT EMPLOYEE_TBL.*, EMPLOYEE_PAY_TBL.SALARY
FROM EMPLOYEE_TBL
NATURAL JOIN EMPLOYEE_PAY_TBL
ON EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;
```

The following example selects all columns from the `EMPLOYEE_TBL` table and only one column from the `EMPLOYEE_PAY_TBL` table. Remember that the asterisk (\*) represents all the columns in a table.

```
SELECT EMPLOYEE_TBL.*, EMPLOYEE_PAY_TBL.POSITION
FROM EMPLOYEE_TBL, EMPLOYEE_PAY_TBL
WHERE EMPLOYEE_TBL.EMP_ID = EMPLOYEE_PAY_TBL.EMP_ID;
```

EMP_ID	LAST_NAM	FIRST_NA	M	ADDRESS	CITY	ST	ZIP	PHONE
	PAGER	POSITION						
311549902	STEPHENS	TINA		D RR 3 BOX 17A	GREENWOOD	IN	47890	3178784465
		MARKETING						
442346889	PLEW	LINDA		C 3301 BEACON	INDIANAPOLIS	IN	46224	3172978990
		TEAM LEADER						
213764555	GLASS	BRANDON		S 1710 MAIN ST	WHITELAND	IN	47885	3178984321
3175709980		SALES MANAGER						
313782439	GLASS	JACOB		3789 RIVER BLVD	INDIANAPOLIS	IN	45734	3175457676
8887345678		SALESMAN						
220984332	WALLACE	MARIAH		7889 KEYSTONE	INDIANAPOLIS	IN	46741	3173325986
		SHIPPER						
443679012	SPURGEON	TIFFANY		5 GEORGE COURT	INDIANAPOLIS	IN	46234	3175679007
		SHIPPER						

6 rows selected.

Notice how the output has wrapped in the previous example. The wrap occurred because the length of the line has exceeded the limit for the line length within the query editor window (which is 80 characters per line).

## Using Table Aliases

The use of *table aliases* means to rename a table in a particular SQL statement. The renaming is a temporary change. The actual table name does not change in the database. As you will learn later in the section on Self Joins, giving the tables aliases is a necessity for the self join. Giving tables aliases is most often used to save keystrokes, which results in a shorter and easier-to-read SQL statement. In addition, fewer keystrokes means fewer keystroke errors. Also, programming errors are typically less frequent if you can refer to an alias, which is often shorter in length and more descriptive of the data with which you are working. Giving tables aliases also means that the columns being selected must be qualified with the table alias. The following are some examples of table aliases and the corresponding columns:

```
SELECT E.EMP_ID, EP.SALARY, EP.DATE_HIRE, E.LAST_NAME
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
AND EP.SALARY > 20000;
```

The tables have been given aliases in the preceding SQL statement. The `EMPLOYEE_TBL` has been renamed `E`. The `EMPLOYEE_PAY_TBL` has been renamed `EP`. The choice of what to rename the tables is arbitrary. The letter *E* is chosen because the `EMPLOYEE_TBL` starts with *E*. Because the `EMPLOYEE_PAY_TBL` also begins with the letter *E*, you could not use *E* again. Instead, the first letter (*E*) and the first letter of the second word in the name (*PAY*) are used as the alias. The selected columns were justified with the corresponding table alias. Note that `SALARY` was used in the `WHERE` clause and must also be justified with the table alias.

## Joins of Non-Equality

A *non-equi*join joins two or more tables based on a specified column value not equaling a specified column value in another table. The syntax for the non-equi join is

```
FROM TABLE1, TABLE2 [, TABLE3 ]
WHERE TABLE1.COLUMN_NAME != TABLE2.COLUMN_NAME
[ AND TABLE1.COLUMN_NAME != TABLE2.COLUMN_NAME ]
```

An example is as follows:

```
SELECT EMPLOYEE_TBL.EMP_ID, EMPLOYEE_PAY_TBL.DATE_HIRE
FROM EMPLOYEE_TBL,
     EMPLOYEE_PAY_TBL
WHERE EMPLOYEE_TBL.EMP_ID != EMPLOYEE_PAY_TBL.EMP_ID;
```

The preceding SQL statement returns the employee identification and the date of hire for all employees who do not have a corresponding record in both tables. The following example is a join of non-equality:

```

SELECT E.EMP_ID, E.LAST_NAME, P.POSITION
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL P
WHERE E.EMP_ID <> P.EMP_ID;

```

```

EMP_ID   LAST_NAM POSITION
-----
442346889 PLEW     MARKETING
213764555 GLASS    MARKETING
313782439 GLASS    MARKETING
220984332 WALLACE  MARKETING
443679012 SPURGEON MARKETING
311549902 STEPHENS TEAM LEADER
213764555 GLASS    TEAM LEADER
313782439 GLASS    TEAM LEADER
220984332 WALLACE  TEAM LEADER
443679012 SPURGEON TEAM LEADER
311549902 STEPHENS SALES MANAGER
442346889 PLEW     SALES MANAGER
313782439 GLASS    SALES MANAGER
220984332 WALLACE  SALES MANAGER
443679012 SPURGEON SALES MANAGER
311549902 STEPHENS SALESMAN
442346889 PLEW     SALESMAN
213764555 GLASS    SALESMAN
220984332 WALLACE  SALESMAN
443679012 SPURGEON SALESMAN
311549902 STEPHENS SHIPPER
442346889 PLEW     SHIPPER
213764555 GLASS    SHIPPER
313782439 GLASS    SHIPPER
443679012 SPURGEON SHIPPER
311549902 STEPHENS SHIPPER
442346889 PLEW     SHIPPER
213764555 GLASS    SHIPPER
313782439 GLASS    SHIPPER
220984332 WALLACE  SHIPPER

```

30 rows selected.

You might be curious why 30 rows were retrieved when only 6 rows exist in each table. For every record in `EMPLOYEE_TBL`, there is a corresponding record in `EMPLOYEE_PAY_TBL`. Because non-equality was tested in the join of the two tables, each row in the first table is paired with all rows from the second table, except for its own corresponding row. This means that each of the 6 rows is paired with 5 unrelated rows in the second table; 6 rows multiplied by 5 rows equals 30 rows total.

In the earlier section's test for equality example, each of the six rows in the first table were paired with only one row in the second table (each row's corresponding row); six rows multiplied by one row yields a total of six rows.

**Watch  
Out!**

When using non-equi joins, you might receive several rows of data that are of no use to you. Check your results carefully.

## Outer Joins

An *outer join* is used to return all rows that exist in one table, even though corresponding rows do not exist in the joined table. The (+) symbol is used to denote an outer join in a query. The (+) is placed at the end of the table name in the WHERE clause. The table with the (+) should be the table that does not have matching rows. In many implementations, the outer join is broken down into joins called left outer join, right outer join, and full outer join. The outer join in these implementations is normally optional.

**By the  
Way**

You must check your particular implementation for exact usage and syntax of the outer join. The (+) symbol is used by some major implementations, but is non-standard.

The general syntax for an outer join is

```
FROM TABLE1
{RIGHT | LEFT | FULL} [OUTER] JOIN
ON TABLE2
```

The Oracle syntax is

```
FROM TABLE1, TABLE2 [, TABLE3 ]
WHERE TABLE1.COLUMN_NAME[ (+) ] = TABLE2.COLUMN_NAME[ (+) ]
[ AND TABLE1.COLUMN_NAME[ (+) ] = TABLE3.COLUMN_NAME[ (+) ]]
```

**By the  
Way**

The outer join can only be used on one side of a JOIN condition; however, you can use an outer join on more than one column of the same table in the JOIN condition.

The concept of the outer join is explained in the next two examples. In the first example, the product description and the quantity ordered are selected; both values are extracted from two separate tables. One important factor to keep in mind is that there might not be a corresponding record in the ORDERS\_TBL table for every product. A regular join of equality is performed:

```

SELECT P.PROD_DESC, O.QTY
FROM PRODUCTS_TBL P,
     ORDERS_TBL O
WHERE P.PROD_ID = O.PROD_ID;

```

PROD_DESC	QTY
WITCHES COSTUME	1
PLASTIC PUMPKIN 18 INCH	25
PLASTIC PUMPKIN 18 INCH	2
LIGHTED LANTERNS	10
FALSE PARAFFIN TEETH	20
KEY CHAIN	1

6 rows selected.

Only six rows were selected, but there are 10 distinct products. You want to display all products, whether the products have been placed on order or not.

The next example accomplishes the desired output through the use of an outer join. Oracle's syntax is used here.

```

SELECT P.PROD_DESC, O.QTY
FROM PRODUCTS_TBL P,
     ORDERS_TBL O
WHERE P.PROD_ID = O.PROD_ID(+);

```

PROD_DESC	QTY
WITCHES COSTUME	1
ASSORTED MASKS	
FALSE PARAFFIN TEETH	20
ASSORTED COSTUMES	
PLASTIC PUMPKIN 18 INCH	25
PLASTIC PUMPKIN 18 INCH	2
PUMPKIN CANDY	
PLASTIC SPIDERS	
CANDY CORN	
LIGHTED LANTERNS	10
KEY CHAIN	1
OAK BOOKSHELF	

12 rows selected.

All products were returned by the query, even though they might not have had a quantity ordered. The outer join is inclusive of all rows of data in the PRODUCTS\_TBL table, whether a corresponding row exists in the ORDERS\_TBL table or not.

## Self Joins

The *self join* is used to join a table to itself, as if the table were two tables, temporarily renaming at least one table in the SQL statement using a table alias. The syntax is as follows:

```
SELECT A.COLUMN_NAME, B.COLUMN_NAME, [ C.COLUMN_NAME ]
FROM TABLE1 A, TABLE2 B [, TABLE3 C ]
WHERE A.COLUMN_NAME = B.COLUMN_NAME
[ AND A.COLUMN_NAME = C.COLUMN_NAME ]
```

The following is an example:

```
SELECT A.LAST_NAME, B.LAST_NAME, A.FIRST_NAME
FROM EMPLOYEE_TBL A,
     EMPLOYEE_TBL B
WHERE A.LAST_NAME = B.LAST_NAME;
```

The preceding SQL statement returns the employees' first names for all the employees with the same last name from the EMPLOYEE\_TBL. Self joins are useful when all of the data you want to retrieve resides in one table, but you must somehow compare records in the table to other records in the table.

You may also use the alternate INNER JOIN syntax as shown below to give the same result:

```
SELECT A.LAST_NAME, B.LAST_NAME, A.FIRST_NAME
FROM EMPLOYEE_TBL A
INNER JOIN EMPLOYEE_TBL B
ON A.LAST_NAME = B.LAST_NAME;
```

Another common example used to explain a self join is as follows: Suppose you have a table that stores an employee identification number, the employee's name, and the employee identification number of the employee's manager. You might want to produce a list of all employees and their managers' names. The problem is that the manager name does not exist as a category in the table:

```
SELECT * FROM EMP;
```

ID	NAME	MGR_ID
1	JOHN	0
2	MARY	1
3	STEVE	1
4	JACK	2
5	SUE	2

In the following example, we have included the table EMP twice in the FROM clause of the query, giving the table two aliases for the purpose of the query. By providing two aliases, it is as if you are selecting from two distinct tables. All managers are

also employees, so the JOIN condition between the two tables compares the value of the employee identification number from the first table with the manager identification number in the second table. The first table acts as a table that stores employee information, whereas the second table acts as a table that stores manager information:

```
SELECT E1.NAME, E2.NAME
FROM EMP E1, EMP E2
WHERE E1.MGR_ID = E2.ID;
```

```
NAME      NAME
-----
MARY      JOHN
STEVE     JOHN
JACK      MARY
SUE       MARY
```

## Joining on Multiple Keys

Most join operations involve the merging of data based on a key in one table and a key in another table. Depending on how your database has been designed, you might have to join on more than one key field to accurately depict that data in your database. You might have a table that has a primary key that is comprised of more than one column. You might also have a foreign key in a table that consists of more than one column, which references the multiple column primary key.

Consider the following Oracle tables that are used here for examples only:

```
SQL> desc prod
Name                                     Null?    Type
-----
SERIAL_NUMBER                           NOT NULL NUMBER(10)
VENDOR_NUMBER                           NOT NULL NUMBER(10)
PRODUCT_NAME                            NOT NULL VARCHAR2(30)
COST                                      NOT NULL NUMBER(8,2)
```

```
SQL> desc ord
Name                                     Null?    Type
-----
ORD_NO                                   NOT NULL NUMBER(10)
PROD_NUMBER                             NOT NULL NUMBER(10)
VENDOR_NUMBER                           NOT NULL NUMBER(10)
QUANTITY                                 NOT NULL NUMBER(5)
ORD_DATE                                 NOT NULL DATE
```

The primary key in PROD is the combination of the columns SERIAL\_NUMBER and VENDOR\_NUMBER. Perhaps two products can have the same serial number within the distribution company, but each serial number is unique per vendor.

The foreign key in ORD is also the combination of the columns SERIAL\_NUMBER and VENDOR\_NUMBER.

When selecting data from both tables (PROD and ORD), the join operation might appear as follows:

```
SELECT P.PRODUCT_NAME, O.ORD_DATE, O.QUANTITY
FROM PROD P, ORD O
WHERE P.SERIAL_NUMBER = O.SERIAL_NUMBER
      AND P.VENDOR_NUMBER = O.VENDOR_NUMBER;
```

Similarly, if you were using the INNER JOIN syntax, you would merely list the multiple join operations after the ON keyword, as shown below:

```
SELECT P.PRODUCT_NAME, O.ORD_DATE, O.QUANTITY
FROM PROD P,
INNER JOIN ORD O ON P.SERIAL_NUMBER = O.SERIAL_NUMBER
      AND P.VENDOR_NUMBER = O.VENDOR_NUMBER;
```

## Join Considerations

Several things should be considered before using joins: what columns(s) to join on, whether there is no common column to join on, and performance issues. More joins in a query means the database server has to do more work, which means that more time is taken to retrieve data. Joins cannot be avoided when retrieving data from a normalized database, but it is imperative to ensure that joins are performed correctly from a logical standpoint. Incorrect joins can result in serious performance degradation and inaccurate query results. Performance issues are discussed in more detail in Hour 18, “Managing Database Users.”

## Using a Base Table

What to join on? Should you have the need to retrieve data from two tables that do not have a common column to join, you must join on another table that has a common column or columns to both tables. That table becomes the base table. A *base table* is used to join one or more tables that have common columns, or to join tables that do not have common columns. Use the following three tables for an example of a base table:

CUSTOMER_TBL			
CUST_ID	VARCHAR(10)	NOT NULL	primary key
CUST_NAME	VARCHAR(30)	NOT NULL	
CUST_ADDRESS	VARCHAR(20)	NOT NULL	
CUST_CITY	VARCHAR(15)	NOT NULL	
CUST_STATE	VARCHAR(2)	NOT NULL	
CUST_ZIP	INTEGER(5)	NOT NULL	
CUST_PHONE	INTEGER(10)		
CUST_FAX	INTEGER(10)		

```

ORDERS_TBL
ORD_NUM          VARCHAR(10)    NOT NULL    primary key
CUST_ID          VARCHAR(10)    NOT NULL
PROD_ID          VARCHAR(10)    NOT NULL
QTY              INTEGER(6)     NOT NULL
ORD_DATE         DATETIME

PRODUCTS_TBL
PROD_ID          VARCHAR(10)    NOT NULL    primary key
PROD_DESC        VARCHAR(40)    NOT NULL
COST             DECIMAL(6,2)   NOT NULL

```

You have a need to use the CUSTOMERS\_TBL and the PRODUCTS\_TBL. There is no common column in which to join the tables. Now look at the ORDERS\_TBL. The ORDERS\_TBL has a CUST\_ID column to join with CUSTOMERS\_TBL, which also has a CUST\_ID column. The PRODUCTS\_TBL has a PROD\_ID column, which is also in ORDERS\_TBL. The JOIN conditions and results look like the following:

```

SELECT C.CUST_NAME, P.PROD_DESC
FROM CUSTOMER_TBL C,
     PRODUCTS_TBL P,
     ORDERS_TBL O
WHERE C.CUST_ID = O.CUST_ID
     AND P.PROD_ID = O.PROD_ID;

```

CUST_NAME	PROD_DESC
LESLIE GLEASON	WITCHES COSTUME
SCHYLERS NOVELTIES	PLASTIC PUMPKIN 18 INCH
WENDY WOLF	PLASTIC PUMPKIN 18 INCH
GAVINS PLACE	LIGHTED LANTERNS
SCOTTYS MARKET	FALSE PARAFFIN TEETH
ANDYS CANDIES	KEY CHAIN

6 rows selected.

Note the use of table aliases and their use on the columns in the WHERE clause.

**By the  
Way**

## The Cartesian Product

The *Cartesian product* is a result of a Cartesian join or “no join.” If you select from two or more tables and do not join the tables, your output is all possible rows from all the tables selected. If your tables were large, the result could be hundreds of thousands, or even millions, of rows of data. A WHERE clause is highly recommended for SQL statements retrieving data from two or more tables. The Cartesian product is also known as a *cross join*.

The syntax is

```
FROM TABLE1, TABLE2 [, TABLE3 ]
WHERE TABLE1, TABLE2 [, TABLE3 ]
```

The following is an example of a cross join, or the dreaded Cartesian product:

```
SELECT E.EMP_ID, E.LAST_NAME, P.POSITION
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL P;
```

```
EMP_ID    LAST_NAM POSITION
-----
311549902 STEPHENS MARKETING
442346889 PLEW      MARKETING
213764555 GLASS     MARKETING
313782439 GLASS     MARKETING
220984332 WALLACE  MARKETING
443679012 SPURGEON MARKETING
311549902 STEPHENS TEAM LEADER
442346889 PLEW      TEAM LEADER
213764555 GLASS     TEAM LEADER
313782439 GLASS     TEAM LEADER
220984332 WALLACE  TEAM LEADER
443679012 SPURGEON TEAM LEADER
311549902 STEPHENS SALES MANAGER
442346889 PLEW      SALES MANAGER
213764555 GLASS     SALES MANAGER
313782439 GLASS     SALES MANAGER
220984332 WALLACE  SALES MANAGER
443679012 SPURGEON SALES MANAGER
311549902 STEPHENS SALESMAN
442346889 PLEW      SALESMAN
213764555 GLASS     SALESMAN
313782439 GLASS     SALESMAN
220984332 WALLACE  SALESMAN
443679012 SPURGEON SALESMAN
311549902 STEPHENS SHIPPER
442346889 PLEW      SHIPPER
213764555 GLASS     SHIPPER
313782439 GLASS     SHIPPER
220984332 WALLACE  SHIPPER
443679012 SPURGEON SHIPPER
311549902 STEPHENS SHIPPER
442346889 PLEW      SHIPPER
213764555 GLASS     SHIPPER
313782439 GLASS     SHIPPER
220984332 WALLACE  SHIPPER
443679012 SPURGEON SHIPPER
```

36 rows selected.

Data is being selected from two separate tables, yet no JOIN operation is performed. Because you have not specified how to join rows in the first table with rows in the second table, the database server pairs every row in the first table with every row in the second table. Because each table has 6 rows of data each, the product of 36 rows selected is achieved from 6 rows multiplied by 6 rows.

To fully understand exactly how the Cartesian product is derived, study the following example.

```
SQL> SELECT X FROM TABLE1;
```

```
X  
-  
A  
B  
C  
D
```

```
4 rows selected.
```

```
SQL> SELECT V FROM TABLE2;
```

```
X  
-  
A  
B  
C  
D
```

```
4 rows selected.
```

```
SQL> SELECT TABLE1.X, TABLE2.X  
2* FROM TABLE1, TABLE2;
```

```
X X  
- -  
A A  
B A  
C A  
D A  
A B  
B B  
C B  
D B  
A C  
B C  
C C  
D C  
A D  
B D  
C D  
D D
```

```
16 rows selected.
```

**Watch  
Out!**

Be careful to always join all tables in a query. If two tables in a query have not been joined and each table contains 1,000 rows of data, the Cartesian product consists of 1,000 rows multiplied by 1,000 rows, which results in a total of 1,000,000 rows of data returned. Cartesian products, when dealing with large amounts of data, can cause the host computer to stall or crash in some cases, based on resource usage on the host computer. Therefore, it is important for the DBA and system administrator to closely monitor for long-running queries.

## Summary

You have been introduced to one of the most robust features of SQL—the table join. Imagine the limits if you were not able to extract data from more than one table in a single query. You were shown several types of joins, each serving its own purpose depending on conditions placed on the query. Joins are used to link data from tables based on equality and non-equality. Outer joins are very powerful, allowing data to be retrieved from one table, even though associated data is not found in a joined table. Self joins are used to join a table to itself. Beware of the cross join, more commonly known as the Cartesian product. The Cartesian product is the resultset of a multiple table query without a join, often yielding a large amount of unwanted output. When selecting data from more than one table, be sure to properly join the tables according to the related columns (normally primary keys). Failure to properly join tables could result in incomplete or inaccurate output.

## Q&A

- Q.** *When joining tables, must they be joined in the same order that they appear in the FROM clause?*
- A.** No, they do not have to appear in the same order; however, performance might be benefited depending on the order of tables in the FROM clause and the order in which tables are joined.
- Q.** *When using a base table to join unrelated tables, must I select any columns from the base table?*
- A.** No, the use of a base table to join unrelated tables does not mandate that columns from the base table be selected.

**Q. Can I join on more than one column between tables?**

- A.** Yes, some queries might require you to join on more than one column per table to provide a complete relationship between rows of data in the joined tables.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

### Quiz

1. What type of join would you use to return records from one table, regardless of the existence of associated records in the related table?
2. The JOIN conditions are located in which parts of the SQL statement?
3. What type of JOIN do you use to evaluate equality among rows of related tables?
4. What happens if you select from two different tables but fail to join the tables?
5. Use the following tables:

ORDERS_TBL			
ORD_NUM	VARCHAR(10)	NOT NULL	primary key
CUST_ID	VARCHAR(10)	NOT NULL	
PROD_ID	VARCHAR(10)	NOT NULL	
QTY	Integer(6)	NOT NULL	
ORD_DATE	DATETIME		
PRODUCTS_TBL			
PROD_ID	VARCHAR(10)	NOT NULL	primary key
PROD_DESC	VARCHAR(40)	NOT NULL	
COST	DECIMAL(,2)	NOT NULL	

Is the following syntax correct for using an outer join?

```
SELECT C.CUST_ID, C.CUST_NAME, O.ORD_NUM
FROM CUSTOMER_TBL C, ORDERS_TBL O
WHERE C.CUST_ID(+) = O.CUST_ID(+)
```

## Exercises

1. Invoke MySQL, point to your `learnsql` database, and type the following code and study the resultset (Cartesian product):

```
SELECT E.LAST_NAME, E.FIRST_NAME, EP.DATE_HIRE
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP;
```

2. Type the following code to properly join the `EMPLOYEE_TBL` and `EMPLOYEE_PAY_TBL` tables:

```
SELECT E.LAST_NAME, E.FIRST_NAME, EP.DATE_HIRE
FROM EMPLOYEE_TBL E,
     EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID;
```

3. Rewrite the SQL query from Exercise 2, using the `INNER JOIN` syntax.
4. Write a SQL statement to return the `EMP_ID`, `LAST_NAME`, and `FIRST_NAME` columns from the `EMPLOYEE_TBL` and `SALARY` and `BONUS` columns from the `EMPLOYEE_PAY_TBL`. Use both types of `INNER JOIN` techniques.
5. What is the average employee salary per city?
6. Try writing a few queries with join operations on your own.

## HOUR 14

# Using Subqueries to Define Unknown Data

During this hour, you are presented with the concept of using subqueries to return results from a database query more effectively.

---

### ***The highlights of this hour include***

- ▶ What a subquery is
- ▶ The justifications of using subqueries
- ▶ Examples of subqueries in regular database queries
- ▶ Using subqueries with data manipulation commands
- ▶ Embedded subqueries

## **What Is a Subquery?**

A *subquery*, also known as a *nested query*, is a query embedded within the WHERE clause of another query to further restrict data returned by the query. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Subqueries are used with the SELECT, INSERT, UPDATE, and DELETE statements.

A subquery can be used in some cases in place of a join operation by indirectly linking data between the tables based on one or more conditions. When a subquery is used in a query, the subquery is resolved first, and then the main query is resolved according to the condition(s) resolved by the subquery. The results of the subquery are used to process expressions in the WHERE clause of the main query. The subquery can be used either in the WHERE clause or the HAVING clause of the main query. Logical and relational operators, such as =, >, <, <>, !=, IN, NOT IN, AND, OR, and so on, can be used within the subquery as well as to evaluate a subquery in the WHERE or HAVING clause.



The following is an example of an illegal use of BETWEEN with a subquery:

```
SELECT COLUMN_NAME
FROM TABLE
WHERE COLUMN_NAME BETWEEN VALUE AND (SELECT COLUMN_NAME
                                       FROM TABLE)
```

BETWEEN cannot be used as an operator outside the subquery.

## Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement, although they can be used within a data manipulation statement as well. The subquery, when used with the SELECT statement, retrieves data for the main query to use.

The basic syntax is as follows:

```
SELECT COLUMN_NAME [, COLUMN_NAME ]
FROM TABLE1 [, TABLE2 ]
WHERE COLUMN_NAME OPERATOR
      (SELECT COLUMN_NAME [, COLUMN_NAME ]
       FROM TABLE1 [, TABLE2 ]
       [ WHERE ])
```

The following is an example:

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.PAY_RATE
FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
AND EP.PAY_RATE < (SELECT PAY_RATE
                   FROM EMPLOYEE_PAY_TBL
                   WHERE EMP_ID = '443679012');
```

The preceding SQL statement returns the employee identification, last name, first name, and pay rate for all employees who have a pay rate greater than that of the employee with the identification 313782439. In this case, you do not necessarily know (or care) what the exact pay rate is for this particular employee; you only care about the pay rate for the purpose of getting a list of employees who bring home more than the employee specified in the subquery.

The next query selects the pay rate for a particular employee. This query is used as the subquery in the following example.

```
SELECT PAY_RATE
FROM EMPLOYEE_PAY_TBL
WHERE EMP_ID = '220984332';
```

```
  PAY_RATE
  ....
11
```

1 row selected.

## HOOR 14: Using Subqueries to Define Unknown Data

The previous query is used as a subquery in the WHERE clause of the following query:

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.PAY_RATE
FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
      AND EP.PAY_RATE > (SELECT PAY_RATE
                        FROM EMPLOYEE_PAY_TBL
                        WHERE EMP_ID = '220984332');
```

EMP_ID	LAST_NAME	FIRST_NAME	PAY_RATE
442346889	PLEW	LINDA	14.75
443679012	SPURGEON	TIFFANY	15

2 rows selected.

The result of the subquery is 11 (shown in the last example), so the last condition of the WHERE clause is evaluated as

```
AND EP.PAY_RATE > 11
```

You did not know the value of the pay rate for the given individual when you executed the query. However, the main query was able to compare each individual's pay rate to the subquery results.

Subqueries are frequently used to place conditions on a query when the exact conditions are unknown. The salary for 220984332 was unknown, but the subquery was designed to do the footwork for you.

## Subqueries with the INSERT Statement

Subqueries also can be used in conjunction with Data Manipulation Language (DML) statements. The INSERT statement is the first instance you will examine. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery can be modified with any of the character, date, or number functions.

The basic syntax is as follows:

```
INSERT INTO TABLE_NAME [ (COLUMN1 [, COLUMN2] ) ]
SELECT [ *|COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE VALUE OPERATOR ]
```

The following is an example of the INSERT statement with a subquery:

```
INSERT INTO RICH_EMPLOYEES
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME, EP.PAY_RATE
FROM EMPLOYEE_TBL E, EMPLOYEE_PAY_TBL EP
WHERE E.EMP_ID = EP.EMP_ID
      AND EP.PAY_RATE > (SELECT PAY_RATE
                        FROM EMPLOYEE_PAY_TBL
                        WHERE EMP_ID = '220984332');
```

2 rows created.

This INSERT statement inserts the EMP\_ID, LAST\_NAME, FIRST\_NAME, and PAY\_RATE into a table called RICH\_EMPLOYEES for all records of employees who have a pay rate greater than the pay rate of the employee with identification 220984332.

Remember to use the COMMIT and ROLLBACK commands when using DML commands such as the INSERT statement.

**By the  
Way**

## Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement to update single or multiple columns in a table. The basic syntax is as follows:

```
UPDATE TABLE
SET COLUMN_NAME [, COLUMN_NAME] =
  (SELECT )COLUMN_NAME [, COLUMN_NAME]
FROM TABLE
[ WHERE ]
```

Examples showing the use of the UPDATE statement with a subquery follow. The first query returns the employee identification of all employees who reside in Indianapolis. You can see that four individuals meet this criterion.

```
SELECT EMP_ID
FROM EMPLOYEE_TBL
WHERE CITY = 'INDIANAPOLIS';
```

```
EMP_ID
-----
442346889
313782439
220984332
443679012
```

4 rows selected.

## HOOR 14: Using Subqueries to Define Unknown Data

The first query is used as the subquery in the following UPDATE statement. The first query proves how many employee identifications are returned by the subquery. The following is the UPDATE with the subquery:

```
UPDATE EMPLOYEE_PAY_TBL
SET PAY_RATE = PAY_RATE * 1.1
WHERE EMP_ID IN (SELECT EMP_ID
                 FROM EMPLOYEE_TBL
                 WHERE CITY = 'INDIANAPOLIS');
```

4 rows updated.

As expected, four rows are updated. One very important thing to notice is that, unlike the example in the first section, this subquery returns multiple rows of data. Because you expect multiple rows to be returned, you have used the IN operator instead of the equal sign. Remember that IN is used to compare an expression to values in a list. If the equal sign were used, an error would have been returned.

### Watch Out!

Be sure to use the correct operator when evaluating a subquery. For example, an operator used to compare an expression to one value, such as the equal sign, cannot be used to evaluate a subquery that returns more than one row of data.

## Subqueries with the DELETE Statement

The subquery also can be used in conjunction with the DELETE statement. The basic syntax is as follows:

```
DELETE FROM TABLE_NAME
[ WHERE OPERATOR [ VALUE ]
  (SELECT COLUMN_NAME
   FROM TABLE_NAME)
[ WHERE ) ]
```

In this example, you delete the BRANDON GLASS record from the EMPLOYEE\_PAY\_TBL table. You do not know Brandon's employee identification number, but you can use a subquery to get his identification number from the EMPLOYEE\_TBL table, which contains the FIRST\_NAME and LAST\_NAME columns.

```
DELETE FROM EMPLOYEE_PAY_TBL
WHERE EMP_ID = (SELECT EMP_ID
               FROM EMPLOYEE_TBL
               WHERE LAST_NAME = 'GLASS'
                 AND FIRST_NAME = 'BRANDON');
```

1 row deleted.

Do not forget the use of the WHERE clause with the UPDATE and DELETE statements. All rows are updated or deleted from the target table if the WHERE clause is not used. See Hour 5, "Manipulating Data."

**Watch  
Out!**

## Embedded Subqueries

A subquery can be embedded within another subquery, just as you can embed the subquery within a regular query. When a subquery is used, that subquery is resolved before the main query. Likewise, the lowest level subquery is resolved first in embedded or nested subqueries, working out to the main query.

You must check your particular implementation for limits on the number of subqueries, if any, that can be used in a single statement. It may differ between vendors.

**By the  
Way**

The basic syntax for embedded subqueries is as follows:

```
SELECT COLUMN_NAME [, COLUMN_NAME ]
FROM TABLE1 [, TABLE2 ]
WHERE COLUMN_NAME OPERATOR (SELECT COLUMN_NAME
                             FROM TABLE
                             WHERE COLUMN_NAME OPERATOR
                             (SELECT COLUMN_NAME
                              FROM TABLE
                              [ WHERE COLUMN_NAME OPERATOR VALUE ]))
```

The following example uses two subqueries, one embedded within the other. You want to find out what customers have placed orders where the quantity multiplied by the cost of a single order is greater than the sum of the cost of all products.

```
SELECT CUST_ID, CUST_NAME
FROM CUSTOMER_TBL
WHERE CUST_ID IN (SELECT O.CUST_ID
                 FROM ORDERS_TBL O, PRODUCTS_TBL P
                 WHERE O.PROD_ID = P.PROD_ID
                 AND O.QTY + P.COST < (SELECT SUM(COST)
                                       FROM
                                       PRODUCTS_TBL));
```

CUST_ID	CUST_NAME
-----	-----
090	WENDY WOLF
232	LESLIE GLEASON
287	GAVINS PLACE
43	SCHYLERS NOVELTIES
432	SCOTTYS MARKET
560	ANDYS CANDIES

6 rows selected.

**HOURL 14: Using Subqueries to Define Unknown Data**

Six rows that met the criteria of both subqueries were selected.

The following two examples show the results of each of the subqueries to aid your understanding of how the main query was resolved.

```
SELECT SUM(COST) FROM PRODUCTS_TBL;
```

```
SUM(COST)
-----
138.08
```

1 row selected.

```
SELECT O.CUST_ID
FROM ORDERS_TBL O, PRODUCTS_TBL P
WHERE O.PROD_ID = P.PROD_ID
      AND O.QTY + P.COST > 138.08;
```

```
CUST_ID
-----
43
287
```

2 rows selected.

In essence, the main query, after the substitution of the second subquery, is evaluated as shown in the following example:

```
SELECT CUST_ID, CUST_NAME
FROM CUSTOMER_TBL
WHERE CUST_ID IN (SELECT O.CUST_ID
                  FROM ORDERS_TBL O, PRODUCTS_TBL P
                  WHERE O.PROD_ID = P.PROD_ID
                        AND O.QTY + P.COST > 138.08);
```

The following shows how the main query is evaluated after the substitution of the first subquery:

```
SELECT CUST_ID, CUST_NAME
FROM CUSTOMER_TBL
WHERE CUST_ID IN ('287', '43');
```

The following is the final result:

```
CUST_ID      CUST_NAME
-----
43           SCHYLERS NOVELTIES
287          GAVINS PLACE
```

2 rows selected.

The use of multiple subqueries results in slower response time and might result in reduced accuracy of the results due to possible mistakes in the statement coding.

**Watch  
Out!**

## Correlated Subqueries

Correlated subqueries are common in many SQL implementations. The concept of correlated subqueries is discussed as an ANSI-standard SQL topic and is covered briefly in this hour. A *correlated subquery* is a subquery that is dependent upon information in the main query. This means that tables in a subquery can be related to tables in the main query.

In the following example, the table join between CUSTOMER\_TBL and ORDERS\_TBL in the subquery is dependent on the alias for CUSTOMER\_TBL (C) in the main query. This query returns the name of all customers that have ordered more than 10 units of one or more items.

```
SELECT C.CUST_NAME
FROM CUSTOMER_TBL C
WHERE 10 < (SELECT SUM(O.QTY)
           FROM ORDERS_TBL O
           WHERE O.CUST_ID = C.CUST_ID);
```

CUST\_NAME

-----

SCOTTYS MARKET  
SCHYLERS NOVELTIES  
MARYS GIFT SHOP

3 rows selected.

In the case of a correlated subquery, the reference to the table in the main query must be accomplished before the subquery can be resolved.

**By the  
Way**

The subquery is slightly modified in the next statement to show you the total quantity of units ordered for each customer, allowing the previous results to be verified.

```
SELECT C.CUST_NAME, SUM(O.QTY)
FROM CUSTOMER_TBL C,
     ORDERS_TBL O
WHERE C.CUST_ID = O.CUST_ID
GROUP BY C.CUST_NAME;
```

CUST_NAME	SUM(O.QTY)
-----	-----
ANDYS CANDIES	1
GAVINS PLACE	10
LESLIE GLEASON	1
MARYS GIFT SHOP	100
SCHYLERS NOVELTIES	25
SCOTTYS MARKET	20
WENDY WOLF	2

7 rows selected.

The `GROUP BY` clause in this example is required because another column is being selected with the aggregate function `SUM`. This gives you a sum for each customer. In the original subquery, a `GROUP BY` clause is not required because `SUM` is used to achieve a total for the entire query, which is run against the record for each individual customer.

## Summary

By simple definition and general concept, a subquery is a query that is performed within another query to place further conditions on a query. A subquery can be used in a SQL statement's `WHERE` clause or `HAVING` clause. Queries are typically used within other queries (Data Query Language), but can also be used in the resolution of DML statements such as `INSERT`, `UPDATE`, and `DELETE`. All basic rules for DML apply when using subqueries with DML commands.

The subquery's syntax is virtually the same as that of a standalone query, with a few minor restrictions. One of these restrictions is that the `ORDER BY` clause cannot be used within a subquery; a `GROUP BY` clause can be used, however, which renders virtually the same effect. Subqueries are used to place conditions that are not necessarily known for a query, providing more power and flexibility with SQL.

## Q&A

- Q.** *In the examples of subqueries, I noticed quite a bit of indentation. Is this necessary in the syntax of a subquery?*
- A.** Absolutely not. The indentation is used merely to break the statement into separate parts, making the statement more readable and easier to follow.
- Q.** *Is there a limit on the number of embedded subqueries that can be used in a single query?*
- A.** Limitations such as the number of embedded subqueries allowed and the number of tables joined in a query are specific to each implementation. Some implementations might not have limits, although the use of too many embedded subqueries could drastically hinder SQL statement performance. Most limitations are affected by the actual hardware, CPU speed, and system memory available, although there are many other considerations.
- Q.** *It seems that debugging a query with subqueries can prove to be very confusing, especially with embedded subqueries. What is the best way to debug a query with subqueries?*
- A.** The best way to debug a query with subqueries is to evaluate the query in sections. First, evaluate the lowest-level subquery, and then work your way to the main query (the same way the database evaluates the query). When you evaluate each subquery individually, you can substitute the returned values for each subquery to check your main query's logic. An error with a subquery is often in the use of the operator used to evaluate the subquery, such as (=), IN, >, <, and so on.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, "Answers to Quizzes and Exercises," for answers.

### Quiz

1. What is the function of a subquery when used with a SELECT statement?
2. Can you update more than one column when using the UPDATE statement in conjunction with a subquery?
3. Are the following syntax(s) correct? If not, what is the correct syntax?

**A.**

```
SELECT CUST_ID, CUST_NAME
      FROM CUSTOMER_TBL
      WHERE CUST_ID =
              (SELECT CUST_ID
               FROM ORDERS_TBL
               WHERE ORD_NUM = '16C17');
```

**B.**

```
SELECT EMP_ID, SALARY
      FROM EMPLOYEE_PAY_TBL
      WHERE SALARY BETWEEN '20000'
              AND (SELECT SALARY
                  FROM EMPLOYEE_ID
                  WHERE SALARY = '40000');
```

**C.**

```
UPDATE PRODUCTS_TBL
      SET COST = 1.15
      WHERE CUST_ID =
              (SELECT CUST_ID
               FROM ORDERS_TBL
               WHERE ORD_NUM = '32A132');
```

4. What would happen if the following statement were run?

```
DELETE FROM EMPLOYEE_TBL
      WHERE EMP_ID IN
              (SELECT EMP_ID
               FROM EMPLOYEE_PAY_TBL);
```

### Exercises

1. Write the MySQL SQL code for the requested subqueries by hand on a sheet of paper and compare your results to ours. Use the following tables to complete the exercises:

EMPLOYEE_TBL			
EMP_ID	VARCHAR(9)	NOT NULL	primary key
LAST_NAME	VARCHAR(15)	NOT NULL	
FIRST_NAME	VARCHAR(15)	NOT NULL	
MIDDLE_NAME	VARCHAR(15)		
ADDRESS	VARCHAR(30)	NOT NULL	

```

CITY          VARCHAR(15)   NOT NULL
STATE        VARCHAR(2)    NOT NULL
ZIP          INTEGER(5)   NOT NULL
PHONE       VARCHAR(10)
PAGER       VARCHAR(10)

EMPLOYEE_PAY_TBL
EMP_ID       VARCHAR(9)    NOT NULL      primary key
POSITION    VARCHAR(15)   NOT NULL
DATE_HIRE   DATETIME
PAY_RATE    DECIMAL(4,2)  NOT NULL
DATE_LAST_RAISE DATETIME
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID_ REFERENCES
EMPLOYEE_TBL (EMP_ID)

CUSTOMER_TBL
CUST_ID     VARCHAR(10)   NOT NULL      primary key
CUST_NAME   VARCHAR(30)   NOT NULL
CUST_ADDRESS VARCHAR(20)  NOT NULL
CUST_CITY   VARCHAR(15)   NOT NULL
CUST_STATE  VARCHAR(2)    NOT NULL
CUST_ZIP    INTEGER(5)   NOT NULL
CUST_PHONE  INTEGER(10)
CUST_FAX    INTEGER(10)

ORDERS_TBL
ORD_NUM     VARCHAR(10)   NOT NULL      primary key
CUST_ID     VARCHAR(10)   NOT NULL
PROD_ID     VARCHAR(10)   NOT NULL
QTY         INTEGER(6)   NOT NULL
ORD_DATE    DATETIME

PRODUCTS_TBL
PROD_ID     VARCHAR(10)   NOT NULL      primary key
PROD_DESC   VARCHAR(40)   NOT NULL
COST        DECIMAL(6,2)  NOT NULL

```

2. Using a subquery, write an SQL statement to update the CUSTOMER\_TBL table. Find the customer with the order number 23E934 and change the customer name to DAVIDS MARKET.
3. Using a subquery, write a query that returns all the names of all employees who have a pay rate greater than JOHN DOE, whose employee identification number is 343559876.
4. Using a subquery, write a query that lists all products that cost more than the average cost of all products.

*This page intentionally left blank*

## HOOR 15

# Combining Multiple Queries into One

During this hour, you learn how to combine SQL queries into one by using the UNION, UNION ALL, INTERSECT, and EXCEPT operators. Once again, you must check your particular implementation for any variations in the use of these operators.

---

### ***The highlights of this hour include***

- ▶ An overview of the operators used to combine queries
- ▶ When to use the commands to combine queries
- ▶ Using the GROUP BY clause with the compound operators
- ▶ Using the ORDER BY clause with the compound operators
- ▶ How to retrieve accurate data

Some of the query operators covered in this hour are not currently supported by MySQL, as of the current release of version 5.0.45.

***By the  
Way***

## **Single Queries Versus Compound Queries**

The single query is one SELECT statement, whereas the compound query includes two or more SELECT statements.

Compound queries are formed by using some type of operator to join the two queries. The UNION operator in the following examples is used to join two queries.

## HOOR 15: Combining Multiple Queries into One

A single SQL statement could be written as follows:

```
SELECT EMP_ID, SALARY, PAY_RATE
FROM EMPLOYEE_PAY_TBL
WHERE SALARY IS NOT NULL OR
PAY_RATE IS NOT NULL;
```

This is the same statement using the UNION operator:

```
SELECT EMP_ID, SALARY
FROM EMPLOYEE_PAY_TBL
WHERE SALARY IS NOT NULL
UNION
SELECT EMP_ID, PAY_RATE
FROM EMPLOYEE_PAY_TBL
WHERE PAY_RATE IS NOT NULL;
```

The previous statements return pay information for all employees who are paid either hourly or on a salary.

### **By the Way**

If you executed the second query, the output has two column headings: EMP\_ID and SALARY. Each individual's pay rate is listed under the SALARY column. When using the UNION operator, column headings are determined by column names or column aliases used in the first SELECT statement.

Compound operators are used to combine and restrict the results of two SELECT statements. These operators can be used to return or suppress the output of duplicate records. Compound operators can bring together similar data that is stored in different fields.

Compound queries allow you to combine the results of more than one query to return a single set of data. Compound queries are often simpler to write than a single query with complex conditions. Compound queries also allow for more flexibility regarding the never-ending task of data retrieval.

## Compound Query Operators

The compound query operators vary among database vendors. The ANSI standard includes the UNION, UNION ALL, EXCEPT, and INTERSECT operators, all of which are discussed in the following sections.

## The UNION Operator

The UNION operator is used to combine the results of two or more SELECT statements without returning any duplicate rows. In other words, if a row of output exists in the results of one query, the same row is not returned, even though it exists in the second query. To use the UNION operator, each SELECT statement must have the same number of columns selected, the same number of column expressions, the same data type, and the same order—but they do not have to be the same length.

The syntax is as follows:

```
SELECT COLUMN1 [ , COLUMN2 ]
FROM TABLE1 [ , TABLE2 ]
[ WHERE ]
UNION
SELECT COLUMN1 [ , COLUMN2 ]
FROM TABLE1 [ , TABLE2 ]
[ WHERE ]
```

Look at the following example:

```
SELECT EMP_ID FROM EMPLOYEE_TBL
UNION
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL;
```

Those employee IDs that are in both tables appear only once in the results.

This hour's examples begin with a simple SELECT statement from two tables:

```
SELECT PROD_DESC FROM PRODUCTS_TBL;
```

```
PROD_DESC
-----
WITCHES COSTUME
PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
ASSORTED COSTUMES
CANDY CORN
PUMPKIN CANDY
PLASTIC SPIDERS
ASSORTED MASKS
KEY CHAIN
OAK BOOKSHELF
```

1 rows selected.

```
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
-----
WITCHES COSTUME
PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
```

```

ASSORTED COSTUMES
CANDY CORN
PUMPKIN CANDY
PLASTIC SPIDERS
ASSORTED MASKS
KEY CHAIN
OAK BOOKSHELF

```

11 rows selected.

**By the  
Way**

The PRODUCTS\_TMP table was created in Hour 3, “Managing Database Objects.” Refer back to Hour 3 if you need to re-create this table.

Now, combine the same two queries with the UNION operator, making a compound query.

```

SELECT PROD_DESC FROM PRODUCTS_TBL
UNION
SELECT PROD_DESC FROM PRODUCTS_TMP;

```

```

PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
CANDY CORN
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY
WITCHES COSTUME
KEY CHAIN
OAK BOOKSHELF

```

11 rows selected.

In the first query, nine rows of data were returned, and six rows of data were returned from the second query. Nine rows of data are returned when the UNION operator combines the two queries. Only nine rows are returned because duplicate rows of data are not returned when using the UNION operator.

The following code shows an example of combining two unrelated queries with the UNION operator:

```

SELECT PROD_DESC FROM PRODUCTS_TBL
UNION
SELECT LAST_NAME FROM EMPLOYEE_TBL;

```

```
PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
CANDY CORN
FALSE PARAFFIN TEETH
GLASS
KEY CHAIN
LIGHTED LANTERNS
OAK BOOKSHELF
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PLEW
PUMPKIN CANDY
SPURGEON
STEPHENS
WALLACE
WITCHES COSTUME

16 rows selected.
```

The PROD\_DESC and LAST\_NAME values are listed together, and the column heading is taken from the column name in the first query.

## The UNION ALL Operator

The UNION ALL operator is used to combine the results of two SELECT statements, including duplicate rows. The same rules that apply to UNION apply to the UNION ALL operator. The UNION and UNION ALL operators are the same, although one returns duplicate rows of data where the other does not.

The syntax is as follows:

```
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
UNION ALL
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
```

Look at the following example:

```
SELECT EMP_ID FROM EMPLOYEE_TBL
UNION ALL
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
```

The preceding SQL statement returns all employee IDs from both tables and shows duplicates.

The following is the same compound query in the previous section with the UNION ALL operator:

```
SELECT PROD_DESC FROM PRODUCTS_TBL
UNION ALL
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
-----
WITCHES COSTUME
PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
ASSORTED COSTUMES
CANDY CORN
PUMPKIN CANDY
PLASTIC SPIDERS
ASSORTED MASKS
KEY CHAIN
OAK BOOKSHELF
WITCHES COSTUME
PLASTIC PUMPKIN 18 INCH
FALSE PARAFFIN TEETH
LIGHTED LANTERNS
ASSORTED COSTUMES
CANDY CORN
PUMPKIN CANDY
PLASTIC SPIDERS
ASSORTED MASKS
KEY CHAIN
OAK BOOKSHELF
```

22 rows selected.

Notice that there were 22 rows returned in this query (9+6) because duplicate records are retrieved with the UNION ALL operator.

## The INTERSECT Operator

The INTERSECT operator is used to combine two SELECT statements, but returns only rows from the first SELECT statement that are identical to a row in the second SELECT statement. Just as with the UNION operator, the same rules apply when using the INTERSECT operator. Currently, the INTERSECT operator is not supported by MySQL.

The syntax is as follows:

```
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
INTERSECT
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
```

Look at the following example:

```
SELECT CUST_ID FROM CUSTOMER_TBL
INTERSECT
SELECT CUST_ID FROM ORDERS_TBL;
```

The preceding SQL statement returns the customer identification for those customers who have placed an order.

The following example illustrates the INTERSECT operator using the two original queries in this hour:

```
SELECT PROD_DESC FROM PRODUCTS_TBL
INTERSECT
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
CANDY CORN
FALSE PARAFFIN TEETH
KEY CHAIN
LIGHTED LANTERNS
OAK BOOKSHELF
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY
WITCHES COSTUME
```

11 rows selected.

Only eleven rows are returned because only eleven rows were identical between the output of the two single queries.

## The EXCEPT Operator

The EXCEPT operator combines two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement. Once again, the same rules that apply to the UNION operator also apply to the EXCEPT operator. The EXCEPT operator is not currently supported in MySQL.

The syntax is as follows:

```
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
EXCEPT
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
```

Study the following example:

```
SELECT PROD_DESC FROM PRODUCTS_TBL
EXCEPT
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
-----
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY
```

3 rows selected.

According to the results, there were three rows of data returned by the first query that were not returned by the second query.

The EXCEPT operator is known as the MINUS operator in some implementations. Check your implementation for the operator name that performs the EXCEPT operator's function.

The following example demonstrates the use of the MINUS operator as a replacement for the EXCEPT operator.

```
SELECT PROD_DESC FROM PRODUCTS_TBL
MINUS
SELECT PROD_DESC FROM PRODUCTS_TMP;
```

```
PROD_DESC
-----
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY
```

3 rows selected.

## Using ORDER BY with a Compound Query

The ORDER BY clause can be used with a compound query. However, the ORDER BY clause can only be used to order the results of both queries. Therefore, there can be only one ORDER BY clause in a compound query, even though the compound query might consist of multiple individual queries or SELECT statements. The ORDER BY clause must reference the columns being ordered by an alias or by the column number.

The syntax is as follows:

```
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
```

```

OPERATOR{UNION | EXCEPT | INTERSECT | UNION ALL}
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
[ ORDER BY ]

```

Examine the following example:

```

SELECT EMP_ID FROM EMPLOYEE_TBL
UNION
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
ORDER BY 1;

```

The results of the compound query are sorted by the first column of each individual query. Duplicate records can easily be recognized by sorting compound queries.

The column in the ORDER BY clause is referenced by the number 1 instead of the actual column name.

**By the  
Way**

The preceding SQL statement returns the employee ID from the EMPLOYEE\_TBL and the EMPLOYEE\_PAY\_TBL, but it does not show duplicates and it orders by the employee ID.

The following example shows the use of the ORDER BY clause with a compound query. The column name can be used in the ORDER BY clause if the column sorted by has the same name in all individual queries of the statement.

```

SELECT PROD_DESC FROM PRODUCTS_TBL
UNION
SELECT PROD_DESC FROM PRODUCTS_TBL
ORDER BY PROD_DESC;

```

```

PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
CANDY CORN
FALSE PARAFFIN TEETH
KEY CHAIN
LIGHTED LANTERNS
OAK BOOKSHELF
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY
WITCHES COSTUME

```

11 rows selected.

The following query uses a numeric value in place of the actual column name in the ORDER BY clause:

```
SELECT PROD_DESC FROM PRODUCTS_TBL
UNION
SELECT PROD_DESC FROM PRODUCTS_TBL;
```

```
PROD_DESC
-----
ASSORTED COSTUMES
ASSORTED MASKS
CANDY CORN
FALSE PARAFFIN TEETH
KEY CHAIN
LIGHTED LANTERNS
OAK BOOKSHELF
PLASTIC PUMPKIN 18 INCH
PLASTIC SPIDERS
PUMPKIN CANDY
WITCHES COSTUME
```

11 rows selected.

## Using GROUP BY with a Compound Query

Unlike ORDER BY, GROUP BY can be used in each SELECT statement of a compound query, but it also can be used following all individual queries. In addition, the HAVING clause (sometimes used with the GROUP BY clause) can be used in each SELECT statement of a compound statement.

The syntax is as follows:

```
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
[ GROUP BY ]
[ HAVING ]
OPERATOR {UNION | EXCEPT | INTERSECT | UNION ALL}
SELECT COLUMN1 [, COLUMN2 ]
FROM TABLE1 [, TABLE2 ]
[ WHERE ]
[ GROUP BY ]
[ HAVING ]
[ ORDER BY ]
```

The compound query operators covered in this hour are not currently supported in MySQL.

In the following example, you select a literal string to represent customer records, employee records, and product records. Each individual query is simply a count of all records in each appropriate table. The GROUP BY clause is used to group the results of the entire report by the numeric value 1, which represents the first column in each individual query.

```
SELECT 'CUSTOMERS' TYPE, COUNT(*)
FROM CUSTOMER_TBL
UNION
SELECT 'EMPLOYEES' TYPE, COUNT(*)
FROM EMPLOYEE_TBL
UNION
SELECT 'PRODUCTS' TYPE, COUNT(*)
FROM PRODUCTS_TBL
GROUP BY 1;
```

TYPE	COUNT(*)
CUSTOMERS	15
EMPLOYEES	6
PRODUCTS	9

3 rows selected.

The following query is identical to the previous query, except that the ORDER BY clause is used as well:

```
SELECT 'CUSTOMERS' TYPE, COUNT(*)
FROM CUSTOMER_TBL
UNION
SELECT 'EMPLOYEES' TYPE, COUNT(*)
FROM EMPLOYEE_TBL
UNION
SELECT 'PRODUCTS' TYPE, COUNT(*)
FROM PRODUCTS_TBL
GROUP BY 1
ORDER BY 2;
```

TYPE	COUNT(*)
EMPLOYEES	6
PRODUCTS	9
CUSTOMERS	15

3 rows selected.

This is sorted by column 2, which was the count on each table. Hence, the final output is sorted by the count from least to greatest.

## Retrieving Accurate Data

Be cautious when using the compound operators. Incorrect or incomplete data might be returned if you were using the `INTERSECT` operator and you used the wrong `SELECT` statement as the first individual query. In addition, consider whether duplicate records are wanted when using the `UNION` and `UNION ALL` operators. What about `EXCEPT`? Do you need any of the rows that were not returned by the second query? As you can see, the wrong compound query operator or the wrong order of individual queries in a compound query can easily cause misleading data to be returned.

**By the  
Way**

Incomplete data returned by a query qualifies as incorrect data.

## Summary

You have been introduced to compound queries. All SQL statements previous to this hour have consisted of a single query. Compound queries allow multiple individual queries to be used together as a single query to achieve the data resultset desired as output. The compound query operators discussed included `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT (MINUS)`. `UNION` returns the output of two single queries without displaying duplicate rows of data. `UNION ALL` simply displays all output of single queries, regardless of existing duplicate rows. `INTERSECT` is used to return identical rows between two queries. `EXCEPT` (the same as `MINUS`) is used to return the results of one query that do not exist in another query. Compound queries provide greater flexibility when trying to satisfy the requirements of various queries, which, without the use of compound operators, could result in very complex queries.

## Q&A

- Q.** *How are the columns referenced in the `GROUP BY` clause in a compound query?*
- A.** The columns can be referenced by the actual column name or by the number of the column placement in the query if the column names are not identical in the two queries.

**Q.** *I understand what the EXCEPT operator does, but would the outcome change if I were to reverse the SELECT statements?*

**A.** Yes. The order of the individual queries is very important when using the EXCEPT or MINUS operator. Remember that all rows are returned from the first query that are not returned by the second query. Changing the order of the two individual queries in the compound query could definitely affect the results.

**Q.** *Must the data type and the length of columns in a compound query be the same in both queries?*

**A.** No. Only the data type must be the same. The length can differ.

**Q.** *What determines the column names when using the UNION operator?*

**A.** The first query set determines the column names for the data returned when using a UNION operator.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

Refer to the Oracle syntax covered in this hour for the following quiz questions when referring to the INTERSECT and EXCEPT operators.

1. Is the syntax correct for the following compound queries? If not, what would correct the syntax? Use the EMPLOYEE\_TBL and the EMPLOYEE\_PAY\_TBL shown as follows:

```
EMPLOYEE_TBL
EMP_ID      VARCHAR(9)      NOT NULL,
LAST_NAME   VARCHAR(15)   NOT NULL,
FIRST_NAME  VARCHAR(15)   NOT NULL,
MIDDLE_NAME VARCHAR(15),
ADDRESS     VARCHAR(30)  NOT NULL,
```

## HOURL 15: Combining Multiple Queries into One

```
CITY          VARCHAR(15)    NOT NULL,
STATE        VARCHAR(2)    NOT NULL,
ZIP          INTEGER(5)    NOT NULL,
PHONE        VARCHAR(10),
PAGER        VARCHAR(10),
CONSTRAINT EMP_PK PRIMARY KEY (EMP_ID)
```

```
EMPLOYEE_PAY_TBL
EMP_ID        VARCHAR(9)    NOT NULL    primary key,
POSITION      VARCHAR(15)  NOT NULL,
DATE_HIRE     DATETIME,
PAY_RATE      DECIMAL(4,2) NOT NULL,
DATE_LAST_RAISE DATE,
SALARY        DECIMAL(8,2),
BONUS         DECIMAL(6,2),
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID)
REFERENCES EMPLOYEE_TBL (EMP_ID)
```

### A.

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
UNION
SELECT EMP_ID, POSITION, DATE_HIRE
FROM EMPLOYEE_PAY_TBL;
```

### B.

```
SELECT EMP_ID FROM EMPLOYEE_TBL
UNION ALL
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
ORDER BY EMP_ID;
```

### C.

```
SELECT EMP_ID FROM EMPLOYEE_PAY_TBL
INTERSECT
SELECT EMP_ID FROM EMPLOYEE_TBL
ORDER BY 1;
```

2. Match the correct operator to the following statements.

Statement	Operator
a. Show duplicates	UNION
b. Return only rows from the first query that match those in the second query	INTERSECT
c. Return no duplicates	UNION ALL
d. Return only rows from the first query not returned by the second	EXCEPT

## Exercises

Refer to the Oracle syntax covered in this hour for the following exercises. Write your queries out by hand on a sheet of paper because MySQL does not support some of the operators covered in this hour. When you are finished, compare your results to ours.

Use the CUSTOMER\_TBL and the ORDERS\_TBL as listed:

```
CUSTOMER_TBL
CUST_IN      VARCHAR(10)   NOT NULL      primary key,
CUST_NAME    VARCHAR(30)   NOT NULL,
CUST_ADDRESS VARCHAR(20)   NOT NULL,
CUST_CITY    VARCHAR(15)   NOT NULL,
CUST_STATE   VARCHAR(2)     NOT NULL,
CUST_ZIP     INTEGER(5)    NOT NULL,
CUST_PHONE   INTEGER(10),
CUST_FAX     INTEGER(10)

ORDERS_TBL
ORD_NUM      VARCHAR(10)   NOT NULL      primary key,
CUST_ID      VARCHAR(10)   NOT NULL,
PROD_ID      VARCHAR(10)   NOT NULL,
QTY          INTEGER(6)    NOT NULL,
ORD_DATE     DATETIME
```

1. Write a compound query to find the customers that have placed an order.
2. Write a compound query to find the customers that have not placed an order.

*This page intentionally left blank*

## **PART V**

# **SQL Performance Tuning**

<b>HOUR 16</b>	Using Indexes to Improve Performance	<b>253</b>
<b>HOUR 17</b>	Improving Database Performance	<b>265</b>

*This page intentionally left blank*

## HOUR 16

# Using Indexes to Improve Performance

During this hour, you learn how to improve SQL statement performance by creating and using indexes. You begin with the `CREATE INDEX` command and learn how to use indexes that have been created on tables.

---

### ***The highlights of this hour include***

- ▶ How to create an index
- ▶ How indexes work
- ▶ The different types of indexes
- ▶ When to use indexes
- ▶ When not to use indexes

## **What Is an Index?**

Simply put, an *index* is a pointer to data in a table. An index in a database is very similar to an index in the back of a book. For example, if you want to reference all pages in a book that discuss a certain topic, you first refer to the index, which lists all topics alphabetically, and it refers you to one or more specific page numbers. An index in a database works the same way in that a query is pointed to the exact physical location of data in a table. You are actually being directed to the data's location in an underlying file of the database, but as far as you are concerned, you are referring to a table.

Which would be faster, looking through a book page by page for some information or searching the book's index and getting a page number? Of course, using the book's index is the most efficient method. A lot of time can be saved if that book is large. Say you have a small book of just a few pages. In this case, it might be faster to check the pages for the

information than to flip back and forth between the index and pages of the book. When a database does not use an index, it is performing what is typically called a *full table scan*, the same as flipping through a book page by page. Full table scans are discussed in Hour 17, “Improving Database Performance.”

An index is typically stored separately from the table for which the index was created. An index’s main purpose is to improve the performance of data retrieval. Indexes can be created or dropped with no effect on the data. However, after an index is dropped, performance of data retrieval might be slowed. Indexes do take up physical space and can often grows larger than the table itself. Therefore, they need to be considered when estimating the size your database storage needs.

## How Do Indexes Work?

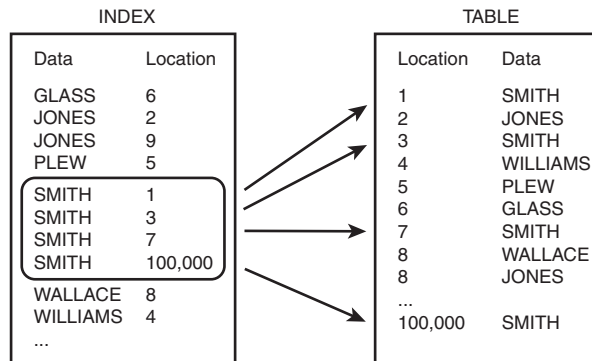
When an index is created, it records the location of values in a table that are associated with the column that is indexed. Entries are added to the index when new data is added to the table. When a query is executed against the database and a condition is specified on a column in the WHERE clause that is indexed, the index is first searched for the values specified in the WHERE clause. If the value is found in the index, the index returns the exact location of the searched data in the table.

Figure 16.1 illustrates how an index functions.

Suppose the following query was issued:

```
SELECT *
FROM TABLE_NAME
WHERE NAME = 'SMITH';
```

**FIGURE 16.1**  
Table access using an index.



As shown in Figure 16.1, the NAME index is referenced to resolve the location of all names equal to 'SMITH'. After the location is determined, the data can be quickly retrieved from the table. The data, in this case names, is alphabetized in the index.

A full table scan would occur if there were no index on the table and the same query was executed, which means that every row of data in the table would be read to retrieve information pertaining to all individuals with the name SMITH.

An index is faster because it typically stores information in an orderly tree-like format. Consider if we had a list of books upon which we placed an index. The index would have a root node, which would be the beginning point of each query. Then it would be split into branches. Maybe in our case there are two branches, one for letters *A–L* and the other for letters *M–Z*. Now if you ask for a book with a name that starts with the letter *M*, you will enter the index at the root node and immediately travel to the branch containing letters *M–Z*. This would effectively cut your time to find the book by eliminating close to half the possibilities.

## The CREATE INDEX Command

The CREATE INDEX statement, as with many other statements in SQL, varies greatly among different relational database vendors. Most relational database implementations use the CREATE INDEX statement:

```
CREATE INDEX INDEX_NAME ON TABLE_NAME
```

The syntax is where the vendors start varying greatly on the CREATE INDEX statement options. Some implementations allow the specification of a storage clause (as with the CREATE TABLE statement), ordering (DESC|ASC), and the use of clusters. You must check your particular implementation for its correct syntax.

Indexes can be created during table creation in some implementations. Most implementations accommodate a command, aside from the CREATE TABLE command, used to create indexes. You must check your particular implementation for the exact syntax for the command, if any, that is available to create an index.

**By the  
Way**

## Types of Indexes

Different types of indexes can be created on tables in a database, all of which serve the same goal—to improve database performance by expediting data retrieval. This hour discusses single-column indexes, composite indexes, and unique indexes.

## Single-Column Indexes

Indexing on a single column of a table is the simplest and most common manifestation of an index. Obviously, a *single-column index* is one that is created based on only one table column. The basic syntax is as follows:

```
CREATE INDEX INDEX_NAME  
ON TABLE_NAME (COLUMN_NAME)
```

For example, if you want to create an index on the EMPLOYEE\_TBL table for employees' last names, the command used to create the index would look like the following:

```
CREATE INDEX NAME_IDX  
ON EMPLOYEE_TBL (LAST_NAME);
```

### **Did you Know?**

Single-column indexes are most effective when used on columns that are frequently used alone in the WHERE clause as query conditions. Good candidates for a single-column index are an individual identification number, a serial number, or a system-assigned key.

## Unique Indexes

*Unique indexes* are used for performance and data integrity. A unique index does not allow any duplicate values to be inserted into the table. Otherwise, the unique index performs the same way a regular index performs. The syntax is as follows:

```
CREATE UNIQUE INDEX INDEX_NAME  
ON TABLE_NAME (COLUMN_NAME)
```

If you want to create a unique index on the EMPLOYEE\_TBL table for an employee's last name, the command used to create the unique index would look like the following:

```
CREATE UNIQUE INDEX NAME_IDX  
ON EMPLOYEE_TBL (LAST_NAME);
```

The only problem with this index is that every individual's last name in the EMPLOYEE\_TBL table must be unique—pretty impractical. However, a unique index should be created for a column, such as an individual's Social Security number, because each of these numbers for each individual is unique.

You might be wondering, "What if an employee's SSN were the primary key for a table?" An index is usually implicitly created when you define a primary key for a table. However, a company can use a fictitious number for an employee ID, but

maintain each employee's SSN for tax purposes. You probably want to index this column and ensure that all entries into this column are unique values.

A unique index can only be created on a column in a table whose values are unique. In other words, you cannot create a unique index on an existing table with data that already contains records on the indexed key.

**By the  
Way**

## Composite Indexes

A *composite index* is an index on two or more columns of a table. You should consider performance when creating a composite index because the order of columns in the index has a measurable effect on data retrieval speed. Generally, the most restrictive value should be placed first for optimum performance. However, the columns that will always be specified should be placed first. The syntax is as follows:

```
CREATE INDEX INDEX_NAME  
ON TABLE_NAME (COLUMN1, COLUMN2)
```

An example of a composite index follows:

```
CREATE INDEX ORD_IDX  
ON ORDERS_TBL (CUST_ID, PROD_ID);
```

In this example, you create a composite index based on two columns in the `ORDERS_TBL` table: `CUST_ID` and `PROD_ID`. You assume that these two columns are frequently used together as conditions in the `WHERE` clause of a query.

Composite indexes are most effective on table columns that are used together frequently as conditions in a query's `WHERE` clause.

**Did you  
Know?**

In deciding whether to create a single-column index or a composite index, take into consideration the column(s) that you might use very frequently in a query's `WHERE` clause as filter conditions. If only one column is used, a single-column index should be the choice. If two or more columns are frequently used in the `WHERE` clause as filters, a composite index would be the best choice.

## Implicit Indexes

*Implicit indexes* are indexes that are automatically created by the database server when an object is created. Indexes are automatically created for primary key constraints and unique constraints.

Why are indexes automatically created for these constraints? Imagine that you are the database server. A user adds a new product to the database. The product identification is the primary key on the table, which means that it must be a unique value. To efficiently check to make sure the new value is unique among hundreds or thousands of records, the product identifications in the table must be indexed. Therefore, when you create a primary key or unique constraint, an index is automatically created for you.

## When Should Indexes Be Considered?

Unique indexes are implicitly used in conjunction with a primary key for the primary key to work. Foreign keys are also excellent candidates for an index because they are often used to join the parent table. Most, if not all, columns used for table joins should be indexed.

Columns that are frequently referenced in the `ORDER BY` and `GROUP BY` clauses should be considered for indexes. For example, if you are sorting on an individual's name, it would be quite beneficial to have an index on the name column. It renders an automatic alphabetical order on every name, thus simplifying the actual sort operation and expediting the output results.

Furthermore, indexes should be created on columns with a high number of unique values, or columns that when used as filter conditions in the `WHERE` clause return a low percentage of rows of data from a table. This is where trial and error might come into play. Just as production code and database structures should always be tested before their implementation into production, so should indexes. This testing is time that should be spent trying different combinations of indexes, no indexes, single-column indexes, and composite indexes. There is no cut-and-dried rule for using indexes. The effective use of indexes requires a thorough knowledge of table relationships, query and transaction requirements, and the data itself.

**By the Way**

You should plan your tables and indexes. Do not assume that because an index has been created that all performance issues are resolved. The index might not help at all (it might actually hinder performance) and might just take up disk space.

## When Should Indexes Be Avoided?

Although indexes are intended to enhance a database's performance, there are times when they should be avoided. The following guidelines indicate when the use of an index should be reconsidered:

- ▶ Indexes should not be used on small tables.
- ▶ Indexes should not be used on columns that return a high percentage of data rows when used as a filter condition in a query's WHERE clause. For instance, you would not have an entry for the words *the* or *and* in the index of a book.
- ▶ Tables that have frequent, large batch update jobs run can be indexed. However, the batch job's performance is slowed considerably by the index. The conflict of having an index on a table that is frequently loaded or manipulated by a large batch process can be corrected by dropping the index before the batch job, and then re-creating the index after the job has completed. This is because the indexes are also updated as the data is inserted, causing additional overhead.
- ▶ Indexes should not be used on columns that contain a high number of NULL values.
- ▶ Columns that are frequently manipulated should not be indexed. Maintenance on the index can become excessive.

Caution should be taken when creating indexes on a table's extremely long keys because performance is inevitably slowed by high I/O costs.

**Watch  
Out!**

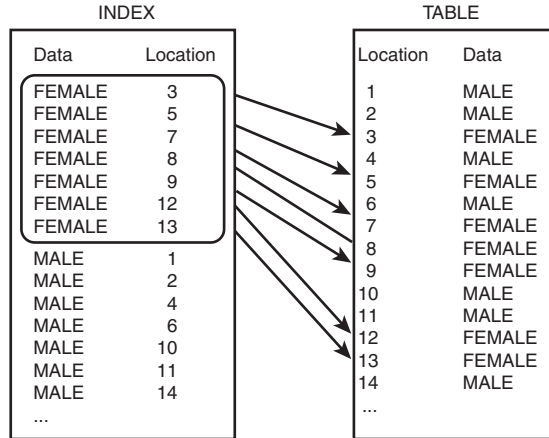
You can see in Figure 16.2 that an index on a column, such as gender, might not prove beneficial. For example, suppose the following query was submitted to the database:

```
SELECT *  
FROM TABLE_NAME  
WHERE GENDER = 'FEMALE';
```

By referring to Figure 16.2, which is based on the previous query, you can see that there is constant activity between the table and its index. Because a high number of data rows is returned for WHERE GENDER = 'FEMALE' (or 'MALE'), the database server constantly has to read the index, and then the table, and then the index, and then the table, and so on. In this case, it might be more efficient for a full table scan to occur because a high percentage of the table must be read anyway.

**FIGURE 16.2**

An example of an ineffective index.



As a general rule, you do not want to use an index on a column used in a query's condition that will return a high percentage of data rows from the table. In other words, do not create an index on a column such as gender, or any column that contains very few distinct values. This is often referred to as a column's *cardinality* or the uniqueness of the data. High-cardinality means very unique and is therefore used to describe things such as identification numbers. Low-cardinality values are not very unique and would refer to columns such as the gender example.

### **Did you Know?**

Indexes can be very good for performance, but in some cases might actually hurt performance. Refrain from creating indexes on columns that will contain few unique values, such as gender, state of residence, and so on.

## **Dropping an Index**

An index can be dropped rather simply. Check your particular implementation for the exact syntax, but most major implementations use the `DROP` command. Care should be taken when dropping an index because performance might be slowed drastically (or improved!). The syntax is as follows:

```
DROP INDEX INDEX_NAME
```

### **By the Way**

MySQL uses the `ALTER TABLE` command to drop indexes. Again, different SQL implementations might vary widely in syntax, especially when dealing with indexes and data storage.

The most common reason for dropping an index is in an attempt to improve performance. Remember that if you drop an index, you can also re-create it. Indexes might need to be rebuilt sometimes to reduce fragmentation. It is often necessary to experiment with the use of indexes in a database to determine the route to best performance, which might involve creating an index, dropping it, and eventually re-creating it, with or without modifications.

## Summary

You have learned that indexes can be used to improve the overall performance of queries and transactions performed within the database. Database indexes, like an index of a book, allow specific data to be quickly referenced from a table. The most common method for creating indexes is through use of the `CREATE INDEX` command. There are different types of indexes available among various SQL implementations. Unique indexes, single-column indexes, and composite indexes are among those different types of indexes. You need to consider many factors when deciding on the index type that best meets the needs of your database. The effective use of indexes often requires some experimentation, a thorough knowledge of table relationships and data, and a little patience—but patience now can save minutes, hours, or even days of work later.

## Q&A

**Q.** *Does an index actually take up space the way a table does?*

**A.** Yes. An index takes up physical space in a database. In fact, an index can become much larger than the table for which the index was created.

**Q.** *If you drop an index so a batch job can complete faster, how long does it take to re-create the index?*

**A.** Many factors are involved, such as the size of the index being dropped, CPU usage, and the machine's power.

**Q.** *Should all indexes be unique indexes?*

**A.** No. Unique indexes allow no duplicate values. There might be a need for the allowance of duplicate values in a table.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

### Quiz

1. What are some major disadvantages of using indexes?
2. Why is the order of columns in a composite important?
3. Should a column with a large percentage of NULL values be indexed?
4. Is the main purpose of an index to stop duplicate values in a table?
5. True or false: The main reason for a composite index is for aggregate function usage in an index.
6. What does cardinality refer to? What would be considered a column of high-cardinality?

### Exercises

1. For the following situations, decide whether an index should be used and, if so, what type of index should be used.
  - A. Several columns, but a rather small table
  - B. Medium-sized table, no duplicates should be allowed
  - C. Several columns, very large table, several columns used as filters in the WHERE clause
  - D. Large table, many columns, a lot of data manipulation

2. Type the following code into the `mysql>` prompt to create an index on the `EMPLOYEE_PAY_TBL` table on the `POSITION` column:  

```
CREATE INDEX EP_POSITION ON EMPLOYEE_PAY_TBL (POSITION);
```
3. Study the tables used in this book. What are some good candidates for indexed columns based on how a user might search for data?
4. Create a multi-column index on the `ORDERS_TBL` table. Include the following columns: `CUST_ID`, `PROD_ID`, and `ORD_DATE`.
5. Create some additional indexes on your tables as desired.

*This page intentionally left blank*

## HOUR 17

# Improving Database Performance

During this hour, you learn how to tune your SQL statement for maximum performance using some very simple methods.

---

### ***The highlights of this hour include***

- ▶ What SQL statement tuning is
- ▶ Database tuning versus SQL statement tuning
- ▶ Formatting your SQL statement
- ▶ Properly joining tables
- ▶ The most restrictive condition
- ▶ Full table scans
- ▶ Invoking the use of indexes
- ▶ Avoiding the use of OR and HAVING
- ▶ Avoiding large sort operations

## **What Is SQL Statement Tuning?**

*SQL statement tuning* is the process of optimally building SQL statements to achieve results in the most effective and efficient manner. SQL tuning begins with the basic arrangement of the elements in a query. Simple formatting can play a rather large role in the optimization of a statement.

SQL statement tuning mainly involves tweaking a statement's FROM and WHERE clauses. It is mostly from these two clauses that the database server decides how to evaluate a query. To this point, you have learned the FROM and WHERE clauses' basics. Now it is time to learn how to fine-tune them for better results and happier users.

## Database Tuning Versus SQL Statement Tuning

Before continuing with your SQL statement tuning lesson, it is important to understand the difference between tuning a database and tuning the SQL statements that access the database.

*Database tuning* is the process of tuning the actual database, which encompasses the allocated memory, disk usage, CPU, I/O, and underlying database processes. Tuning a database also involves the management and manipulation of the database structure itself, such as the design and layout of tables and indexes. Additionally, database tuning often involves the modification of the database architecture to optimize the use of the hardware resources available. Many other things need to be considered when tuning a database, but these tasks are normally accomplished by the database administrator (DBA) in conjunction with a system administrator. The objective of database tuning is to ensure that the database has been designed in a way that best accommodates expected activity within the database.

*SQL tuning* is the process of tuning the SQL statements that access the database. These SQL statements include database queries and transactional operations, such as inserts, updates, and deletes. The objective of SQL statement tuning is to formulate statements that most effectively access the database in its current state, taking advantage of database and system resources and indexes. The objective is to reduce the operational overhead of executing the query on the database.

### **By the Way**

Both database tuning and SQL statement tuning must be performed to achieve optimal results when accessing the database. A poorly tuned database might very well render your efforts in SQL tuning as wasted, and vice versa. Ideally, it is best to first tune the database, ensure that indexes exist where needed, and then tune the SQL code.

## Formatting Your SQL Statement

Formatting your SQL statement sounds like an obvious task; as obvious as it might sound, it is worth mentioning. A newcomer to SQL will probably not take into consideration several things when building a SQL statement. The upcoming sections discuss the following considerations; some are common sense, others are not so obvious:

- ▶ Formatting SQL statements for readability
- ▶ The order of tables in the FROM clause
- ▶ The placement of the most restrictive conditions in the WHERE clause
- ▶ The placement of join conditions in the WHERE clause

Most relational database implementations have what is called an *SQL optimizer*, which evaluates a SQL statement and determines the best method for executing the statement based on the way a SQL statement is written and the availability of indexes in the database. Not all optimizers are the same. Please check your implementation or consult the database administrator to learn how the optimizer reads SQL code. You should understand how the optimizer works to effectively tune a SQL statement.

**By the  
Way**

## Formatting a Statement for Readability

Formatting a SQL statement for readability is fairly obvious, but many SQL statements have not been written neatly. Although the neatness of a statement does not affect the actual performance (the database does not care how neat the statement appears), careful formatting is the first step in tuning a statement. When you look at a SQL statement with tuning intentions, making the statement readable is always the first priority. How can you determine whether the statement is written well if it is difficult to read?

Some basic rules for making a statement readable include

- ▶ Always begin a new line with each clause in the statement—For example, place the FROM clause on a separate line from the SELECT clause. Place the WHERE clause on a separate line from the FROM clause, and so on.
- ▶ Use tabs or spaces for indentation when arguments of a clause in the statement exceed one line.
- ▶ Use tabs and spaces consistently.
- ▶ Use table aliases when multiple tables are used in the statement—The use of the full table name to qualify each column in the statement quickly clutters the statement and makes reading it difficult.
- ▶ Use remarks sparingly in SQL statements if they are available within your specific implementation—Remarks are great for documentation, but too many of them clutter a statement.

**HOURL 17: Improving Database Performance**

- ▶ Begin a new line with each column name in the SELECT clause if many columns are being selected.
- ▶ Begin a new line with each table name in the FROM clause if many tables are being used.
- ▶ Begin a new line with each condition of the WHERE clause—You can easily see all conditions of the statement and the order in which they are used.

The following is an example of an unreadable statement:

```
SELECT CUSTOMER_TBL.CUST_ID, CUSTOMER_TBL.CUST_NAME,
CUSTOMER_TBL.CUST_PHONE, ORDERS_TBL.ORD_NUM, ORDERS_TBL.QTY
FROM CUSTOMER_TBL, ORDERS_TBL
WHERE CUSTOMER_TBL.CUST_ID = ORDERS_TBL.CUST_ID
AND ORDERS_TBL.QTY > 1 AND CUSTOMER_TBL.CUST_NAME LIKE 'G%'
ORDER BY CUSTOMER_TBL.CUST_NAME;
```

CUST_ID	CUST_NAME	CUST_PHONE	ORD_NUM	QTY
287	GAVINS PLACE	3172719991	18D778	10

1 row selected.

Here the statement has been reformatted for improved readability:

```
SELECT C.CUST_ID,
       C.CUST_NAME,
       C.CUST_PHONE,
       O.ORD_NUM,
       O.QTY
FROM ORDERS_TBL O,
     CUSTOMER_TBL C
WHERE O.CUST_ID = C.CUST_ID
      AND O.QTY > 1
      AND C.CUST_NAME LIKE 'G%'
ORDER BY 2;
```

CUST_ID	CUST_NAME	CUST_PHONE	ORD_NUM	QTY
287	GAVINS PLACE	3172719991	18D778	10

1 row selected.

Both statements are exactly the same, but the second statement is much more readable. The second statement has been greatly simplified by using table aliases, which have been defined in the query's FROM clause. Spacing has been used to align the elements of each clause, making each clause stand out.

Again, making a statement more readable does not directly improve its performance, but it assists you in making modifications and debugging a lengthy and otherwise complex statement. Now you can easily identify the columns being

selected, the tables being used, the table joins being performed, and the conditions being placed on the query.

It is especially important to establish coding standards in a multi-user programming environment. If all code is consistently formatted, shared code and modifications to code are much easier to manage.

**By the  
Way**

## Arrangement of Tables in the FROM Clause

The arrangement or order of tables in the FROM clause might make a difference, depending on how the optimizer reads the SQL statement. For example, it might be more beneficial to list the smaller tables first and the larger tables last. Some users with lots of experience have found that listing the larger tables last in the FROM clause proves to be more efficient.

The following is an example FROM clause:

```
FROM SMALLEST TABLE,
     LARGEST TABLE
```

Check your particular implementation for performance tips, if any, when listing multiple tables in the FROM clause.

**By the  
Way**

## Order of Join Conditions

As you learned in Hour 13, “Joining Tables in Queries,” most joins use a base table to link tables that have one or more common columns on which to join. The base table is the main table that most or all tables are joined to in a query. The column from the base table is normally placed on the right side of a join operation in the WHERE clause. The tables being joined to the base table are normally in order from smallest to largest, similar to the tables listed in the FROM clause.

If a base table doesn’t exist, the tables should be listed from smallest to largest, with the largest tables on the right side of the join operation in the WHERE clause. The join conditions should be in the first position(s) of the WHERE clause followed by the filter clause(s), as shown in the following:

FROM TABLE1,	Smallest table
TABLE2,	to
TABLE3	Largest table, also base table
WHERE TABLE1.COLUMN = TABLE3.COLUMN	Join condition
AND TABLE2.COLUMN = TABLE3.COLUMN	Join condition
[ AND CONDITION1 ]	Filter condition
[ AND CONDITION2 ]	Filter condition

In this example, TABLE3 is used as the base table. TABLE1 and TABLE2 are joined to TABLE3 for both simplicity and proven efficiency.

### ***Did you Know?***

Because joins typically return a high percentage of rows of data from the table(s), join conditions should be evaluated after more restrictive conditions.

## **The Most Restrictive Condition**

The most restrictive condition is typically the driving factor in achieving optimal performance for a SQL query. What is the most restrictive condition? The condition in the WHERE clause of a statement that returns the fewest rows of data. Conversely, the least restrictive condition is the condition in a statement that returns the most rows of data. This hour is concerned with the most restrictive condition simply because it is this condition that filters the data that is to be returned by the query the most.

It should be your goal for the SQL optimizer to evaluate the most restrictive condition first because a smaller subset of data is returned by the condition, thus reducing the query's overhead. The effective placement of the most restrictive condition in the query requires knowledge of how the optimizer operates. The optimizers, in some cases, seem to read from the bottom of the WHERE clause up. Therefore, you would want to place the most restrictive condition last in the WHERE clause, which is the condition that is first read by the optimizer.

FROM TABLE1,	Smallest table
TABLE2,	to
TABLE3	Largest table, also base table
WHERE TABLE1.COLUMN = TABLE3.COLUMN	Join condition
AND TABLE2.COLUMN = TABLE3.COLUMN	Join condition
[ AND CONDITION1 ]	Least restrictive
[ AND CONDITION2 ]	Most restrictive

### ***Did you Know?***

If you do not know how your particular implementation's SQL optimizer works, the DBA does not know, or you do not have sufficient documentation, you can execute a large query that takes a while to run, and then rearrange conditions in the WHERE clause. Be sure to record the time it takes the query to complete each time you make changes. You should only have to run a couple of tests to figure out whether the optimizer reads the WHERE clause from the top to bottom or bottom to top. If possible, it is best to turn off database caching during the testing for more accurate results.

The following is an example using a phony table:

Table:	TEST
Row count:	95,867
Conditions:	WHERE LAST_NAME = 'SMITH' returns 2,000 rows
	WHERE CITY = 'INDIANAPOLIS' returns 30,000 rows
Most restrictive condition is:	WHERE LAST_NAME = 'SMITH'

The following is the first query:

```
SELECT COUNT(*)
FROM TEST
WHERE LAST_NAME = 'SMITH'
      AND CITY = 'INDIANAPOLIS';

COUNT(*)
-----
      1,024
```

The following is the second query:

```
SELECT COUNT(*)
FROM TEST
WHERE CITY = 'INDIANAPOLIS'
      AND LAST_NAME = 'SMITH';

COUNT(*)
-----
      1,024
```

Suppose that the first query completed in 20 seconds, whereas the second query completed in 10 seconds. Because the second query returned faster results and the most restrictive condition was listed last in the WHERE clause, it would be safe to assume that the optimizer reads the WHERE clause from the bottom up.

It is a good practice to try to use an indexed column as the most restrictive condition in a query. Indexes generally improve a query's performance.

## Full Table Scans

A full table scan occurs when an index is either not used or there is no index on the table(s) being used by the SQL statement. Full table scans usually return data much slower than when an index is used. The larger the table, the slower that data is returned when a full table scan is performed. The query optimizer decides whether to use an index when executing the SQL statement. The index is used—if it exists—in most cases.

Some implementations have sophisticated query optimizers that can decide whether an index should be used. Decisions such as this are based on statistics that are gathered on database objects, such as the size of an object and the estimated number of rows that are returned by a condition with an indexed column. Please refer to your implementation documentation for specifics on the decision-making capabilities of your relational database's optimizer.

Full table scans should be avoided when reading large tables. For example, a full table scan is performed when a table that does not have an index is read, which usually takes a considerably longer time to return the data. An index should be considered for the majority of larger tables. On small tables, as previously mentioned, the optimizer might choose the full table scan over using the index, if the table is indexed. In the case of a small table with an index, consideration should be given to dropping the index and reserving the space that was used for the index for other needy objects in the database.

### ***Did you Know?***

The easiest and most obvious way to avoid a full table scan—outside of ensuring that indexes exist on the table—is to use conditions in a query's WHERE clause to filter data to be returned.

The following is a reminder of data that should be indexed:

- ▶ Columns used as primary keys
- ▶ Columns used as foreign keys
- ▶ Columns frequently used to join tables
- ▶ Columns frequently used as conditions in a query
- ▶ Columns that have a high percentage of unique values

Sometimes full table scans are good. Full table scans should be performed on queries against small tables or queries whose conditions return a high percentage of rows. The easiest way to force a full table scan is to avoid creating an index on the table.

## Other Performance Considerations

Other performance considerations should be noted when tuning SQL statements. The following concepts are discussed in the next sections:

- ▶ Using the LIKE operator and wildcards
- ▶ Avoiding the OR operator
- ▶ Avoiding the HAVING clause
- ▶ Avoiding large sort operations
- ▶ Using stored procedures

### Using the LIKE Operator and Wildcards

The LIKE operator is a useful tool that is used to place conditions on a query in a flexible manner. The placement and use of wildcards in a query can eliminate many possibilities of data that should be retrieved. Wildcards are very flexible for queries that search for similar data (data that is not equivalent to an exact value specified).

Suppose you want to write a query using the EMPLOYEE\_TBL selecting the EMP\_ID, LAST\_NAME, FIRST\_NAME, and STATE columns. You need to know the employee identification, name, and state for all the employees with the last name Stevens. Three SQL statement examples with different wildcard placements serve as examples.

The following is Query 1:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, STATE
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE 'STEVENS';
```

Next is Query 2:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, STATE
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE '%EVENS%';
```

Here is the last query, Query 3:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME, STATE
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE 'ST%';
```

The SQL statements do not necessarily return the same results. More than likely, Query 1 will return more rows than the other two queries. Query 2 and Query 3 are more specific as to the desired returned data, thus eliminating more possibilities than Query 1 and speeding data retrieval time. Additionally, Query 3 is probably faster than Query 2 because the first letters of the string for which you are searching are specified (and the column LAST\_NAME is likely to be indexed). Query 3 can take advantage of an index.

**By the  
Way**

With Query 1, you might retrieve all individuals with the last name Stevens; but can't Stevens also be spelled different ways? Query 2 picks up all individuals with the last name Stevens and its various spellings. Query 3 also picks up any last name starting with ST; this is the only way to assure that you receive all the Stevens (or Stephens).

## Avoiding the OR Operator

Rewriting the SQL statement using the IN predicate instead of the OR operator consistently and substantially improves data retrieval speed. Your implementation will tell you about tools you can use to time or check the performance between the OR operator and the IN predicate. An example of how to rewrite a SQL statement by taking the OR operator out and replacing the OR operator with the IN predicate follows.

**By the  
Way**

Hour 8, "Using Operators to Categorize Data," can be referenced for the use of the OR operator and the IN predicate.

The following is a query using the OR operator:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
WHERE CITY = 'INDIANAPOLIS'
       OR CITY = 'BROWNSBURG'
       OR CITY = 'GREENFIELD';
```

The following is the same query using the IN operator:

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
WHERE CITY IN ('INDIANAPOLIS', 'BROWNSBURG',
              'GREENFIELD');
```

The SQL statements retrieve the very same data; however, through testing and experience, you find that the data retrieval is measurably faster by replacing OR conditions with the IN predicate, as in the second query.

## Avoiding the HAVING Clause

The HAVING clause is a useful clause; however, you can't use it without cost. Using the HAVING clause gives the SQL optimizer extra work, which results in extra time. If possible, SQL statements should be written without using the HAVING clause.

## Avoiding Large Sort Operations

Large sort operations mean the use of the ORDER BY, GROUP BY, and HAVING clauses. Subsets of data must be stored in memory or to disk (if there is not enough space in allotted memory) whenever sort operations are performed. You must often sort data. The main point is that these sort operations affect a SQL statement's response time. Because large sort operations cannot always be avoided, it is best to schedule queries with large sorts as periodic batch processes during off-peak database usage so that the performance of most user processes is not affected.

## Using Stored Procedures

Stored procedures should be created for SQL statements executed on a regular basis—particularly large transactions or queries. Stored procedures are simply SQL statements that are compiled and permanently stored in the database in an executable format.

Normally, when a SQL statement is issued in the database, the database must check the syntax and convert the statement into an executable format within the database (called *parsing*). The statement, after it is parsed, is stored in memory; however, it is not permanent. This means that when memory is needed for other operations, the statement might be ejected from memory. In the case of stored procedures, the SQL statement is always available in an executable format and remains in the database until it is dropped like any other database object. Stored procedures are discussed in more detail in Hour 22, "Advanced SQL Topics."

## Disabling Indexes During Batch Loads

When a user submits a transaction to the database (INSERT, UPDATE, or DELETE), an entry is made to both the database table and any indexes associated with the table being modified. This means that if there is an index on the EMPLOYEE table, and a user updates the EMPLOYEE table, an update also occurs to the index associated with

the EMPLOYEE table. In a transactional environment, the fact that a write to an index occurs every time a write to the table occurs is usually not an issue.

During batch loads, however, an index can actually cause serious performance degradation. A batch load might consist of hundreds, thousands, or millions of manipulation statements or transactions. Because of their volume, batch loads take a long time to complete and are normally scheduled during off-peak hours—usually during weekends or evenings. To optimize performance during a batch load—which might equate to decreasing the time it takes the batch load to complete from 12 hours to 6 hours—it is recommended that the indexes associated with the table affected during the load are dropped. When the indexes are dropped, changes are written to the tables much faster, so the job completes faster. When the batch load is complete, the indexes should be rebuilt. During the rebuild of the indexes, the indexes will be populated with all the appropriate data from the tables. Although it might take a while for an index to be created on a large table, the overall time expended if you drop the index and rebuild it is less.

Another advantage to rebuilding an index after a batch load completes is the reduction of fragmentation that is found in the index. When a database grows, records are added, removed, and updated, and fragmentation can occur. For any database that experiences a lot of growth, it is a good idea to periodically drop and rebuild large indexes. When an index is rebuilt, the number of physical extents that comprise the index is decreased, there is less disk I/O involved to read the index, the user gets results faster, and everyone is happy.

## Performance Tools

Many relational databases have built-in tools that assist in SQL statement database performance tuning. For example, Oracle has a tool called EXPLAIN PLAN that shows the user the execution plan of a SQL statement. Another tool in Oracle measures the actual elapsed time of a SQL statement is TKPROF. In SQL Server, the Query Analyzer has several options to provide you with an estimated execution plan or statistics from the executed query. Check with your DBA and implementation documentation for more information on tools that might be available to you.

## Summary

You have learned the meaning of tuning SQL statements in a relational database. You have learned that there are two basic types of tuning: database tuning and SQL statement tuning—both of which are vital to the efficient operation of the database and SQL statements within it. Each is equally important and cannot be optimally

tuned without the other. Tuning the database falls to the DBA, whereas tuning SQL statements falls to the individuals writing the statements. This book is more concerned with the latter.

You have read about methods for tuning a SQL statement, starting with a statement's actual readability, which does not directly improve performance but aids the programmer in the development and management of statements. One of the main issues in SQL statement performance is the use of indexes. There are times to use indexes and times to avoid using them. A full table scan is performed when a table is read and an index is not used. In a full table scan, each row of data in a table is completely read. Other considerations for statement tuning, such as the arrangement of elements in a query, were discussed. Of foremost importance is the placement of the most restrictive condition in a statement's WHERE clause. For all measures taken to improve SQL statement performance, it is important to understand the data itself, database design and relationships, and the users' needs as far as accessing the database.

Like building indexes on tables, SQL statement tuning often involves extensive testing, which can be qualified as trial and error. There is no one way to tune a database or SQL statements within a database. All databases are different, as the business needs for each company are different. These differences affect the data within the database and the methods in which the data is retrieved. It is your job to crack the riddle of the most efficient SQL statement design for optimal database performance.

## Q&A

- Q.** *By following what I have learned about performance, what realistic performance gains, as far as data retrieval time, can I really expect to see?*
- A.** Realistically, you could see performance gains from fractions of a second to minutes, hours, or even days.
- Q.** *How can I test my SQL statements for performance?*
- A.** Each implementation should have a tool or system to check performance. Oracle7 was used to test the SQL statements in this book. Oracle has several tools for use in checking performance. Some of these tools are the EXPLAIN PLAN, TKPROF, and SET commands. Check your particular implementation for tools that are similar to Oracle's.

## Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. Would the use of a unique index on a small table be of any benefit?
2. What happens when the optimizer chooses not to use an index on a table when a query has been executed?
3. Should the most restrictive clause(s) be placed before the join condition(s) or after the join conditions in the WHERE clause?

## Exercises

1. Rewrite the following SQL statements to improve their performance. Use the EMPLOYEE\_TBL and the EMPLOYEE\_PAY\_TBL as described here:

```
EMPLOYEE_TBL
EMP_ID          VARCHAR(9)      NOT NULL      Primary key,
LAST_NAME       VARCHAR(15)     NOT NULL,
FIRST_NAME      VARCHAR(15)     NOT NULL,
MIDDLE_NAME     VARCHAR(15),
ADDRESS        VARCHAR(30)   NOT NULL,
CITY            VARCHAR(15)   NOT NULL,
STATE          VARCHAR(2)    NOT NULL,
ZIP            INTEGER(5)    NOT NULL,
PHONE          VARCHAR(10),
PAGER          VARCHAR(10),
CONSTRAINT EMP_PK PRIMARY KEY (EMP_ID)
```

```
EMPLOYEE_PAY_TBL
EMP_ID          VARCHAR(9)      NOT NULL      primary key,
POSITION        VARCHAR(15)     NOT NULL,
DATE_HIRE       DATETIME,
PAY_RATE        DECIMAL(4,2)   NOT NULL,
DATE_LAST_RAISE DATETIME,
SALARY          DECIMAL(8,2),
BONUS           DECIMAL(8,2),
CONSTRAINT EMP_FK FOREIGN KEY (EMP_ID)
REFERENCES EMPLOYEE_TBL (EMP_ID)
```

**A.**

```
SELECT EMP_ID, LAST_NAME, FIRST_NAME,
       PHONE
FROM EMPLOYEE_TBL
WHERE SUBSTRING(PHONE, 1, 3) = '317' OR
      SUBSTRING(PHONE, 1, 3) = '812' OR
      SUBSTRING(PHONE, 1, 3) = '765';
```

**B.**

```
SELECT LAST_NAME, FIRST_NAME
FROM EMPLOYEE_TBL
WHERE LAST_NAME LIKE '%ALL%';
```

**C.**

```
SELECT E.EMP_ID, E.LAST_NAME, E.FIRST_NAME,
       EP.SALARY
FROM EMPLOYEE_TBL E,
EMPLOYEE_PAY_TBL EP
WHERE LAST_NAME LIKE 'S%'
AND E.EMP_ID = EP.EMP_ID;
```

*This page intentionally left blank*

## **PART VI**

# **Using SQL to Manage Users and Security**

<b>HOUR 18</b>	Managing Database Users	<b>283</b>
<b>HOUR 19</b>	Managing Database Security	<b>297</b>

*This page intentionally left blank*

## HOURL 18

# Managing Database Users

During this hour, you learn about one of the most critical administration functions for any relational database: managing database users. You will learn the concepts behind creating users in SQL, user security, the user versus the schema, user profiles, user attributes, and tools users utilize.

---

### ***The highlights of this hour include:***

- ▶ Types of users
- ▶ User management
- ▶ The user's place in the database
- ▶ The user versus the schema
- ▶ User sessions
- ▶ Altering a user's attributes
- ▶ User profiles
- ▶ Dropping users from the database
- ▶ Tools utilized by users

The SQL standard refers to a database user identification as an *Authorization Identifier* (authID). In most major implementations, authIDs are referred to simply as *users*. This book refers to Authorization Identifiers as users, database users, usernames, or database user accounts. The SQL standard states that the Authorization Identifier is a name by which the system knows the database user.

***By the  
Way***

## **Users Are the Reason**

Users are the reason for the season—the season of designing, creating, implementing, and maintaining any database. The user's needs are taken into consideration when the database is designed, and the final goal in implementing a database is making the database available to users, who in turn utilize the database that you and possibly many others have had a hand in developing.

A common perception of users is that if there were no users, nothing bad would ever happen to the database. Although this statement reeks with truth, the database was nevertheless created to hold data so users can function in their day-to-day jobs.

Although user management is often the database administrator's implicit task, other individuals sometimes take a part in the user management process. User management is vital in the life of a relational database and is ultimately managed through the use of SQL concepts and commands, although they vary from vendor to vendor. The ultimate goal of the database administrator in terms of user management is to strike the proper balance between giving users access to the data that they need and still maintaining the integrity of the data within the system.

## **Types of Users**

There are several types of database users:

- ▶ Data entry clerks
- ▶ Programmers
- ▶ System engineers
- ▶ Database administrators
- ▶ System analysts
- ▶ Developers
- ▶ Testers
- ▶ Management
- ▶ End user

Each type of user has its own set of job functions (and problems), all of which are critical to their daily survival and job security. Furthermore, each type of user has different levels of authority and its own place in the database.

Titles, roles, and duties of users vary widely (and wildly) from workplace to workplace, depending on the size of each organization and each organization's specific data processing needs. One organization's DBA might be another organization's "computer guy."

## Who Manages Users?

A company's management staff is responsible for the day-to-day management of users; however, the database administrator or other assigned individuals are ultimately responsible for the management of users within the database.

The *database administrator* (DBA) usually handles the creation of the database user accounts, roles, privileges, and profiles, as well as dropping those user accounts from the database. Because it can become an overwhelming task in a large and active environment, some companies have a security officer who assists the DBA with the user management process.

The *security officer*, if one is assigned, is usually responsible for the paperwork, relaying to the DBA a user's job requirements and letting the DBA know when a user no longer requires access to the database.

The *system analyst*, or system administrator, is usually responsible for the operating system security, which entails creating users and assigning appropriate privileges. The security officer also might assist the system analyst in the same way he does the database administrator.

Maintaining an orderly way in which to assign and remove permissions as well as documenting the changes will make the process much easier to maintain. Documentation also allows you to have a paper trail in which to point to when the security of your system would possibly need to be audited either internally or externally. We will expand on the user management system throughout this hour.

## The User's Place in the Database

A user should be given the roles and privileges necessary to accomplish her job. No user should have database access that extends beyond the scope of her job duties. Protecting the data is the entire reason for setting up user accounts and security. Data can be damaged or lost, even if unintentionally, if the wrong user has access to the wrong data. When the user no longer requires database access, that user's account should be either removed from the database or disabled as quickly as possible.

**By the  
Way**

User account management is vital to the protection and success of any database, and when not managed systematically, it often fails. User account management is one of the simplest database management tasks, theoretically, but is often complicated by politics and communication problems.

All users have their place in the database; some have more responsibilities and different duties than others. Database users are like parts of a human body—all work together in unison (at least that is the way it is supposed to be) to accomplish some goal.

## **How Does a User Differ from a Schema?**

A database's objects are associated with database user accounts, called schemas. A *schema* is a set of database objects that a database user owns. This database user is called the *schema owner*. The difference between a regular database user and a schema owner is that a schema owner owns objects within the database, whereas most users do not own objects. Most users are given database accounts to access data that is contained in other schemas. Because the schema owner actually owns these objects, he has complete control over them.

## **The Management Process**

A stable user management system is mandatory for data security in any database system. The user management system starts with the new user's immediate supervisor, who should initiate the access request, and then go through the company's approval authorities. If the request is accepted by management, it is routed to the security officer or database administrator, who takes action. A good notification process is necessary; the supervisor and the user must be notified that the user account has been created and that access to the database has been granted. The user account password should only be given to the user, who should immediately change the password upon initial login to the database.

## **Creating Users**

The creation of database users involves the use of SQL commands within the database. There is no one standard command for creating database users in SQL; each implementation has a method for doing so. Some implementations have similar commands, while others vary in syntax. The basic concept is the same, regardless of the implementation. There are several graphical user interface (GUI) tools on the market that can be used for user management.

When the DBA or assigned security officer receives a user account request, the request should be analyzed for the necessary information. The information should include your particular company's requirements for establishing a user ID.

Some items that should be included are Social Security number, full name, address, phone number, office or department name, assigned database, and sometimes, a suggested user ID.

Syntactical examples of creating users compared between the different implementations are shown in the following sections.

You must check your particular implementation for the creation of users. Also refer to company policies and procedures when creating and managing users. The following section compares the user creation processes in Oracle, MySQL, Sybase, and Microsoft SQL Server.

**By the  
Way**

## Creating Users in Oracle

Following are the steps for creating a user account in an Oracle database:

1. Create the database user account with default settings.
2. Grant appropriate privileges to the user account.

The following is the syntax for creating a user:

```
CREATE USER USER_ID
IDENTIFIED BY [PASSWORD | EXTERNALLY ]
[ DEFAULT TABLESPACE TABLESPACE_NAME ]
[ TEMPORARY TABLESPACE TABLESPACE_NAME ]
[ QUOTA (INTEGER (K | M) | UNLIMITED) ON TABLESPACE_NAME ]
[ PROFILE PROFILE_TYPE ]
[PASSWORD EXPIRE |ACCOUNT [LOCK | UNLOCK]
```

The previous syntax for creating users can be used to add a user to an Oracle database, as well as a few other major relational database implementations.

The CREATE USER command is not supported by MySQL. Users can be managed using the `mysqladmin` tool. After a local user account is set up on a Windows computer, a login is not required. However, a user should be set up for each user requiring access to the database in a multiuser environment using `mysqladmin`.

**By the  
Way**

If you are not using Oracle, do not overly concern yourself with some of the options in this syntax. A *tablespace* is a logical area that houses database objects, such as tables and indexes, which is managed by the DBA. The DEFAULT TABLESPACE is the

tablespace in which objects created by the particular user reside. The `TEMPORARY TABLESPACE` is the tablespace used for sort operations (`table joins`, `ORDER BY`, `GROUP BY`) from queries executed by the user. The `QUOTA` is the space limit placed on a particular tablespace to which the user has access. `PROFILE` is a particular database profile that has been assigned to the user.

The following is the syntax for granting privileges to the user account:

```
GRANT PRIV1 [ , PRIV2, ... ] TO USERNAME | ROLE [ , USERNAME ]
```

The `GRANT` statement can grant one or more privileges to one or more users in the same statement. The privilege(s) can also be granted to a role, which in turn can be granted to a user(s).

In `MySQL`, the `GRANT` command can be used to grant users on the local computer to the current database. For example:

```
GRANT USAGE ON *.* TO USER@LOCALHOST IDENTIFIED BY 'PASSWORD';
```

Additional privileges can be granted to a user as follows:

```
GRANT SELECT ON TABLENAME TO USER@LOCALHOST;
```

For the most part, multiuser setup and access for `MySQL` is only required in multi-user environments.

## **Creating Users in Sybase and Microsoft SQL Server**

The steps for creating a user account in a `Sybase` and `Microsoft SQL Server` database follow:

1. Create the database user account for `SQL Server` and assign a password and a default database for the user.
2. Add the user to the appropriate database(s).
3. Grant appropriate privileges to the user account.

The following is the syntax for creating the user account:

```
SP_ADDLOGIN USER_ID ,PASSWORD [ , DEFAULT_DATABASE ]
```

The following is the syntax for adding the user to a database:

```
SP_ADDUSER USER_ID [ , NAME_IN_DB [ , GRPNAME ] ]
```

The following is the syntax for granting privileges to the user account:

```
GRANT PRIV1 [ , PRIV2, ... ] TO USER_ID
```

The discussion of privileges within a relational database is further elaborated on during Hour 19, “Managing Database Security.”

**By the  
Way**

## Creating Users in MySQL

The steps for creating a user account in MySQL follow:

1. Create the user account within the database.
2. Grant the appropriate privileges to the user account.

The syntax for creating the user account is very similar to the syntax used in Oracle.

```
SELECT USER user [IDENTIFIED BY [PASSWORD] 'password']
```

The syntax for granting the user’s privileges is also similar to the Oracle version:

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)]] ...
  ON [object_type]
     {tbl_name | * | *.* | db_name.* | db_name.routine_name}
  TO user
```

## Creating Schemas

Schemas are created via the CREATE SCHEMA statement.

The syntax is as follows:

```
CREATE SCHEMA [ SCHEMA_NAME ] [ USER_ID ]
  [ DEFAULT CHARACTER SET CHARACTER_SET ]
  [ PATH SCHEMA_NAME [, SCHEMA_NAME] ]
  [ SCHEMA_ELEMENT_LIST ]
```

The following is an example:

```
CREATE SCHEMA USER1
CREATE TABLE TBL1
  (COLUMN1 DATATYPE [NOT NULL],
   COLUMN2 DATATYPE [NOT NULL]...)
CREATE TABLE TBL2
  (COLUMN1 DATATYPE [NOT NULL],
   COLUMN2 DATATYPE [NOT NULL]...)
GRANT SELECT ON TBL1 TO USER2
GRANT SELECT ON TBL2 TO USER2
[ OTHER DDL COMMANDS ... ]
```

The following is the application of the `CREATE SCHEMA` command in one implementation:

```
CREATE SCHEMA AUTHORIZATION USER1
CREATE TABLE EMP
  (ID      NUMBER      NOT NULL,
  NAME    VARCHAR2(10) NOT NULL)
CREATE TABLE CUST
  (ID      NUMBER      NOT NULL,
  NAME    VARCHAR2(10) NOT NULL)
GRANT SELECT ON TBL1 TO USER2
GRANT SELECT ON TBL2 TO USER2;
```

Schema created.

The `AUTHORIZATION` keyword is added to the `CREATE SCHEMA` command. This example was performed in an Oracle database. This goes to show you, as you have also seen in this book's previous examples, that vendors' syntax for commands often varies in their implementations.

### **By the Way**

Some implementations might not support the `CREATE SCHEMA` command. However, schemas can be implicitly created when a user creates objects. The `CREATE SCHEMA` command is simply a single-step method of accomplishing this task. After objects have been created by a user, the user can grant privileges that allow access to the user's objects to other users.

The `CREATE SCHEMA` command is not supported by MySQL. A schema in MySQL is considered to be a database. So you would use the `CREATE DATABASE` command to essentially create a schema to populate with objects.

## **Dropping a Schema**

A schema can be removed from the database using the `DROP SCHEMA` statement. Two options must be considered when dropping a schema. First, the `RESTRICT` option. If `RESTRICT` is specified, an error occurs if objects currently exist in the schema. The second option is `CASCADE`. The `CASCADE` option must be used if any objects currently exist in the schema. Remember that when you drop a schema, you also drop all database objects associated with that schema.

The syntax is as follows:

```
DROP SCHEMA SCHEMA_NAME { RESTRICT | CASCADE }
```

The absence of objects in a schema is possible because objects, such as tables, can be dropped using the DROP TABLE command. Some implementations might have a procedure or command that drops a user, which can also be used to drop a schema. If the DROP SCHEMA command is not available in your implementation, you can remove a schema by removing the user that owns the schema objects.

## Altering Users

A very important part of managing users is the ability to alter a user's attributes after user creation. Life for the DBA would be a lot simpler if personnel with user accounts were never promoted, never left the company, or if the addition of new employees was minimized. In the real world, high personnel turnover and changes in users' responsibilities are a reality and a significant factor in user management. Nearly everyone changes jobs or job duties. Therefore, user privileges in a database must be adjusted to fit a user's needs.

The following is Oracle's example of altering the current state of a user:

```
ALTER USER USER_ID [ IDENTIFIED BY PASSWORD | EXTERNALLY | GLOBALLY AS 'CN=USER' ]
[ DEFAULT TABLESPACE TABLESPACE_NAME ]
[ TEMPORARY TABLESPACE TABLESPACE_NAME ]
[ QUOTA INTEGER K|M | UNLIMITED ON TABLESPACE_NAME ]
[ PROFILE PROFILE_NAME ]
[ PASSWORD EXPIRE ]
[ ACCOUNT [ LOCK | UNLOCK ] ]
[ DEFAULT ROLE ROLE1 [ , ROLE2 ] | ALL ]
[ EXCEPT ROLE1 [ , ROLE2 | NONE ] ]
```

Many of the user's attributes can be altered in this syntax. Unfortunately, not all implementations provide a simple command that allows the manipulation of database users.

MySQL, for instance, uses several means to modify the user account. For example, you would use the following syntax to reset the user's password in MySQL:

```
UPDATE mysql.user SET Password=PASSWORD('new password')
WHERE user='username';
```

Additionally, you might want to change the username for the user. You could accomplish this with the following syntax:

```
RENAME USER oId_username TO new_username;
```

Some implementations also provide GUI tools that allow users to be created, modified, and removed.

**By the  
Way**

You must check your particular implementation for the correct syntax for altering users. Oracle's ALTER USER syntax is shown here. In most major implementations, there is a tool that is used to alter or change a user's roles, privileges, attributes, and password.

A user can change an established password. You must check your particular implementation for the exact syntax or tool used to reset a password. The ALTER USER command is typically used in Oracle.

## User Sessions

A user database *session* is the time that begins at database login and ends when a user logs out. During the time a user is logged in to the database (a user session), the user can perform various actions that have been granted, such as queries and transactions.

An SQL session is initiated when a user connects from the client to the server using the CONNECT statement. Upon the establishment of the connection and the initiation of the session, any number of transactions can be started and performed until the connection is disconnected; at that time, the database user session terminates.

Users can explicitly connect and disconnect from the database, starting and terminating SQL sessions, using commands such as the following:

```
CONNECT TO DEFAULT | STRING1 [ AS STRING2 ] [ USER STRING3 ]
```

```
DISCONNECT DEFAULT | CURRENT | ALL | STRING
```

```
SET CONNECTION DEFAULT | STRING
```

**By the  
Way**

Remember that the syntax varies between implementations. In addition, most database users do not manually issue the commands to connect or disconnect from the database. Most users access the database through a vendor-provided or third-party tool that prompts the user for a username and password, which in turn connects to the database and initiates a database user session.

User sessions can be—and often are—monitored by the DBA or other personnel having interest in user activities. A user session is associated with a particular user account when a user is monitored. A database user session is ultimately represented as a process on the host operating system.

## Removing User Access

Removing a user from the database or disallowing a user's access can easily be accomplished through a couple of simple commands. Once again, however, variations among different implementations are numerous, so you must check your particular implementation for the syntax or tools used to accomplish user removal or access revocation.

Following are methods used for removing user database access:

- ▶ Change the user's password.
- ▶ Drop the user account from the database.
- ▶ Revoke appropriate previously granted privileges from the user.

The `DROP` command can be used in some implementations to drop a user from the database:

```
DROP USER USER_ID [ CASCADE ]
```

The `REVOKE` command is the counterpart of the `GRANT` command in many implementations, allowing privileges that have been granted to a user to be revoked. An example syntax for this command for SQL Server, Oracle, and MySQL is as follows:

```
REVOKE PRIV1 [ ,PRIV2, ... ] FROM USERNAME
```

## Tools Utilized by Database Users

Some people say that you do not need to know SQL to perform database queries. In a sense, they might be correct; however, knowing SQL definitely helps when querying a database, even when using GUI tools. Even though GUI tools are good and should be used when available, it is most beneficial to understand what is happening behind the scenes so you can maximize the efficiency of utilizing these user-friendly tools.

Many GUI tools that aid the database user automatically generate SQL code by navigating through windows, responding to prompts, and selecting options. There are reporting tools that generate reports. Forms can be created for users to query, update, insert, or delete data from a database. There are tools that convert data into graphs and charts. There are database administration tools that are used to monitor database performance and some that allow remote connectivity to a database. Database vendors provide some of these tools, whereas others are provided as third-party tools from other vendors.

## Summary

All databases have users, whether one or thousands. The user is the reason for the database.

There are three basic steps in the management of users. First, the database user account must be created. Second, privileges must be granted to the user to accommodate the tasks the user must perform within the database. Finally, a user account must either be removed from the database or certain privileges within the database must be revoked from a user.

Some of the most common tasks of managing users have been touched on; too much detail is avoided here, because most databases differ in the user management process. However, it is important to discuss user management due to its relationship with SQL. Many of the commands used to manage users have not been defined or discussed in great detail by the ANSI standard, but the concept remains the same.

## Q&A

**Q. *Is there a SQL standard for adding users to a database?***

**A.** Some commands and concepts are provided by ANSI, although each implementation and each company has its own commands, tools, and rules for creating or adding users to a database.

**Q. *Can user access be temporarily suspended without removing the user ID completely from the database?***

**A.** Yes. User access can temporarily be suspended by simply changing the user's password or by revoking privileges that allow the user to connect to the database. The functionality of the user account can be reinstated by changing and issuing the password to the user or by granting privileges to the user that might have been revoked.

**Q. *Can a user change his own password?***

**A.** Yes, in most major implementations. Upon user creation or addition to the database, a generic password is usually given to the user and must be changed as quickly as possible by the user to a password of his choice. After this has been accomplished, even the DBA does not know the user's password.

# Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. What command is used to establish a session?
2. Which option must be used to drop a schema that still contains database objects?
3. What command is used in MySQL to create a schema?
4. What statement is used to remove a database privilege?
5. What command creates a grouping or collection of tables, views, and privileges?

## Exercises

1. Describe how you would create a new user 'John' in your learnsql database.
2. How would you grant access to the Employee\_tbl to your new user 'John'?
3. Describe how you would assign permissions to all objects within the learnsql database to 'John'.
4. Describe how you would revoke the previous privileges from 'John' and then remove his account.
5. At the `mysql>` prompt, type the following to show the status of your current MySQL session:  
**status;**

*This page intentionally left blank*

## HOUR 19

# Managing Database Security

During this hour, you learn the basics of implementing and managing security within a relational database using SQL and SQL-related commands. Each major implementation differs on syntax with its security commands, but the overall security for the relational database follows the same basic guidelines discussed in the ANSI standard. You must check your particular implementation for syntax and any special guidelines for security.

---

### ***The highlights of this hour include***

- ▶ Database security
- ▶ Security versus user management
- ▶ Database system privileges
- ▶ Database object privileges
- ▶ Granting privileges to users
- ▶ Revoking privileges from users
- ▶ Security features in the database

## **What Is Database Security?**

Database security is simply the process of protecting the data from unauthorized usage. Unauthorized usage includes data access by database users who should have access to part of the database, but not all parts. This protection also includes the act of policing against unauthorized connectivity and distribution of privileges. Many user levels exist in a database, from the database creator to individuals responsible for maintaining the database (such as the database administrator [DBA]) to database programmers to end users. End users, although individuals with the most limited access, are the users for which the database exists. Each user has a different level of access to the database and should be limited to the fewest number of privileges needed to perform his particular job.

You might be wondering what the difference between user management and database security is. After all, the previous hour discussed user management, which seems to cover security. Although user management and database security are definitely related, each has its own purpose and work together to achieve a secure database.

A well-planned and maintained user management program goes hand-in-hand with the overall security of a database. Users are assigned user accounts and passwords that give the users general access to the database. The user accounts within the database should be stored with information, such as the user's actual name, the office and department in which the user works, a telephone number or extension, and the database name to which the user has access. Personal user information should only be accessible to the DBA. An initial password for the database user account is assigned by the DBA or security officer and should be changed immediately by the new user. Remember that the DBA does not need nor should want to know the individual's password. This ensures a separation of duties and protects the DBA's integrity should possible problems with a user's account arise.

Security entails more; for instance, if a user no longer requires certain privileges granted to her, those privileges should be revoked. If a user no longer requires access to the database, the user account should be dropped from the database.

Generally, *user management* is the process of creating user accounts, removing user accounts, and keeping track of users' actions within the database. *Database security* is going a step further by granting privileges for specific database access, revoking those privileges from users, and taking measures to protect other parts of the database, such as the underlying database files.

Because this is a SQL book, not a database book, it focuses on database privileges. However, you should keep in mind other aspects to database security, such as the protection of underlying database files, which holds equal importance with the distribution of database privileges. High-level database security can become complex and differs immensely between relational database implementations. If you would like to learn more about database security, you can find information on The Center for Internet Security's web page: <http://www.cisecurity.org/>.

## What Are Privileges?

*Privileges* are authority levels used to access the database itself, access objects within the database, manipulate data in the database, and perform various administrative functions within the database. Privileges are issued via the GRANT command and are taken away via the REVOKE command.

Just because a user can connect to a database does not mean that the user can access data within a database. Access to data within the database is handled through these privileges. The two types of privileges are system privileges and object privileges.

## System Privileges

*System privileges* are those that allow database users to perform administrative actions within the database, such as creating a database, dropping a database, creating user accounts, dropping users, dropping and altering database objects, altering the state of objects, altering the state of the database, and other actions that could result in serious repercussions if not carefully used.

System privileges vary greatly among the different relational database vendors, so you must check your particular implementation for all the available system privileges and their correct usage.

The following are some common system privileges in Sybase:

- ▶ CREATE DATABASE
- ▶ CREATE DEFAULT
- ▶ CREATE PROCEDURE
- ▶ CREATE RULE
- ▶ CREATE VIEW
- ▶ DUMP DATABASE
- ▶ DUMP TRANSACTION
- ▶ EXECUTE

The following are some common system privileges in Oracle:

- ▶ CREATE TABLE
- ▶ CREATE ANY TABLE
- ▶ ALTER ANY TABLE
- ▶ DROP TABLE
- ▶ CREATE USER
- ▶ DROP USER
- ▶ ALTER USER

- ▶ ALTER DATABASE
- ▶ ALTER SYSTEM
- ▶ BACKUP ANY TABLE
- ▶ SELECT ANY TABLE

The following are some common global (system) privileges in MySQL:

- ▶ CREATE
- ▶ DROP
- ▶ GRANT
- ▶ REFERENCES
- ▶ FILE
- ▶ PROCESS
- ▶ RELOAD
- ▶ SHUTDOWN

**By the  
Way**

MySQL has global privileges and object privileges. Global privileges, similar to system privileges, deal with user access to all database objects.

## Object Privileges

*Object privileges* are authority levels on objects, meaning you must have been granted the appropriate privileges to perform certain operations on database objects. For example, to select data from another user's table, the user must first grant you access to do so. Object privileges are granted to users in the database by the object's owner. Remember that this owner is also called the schema owner.

The ANSI standard for privileges includes the following object privileges:

- ▶ USAGE—Authorizes usage of a specific domain.
- ▶ SELECT—Allows access to a specific table.
- ▶ INSERT(*column\_name*)—Allows data insertion to a specific column of a specified table.
- ▶ INSERT—Allows insertion of data into all columns of a specific table.

- ▶ `UPDATE(column_name)`—Allows a specific column of a specified table to be updated.
- ▶ `UPDATE`—Allows all columns of a specified table to be updated.
- ▶ `REFERENCES(column_name)`—Allows a reference to a specified column of a specified table in integrity constraints; this privilege is required for all integrity constraints.
- ▶ `REFERENCES`—Allows references to all columns of a specified table.

Most implementations of SQL adhere to the standard list of object privileges for controlling access to database objects.

The owner of an object has been automatically granted all privileges that relate to the objects owned. These privileges have also been granted with the `GRANT OPTION`, which is a nice feature available in some SQL implementations. This feature is discussed in the “`GRANT OPTION`” section later this hour.

**By the  
Way**

These object-level privileges should be used to grant and restrict access to objects in a schema. These privileges can be used to protect objects in one schema from database users that have access to another schema in the same database.

A variety of object privileges are available among different implementations not listed in this section. The capability to delete data from another user’s object is another common object privilege available in many implementations. Be sure to check your implementation documentation for all the available object-level privileges.

## Who Grants and Revokes Privileges?

The DBA is usually the one who issues the `GRANT` and `REVOKE` commands, although a security administrator, if one exists, might have the authority to do so. The authority on what to grant or revoke would come from management and would hopefully be in writing.

The owner of an object must grant privileges to other users in the database on the object. Even the DBA cannot grant database users privileges on objects that do not belong to the DBA, although there are ways to work around that.

## Controlling User Access

User access is primarily controlled by a user account and password, but that is not enough to access the database in most major implementations. The creation of a user account is only the first step in allowing and controlling access to the database.

After the user account has been created, the database administrator, security officer, or designated individual must be able to assign appropriate system-level privileges to a user for that user to be allowed to perform actual functions within the database, such as creating tables or selecting from tables. Furthermore, the schema owner usually needs to grant database users access to objects in the schema so that the user can do his job.

Two commands in SQL allow database access control involving the assignment of privileges and the revocation of privileges. The `GRANT` and `REVOKE` commands are used to distribute both system and object privileges in a relational database.

### The GRANT Command

The `GRANT` command is used to grant both system-level and object-level privileges to an existing database user account.

The syntax is as follows:

```
GRANT PRIVILEGE1 [, PRIVILEGE2] [ ON OBJECT ]  
TO USERNAME [ WITH GRANT OPTION | ADMIN OPTION]
```

Granting one privilege to a user is as follows:

```
GRANT SELECT ON EMPLOYEE_TBL TO USER1;
```

Grant succeeded.

Granting multiple privileges to a user is as follows:

```
GRANT SELECT, INSERT ON EMPLOYEE_TBL TO USER1;
```

Grant succeeded.

Notice that when granting multiple privileges to a user in a single statement, each privilege is separated by a comma.

Granting privileges to multiple users is as follows:

```
GRANT SELECT, INSERT ON EMPLOYEE_TBL TO USER1, USER2;
```

Grant succeeded.

Notice the phrase `Grant succeeded`, denoting the successful completion of each grant statement. This is the feedback that you receive when you issue these statements in the implementation used for the book examples (Oracle). Most implementations have some sort of feedback, although the phrase used might vary.

**By the  
Way**

### GRANT OPTION

`GRANT OPTION` is a very powerful `GRANT` command option. When an object's owner grants privileges on an object to another user with `GRANT OPTION`, the new user can also grant privileges on that object to other users, even though the user does not actually own the object. An example follows:

```
GRANT SELECT ON EMPLOYEE_TBL TO USER1 WITH GRANT OPTION;
```

Grant succeeded.

### ADMIN OPTION

`ADMIN OPTION` is similar to `GRANT OPTION` in that the user that has been granted the privileges also inherits the ability to grant those privileges to another user. `GRANT OPTION` is used for object-level privileges, whereas `ADMIN OPTION` is used for system-level privileges. When a user grants system privileges to another user with `ADMIN OPTION`, the new user can also grant the system-level privileges to any other user. An example follows:

```
GRANT CREATE TABLE TO USER1 WITH ADMIN OPTION;
```

Grant succeeded.

When a user that has granted privileges using either `GRANT OPTION` or `ADMIN OPTION` has been dropped from the database, the privileges that the user granted are disassociated with the users to whom the privileges were granted.

**By the  
Way**

## The REVOKE Command

The `REVOKE` command removes privileges that have been granted to database users. The `REVOKE` command has two options: `RESTRICT` and `CASCADE`. When the `RESTRICT` option is used, `REVOKE` succeeds only if the privileges specified explicitly in the `REVOKE` statement leave no other users with abandoned privileges. The `CASCADE` option revokes any privileges that would otherwise be left with other users. In other words, if the owner of an object granted `USER1` privileges with `GRANT OPTION`, `USER1`

granted USER2 privileges with GRANT OPTION, and then the owner revokes USER1's privileges, CASCADE also removes the privileges from USER2.

*Abandoned privileges* are privileges that are left with a user who was granted privileges with the GRANT OPTION from a user who has been dropped from the database or had her privileges revoked.

The syntax for REVOKE is as follows:

```
REVOKE PRIVILEGE1 [, PRIVILEGE2 ] [ GRANT OPTION FOR ] ON OBJECT
FROM USER { RESTRICT | CASCADE }
```

The following is an example:

```
REVOKE INSERT ON EMPLOYEE_TBL FROM USER1;
```

Revoke succeeded.

## Controlling Access on Individual Columns

Instead of granting object privileges (INSERT, UPDATE, or DELETE) on a table as a whole, you can grant privileges on specific columns in the table to restrict user access, as shown in the following example:

```
GRANT UPDATE (NAME) ON EMPLOYEES TO PUBLIC;
```

Grant succeeded.

## The PUBLIC Database Account

The PUBLIC database user account is a database account that represents all users in the database. All users are part of the public account. If a privilege is granted to the PUBLIC account, all database users have the privilege. Likewise, if a privilege is revoked from the PUBLIC account, the privilege is revoked from all database users, unless that privilege was explicitly granted to a specific user. The following is an example:

```
GRANT SELECT ON EMPLOYEE_TBL TO PUBLIC;
```

Grant succeeded.

**Watch Out!**

Extreme caution should be taken when granting privileges to PUBLIC; all database users acquire the privileges granted. Therefore, by granting permissions to public you may unintentionally give access to data to users whom have no business accessing it. For example, giving PUBLIC access to SELECT from the employee salary table would give everyone whom has access to the database the rights to see what everyone in the company is being paid!

## Groups of Privileges

Some implementations have groups of privileges in the database. These groups of permissions are referred to with different names. Having a group of privileges allows simplicity for granting and revoking common privileges to and from users. For example, if a group consists of ten privileges, the group can be granted to a user instead of individually granting all ten privileges.

SQLBase has groups of privileges called *authority levels*, whereas these groups of privileges in Oracle are called roles. SQLBase and Oracle both include the following groups of privileges with their implementations:

- ▶ CONNECT
- ▶ RESOURCE
- ▶ DBA

The CONNECT group allows a user to connect to the database and perform operations on any database objects to which the user has access.

The RESOURCE group allows a user to create objects, drop objects he owns, grant privileges to objects he owns, and so on.

The DBA group allows a user to perform any function within the database. The user can access any database object and perform any operation with this group.

An example for granting a group of privileges to a user follows:

```
GRANT DBA TO USER1;
```

```
Grant succeeded.
```

Each implementation differs on the use of groups of database privileges. If available, this feature should be used for ease of database security administration.

**By the  
Way**

## Controlling Privileges Through Roles

A *role* is an object created in the database that contains group-like privileges. Roles can reduce security maintenance by not having to grant explicit privileges directly to a user. Group privilege management is much easier to handle with roles. A role's privileges can be changed, and such a change is transparent to the user.

If a user needs SELECT and UPDATE table privileges on a table at a specified time within an application, a role with those privileges can temporarily be assigned until the transaction is complete.

When a role is first created, it has no real value other than being a role within a database. It can be granted to users or other roles. Let's say that a schema named APP01 grants the SELECT table privilege to the RECORDS\_CLERK role on the EMPLOYEE\_PAY table. Any user or role granted the RECORDS\_CLERK role now would have SELECT privileges on the EMPLOYEE\_PAY table.

Likewise, if APP01 revoked the SELECT table privilege from the RECORDS\_CLERK role on the EMPLOYEE\_PAY table, any user or role granted the RECORDS\_CLERK role would no longer have SELECT privileges on that table.

Roles are not supported by MySQL. The lack of role usage is a weakness in some implementations of SQL.

### The CREATE ROLE Statement

A role is created with the CREATE ROLE statement.

```
CREATE ROLE role_name;
```

Granting privileges to roles is the same as granting privileges to a user. Study the following example:

```
CREATE ROLE RECORDS_CLERK;
```

Role created.

```
GRANT SELECT, INSERT, UPDATE, DELETE ON EMPLOYEE_PAY TO RECORDS_CLERK;
```

Grant succeeded.

```
GRANT RECORDS_CLERK TO USER1;
```

Grant succeeded.

### The DROP ROLE Statement

A role is dropped using the DROP\_ROLE statement.

```
DROP ROLE role_name;
```

The following is an example:

```
DROP ROLE RECORDS_CLERK;
```

Role dropped.

## The SET ROLE Statement

A role can be set for a user SQL session using the SET\_ROLE statement.

```
SET ROLE role_name;
```

The following is an example:

```
SET ROLE RECORDS_CLERK;
```

Role set.

You can set more than one role at once:

```
SET ROLE RECORDS_CLERK, ROLE2, ROLE3;
```

Role set.

In some implementations, such as Oracle, all roles granted to a user are automatically default roles, which means the roles will be set and available to the user as soon as the user logs in to the database.

## Summary

You were shown the basics on implementing security in a SQL database or a relational database. You learned the basics of managing database users. The first step in implementing security at the database level for users is to create the user; after the user has been created, the user must be assigned certain privileges that allow the user access to specific parts of the database, and now ANSI allows the use of roles as discussed during this hour. Privileges can be granted to users or roles.

The two types of privileges are system and object privileges. System privileges are those that allow the user to perform various different tasks within the database, such as actually connecting to the database, creating tables, creating users, altering the state of the database, and so on. Object privileges are those that allow a user access to specific objects within the database, such as the ability to select data or manipulate data in a specific table.

Two commands in SQL allow a user to grant and revoke privileges to and from other users or roles in the database: GRANT and REVOKE. These two commands are used to control the overall administration of privileges in the database. Although there are many other considerations for implementing security in a relational database, the basics that relate to the language of SQL were discussed during this hour.

## **Q&A**

- Q.** *If a user forgets her password, what should the user do to gain access to the database again?*
- A.** The user should go to her immediate management or an available help desk. A help desk should be able to reset a user's password. If not, the DBA or security officer can reset the password. The user should change the password to a password of her choosing as soon as the password is reset and the user is notified. Sometimes the DBA can affect this by setting a specific property that forces the user to change her password on the next login. Check your particular implementation's documentation for specifics.
- Q.** *What could I do if I wanted to grant CONNECT to a user, but the user does not need all the privileges that are assigned to the connect role?*
- A.** You would simply not grant CONNECT, but only the privileges required. Should you ever grant CONNECT and the user no longer needs all the privileges that go with it, simply revoke CONNECT from the user and grant the specific privileges required.
- Q.** *Why is it so important for the new user to change the password when received from whomever created the new user?*
- A.** An initial password is assigned upon creation of the user ID. No one, not even the DBA or management, should know a user's password. The password should be kept a secret at all times to prevent another user from logging on to the database under another user's account.

# Workshop

The following workshop is composed of a series of quiz questions and practical exercises. The quiz questions are designed to test your overall understanding of the current material. The practical exercises are intended to afford you the opportunity to apply the concepts discussed during the current hour, as well as build upon the knowledge acquired in previous hours of study. Please take time to complete the quiz questions and exercises before continuing. Refer to Appendix C, “Answers to Quizzes and Exercises,” for answers.

## Quiz

1. What option must a user have to grant another user privileges on an object not owned by the user?
2. When privileges are granted to PUBLIC, do all database users acquire the privileges, or only specified users?
3. What privilege is required to look at data in a specific table?
4. What type of privilege is SELECT?
5. What option is used for revoking a user’s privilege to an object as well as the other users that they might have granted privileges to by use of the GRANT option?

## Exercises

1. Log in to MySQL and type the following at the `mysql>` prompt to use the default `mysql` database:  

```
use mysql;
```
2. Type the following at the `mysql>` prompt to get a list of the default tables:  

```
show tables;
```
3. Now, describe each one of the following tables:  

```
describe columns_priv;  
describe db;  
describe host;  
describe tables_priv;  
describe user;
```

Each of these tables relates to database security in MySQL.

4. Create a new database user as follows:

```
GRANT USAGE ON *.* TO 'STEVE@LOCALHOST' IDENTIFIED BY 'STEVE123';
```

Although a user called STEVE has been created in the MySQL database, this new user account is not useful unless there is a user called STEVE at the operating system level (for example, in Windows or Linux). Logins are important for multiple user environments, such as Linux.

5. Get a list of all database users by typing the following:

```
SELECT * FROM USER;
```

## **PART VII**

# **Summarized Data Structures**

<b>HOUR 20</b>	Creating and Using Views and Synonyms	<b>313</b>
<b>HOUR 21</b>	Working with the System Catalog	<b>329</b>

*This page intentionally left blank*





















































































































































































































































































# Index

## Symbols

- + (addition operator), 135, 210
- / (division operator), 136
- || (double pipe signs), 353
- = (equal operator), 118
- > (greater than operator), 119-120
- < (less than operator), 119-120
- \* (multiplication operator), 135-136
- != (non-equality operator), 119
- ;(semicolons), 46
- " (single quotation marks), 353
- (subtraction operator), 135

## A

- abandoned privileges, 304
- ABS (absolute value) function, 178
- accessing
  - remote databases, 361
    - JDBC, 363
    - ODBC, 362

- vendor connectivity tools, 363
- web interface, 363-364
- user access, controlling, 302, 361
  - columns, 304
  - GRANT statement, 302-303
  - groups of privileges, 305
  - PUBLIC database account, 304
  - REVOKE statement, 303-304
- adding
  - auto-incrementing columns to tables, 48
  - characters to strings, 176-177
  - columns to tables, 48
  - data to tables, 74-75
    - from another table, 76-78
    - NULL values, 78-79
    - into specified columns, 75-76
  - rows into views, 321
  - time to dates, 190-191
- addition operator (+), 135
- ADD\_MONTHS function, 190
- ADMIN OPTION (GRANT statement), 303

**aggregate functions****aggregate functions**

AVG, 146-147  
 COUNT, 142-144  
 definition, 141-142  
 GROUP BY clause, 153-156  
 MAX, 147  
 MIN, 147-148  
 SUM, 144-146

**aliases**

columns, 112-113  
 tables, 208

**ALL operator, 126****ALL option (SELECT statement), 104****ALTER TABLE statement, 47-48, 381****American National Standards Institute. See ANSI****AND operator, 127-128****ANSI (American National Standards Institute), 8**

character functions, 165  
   concatenation, 166-168  
   INSTR, 172  
   LOWER, 170  
   LTRIM, 173  
   REPLACE, 169  
   RTRIM, 173-174  
   SUBSTR, 170-171  
   substrings, 166  
   TRANSLATE, 166-169  
   UPPER, 169  
 object privileges, 300  
 SELECT statement syntax, 370  
 trigger creation syntax, 349

**ANSI SQL, 8, 371****ANY operator, 126****arithmetic operators, 134**

addition, 135  
 combining, 136-137

division, 136  
 multiplication, 135-136  
 subtraction, 135

**ascending order, 106****ASCII characters, returning, 178****ASCII chart website, 178****ASCII function, 178****authIDs (Authorization Identifiers), 283****authority levels, 305****AUTHORIZATION keyword (CREATE SCHEMA statement), 290****auto-incrementing columns, 48****automated population, 74****AVG function, 146-147****avoiding**

indexes, 259-260  
 large sort operations, 275

**B****back-end applications, 360-361****base tables, join considerations, 214-215****BETWEEN operator, 122, 222****BLOB data type, 30****book website, 9****BOOLEAN data types, 34****C****call-level interface (CLI), 352****Cartesian product, 215-217****CASCADE option (REVOKE statement), 303**

- case sensitivity (queries), 108**
- CEIL function, 178**
- ceiling values function, 178**
- Center for Internet Security website, 298**
- CHAR data type, 29**
- character functions, 165**
  - ASCII, 178
  - COALESCE, 176
  - combining, 181-182
  - concatenation, 166-168
  - DECODE, 174-175
  - IFNULL, 175-176
  - INSTR, 172
  - LENGTH, 175
  - LOWER, 170
  - LPAD, 176-177
  - LTRIM, 173
  - REPLACE, 169
  - RPAD, 177
  - RTRIM, 173-174
  - SUBSTR, 170-171
  - substrings, 166
  - TRANSLATE, 166-169
  - UPPER, 169
- character string conversions**
  - dates, 196-197
  - to numbers, 179-180
- characters**
  - adding to strings, 176-177
  - ASCII, returning, 178
  - constant, 29
  - lowercase, 170
  - positions, 172
  - replacing, 169
  - trimming, 173-174
  - uppercase, 169
- CHK (check) constraints, 55-56**
- clauses**
  - FROM, 385
    - SELECT statement, 104
    - table arrangement, 269
  - GROUP BY, 152, 385
    - aggregate functions, 153-156
    - compared to ORDER BY clause, 156-159
    - compound queries, 244-245
    - CREATE VIEW statement, 323
    - functions, 152
    - ordering column names with numbers, 156
    - selected data, 152
  - HAVING, 159-160, 275, 385
  - ORDER BY, 385
    - compared to GROUP BY clause, 156-159
    - compound queries, 242-244
    - SELECT statement, 106-108
    - views, 323
  - SELECT, 102, 384
  - WHERE, 385
    - DELETE statement, 81
    - restrictive condition, 270-271
    - SELECT statement, 105-106
- CLI (call-level interface), 352**
- client/server systems, 12**
- closing cursors, 345-346**
- COALESCE function, 176**
- Codd, Dr. E.F., 8**
- columns, 21, 44-45**
  - adding, 48
  - aliases, 112-113
  - attributes, editing, 48

**columns**

- auto-incrementing, adding, 48
  - averaging values, 146-147
  - cardinality, 260
  - check constraints, 55-56
  - counting values, 142-144
  - data, adding, 75-76
  - dropping constraints, 56
  - editing, 49
  - foreign keys, 53-54
  - index considerations, 258
  - maximum values, 147
  - minimum values, 147-148
  - NOT NULL constraints, 55
  - NULL values, 78-79
  - ordering with numbers, 156
  - primary keys, 52-53
  - qualifying, 205
  - totaling values, 144-146
  - unique constraints, 53
  - updating, 79-80
  - user access control, 304
- combining**
- arithmetic operators, 136-137
  - character functions, 181-182
  - comparison operators, 120-121
- commands. See statements**
- COMMIT statement, 89-90, 381**
- comparison operators, 118**
- combining, 120-121
  - equal, 118
  - less than, greater than, 119-120
  - non-equality, 119
- composite indexes, 257**
- compound queries, 235**
- clauses
    - GROUP BY, 244-245
    - ORDER BY, 242-244
  - data retrieval, 246
  - operators
    - EXCEPT, 241-242
    - INTERSECT, 240-241
    - UNION, 237-240
- concatenation, 166-168**
- conditions, queries, 105-106**
- conjunctive operators, 127**
- AND, 127-128
  - OR, 128-130
- CONNECT statement, 14**
- CONNECT group, 305**
- connecting sessions, 14**
- constant characters, 29**
- constraints (integrity), 52**
- check, 55-56
  - dropping, 56
  - foreign keys, 53-54
  - NOT NULL, 55
  - primary keys, 52-53
  - unique, 53
- controlling**
- data, 16
  - transactions, 88-89
    - COMMIT statement, 89-90
    - performance, 95
    - RELEASE SAVEPOINT statement, 94
    - ROLLBACK statement, 90-92
    - ROLLBACK TO SAVEPOINT statement, 92-94
    - SAVEPOINT statement, 92
    - SET TRANSACTION statement, 94
  - statements, 17

- user access, 302
  - columns, 304
  - GRANT statement, 302-303
  - groups of privileges, 305
  - PUBLIC database account, 304
  - REVOKE statement, 303-304
- conversion functions, 179**
  - character strings to numbers, 179-180
  - numeric strings to characters, 180-181
- converting dates, 192**
  - character strings, 196-197
  - date pictures, 193-195
- correlated subqueries, 229-230**
- COUNT function, 111, 142-144**
- counting table records, 111**
- CREATE DOMAIN statement, 381**
- CREATE INDEX statement, 255, 381**
- CREATE ROLE statement, 306, 382**
- CREATE SCHEMA statement, 289-290**
- CREATE TABLE AS statement, 382**
- CREATE TABLE statement, 45-47, 50-51, 382**
  - CUSTOMER TBL statement, 436
  - EMPLOYEE PAY TBL statement, 435
  - EMPLOYEE TBL statement, 435
  - ORDERS TBL statement, 436
  - PRODUCTS TBL statement, 436
- CREATE TRIGGER statement, 349-350**
- CREATE TYPE statement, 382**
- CREATE VIEW statement, 316, 382**
  - GROUP BY clause, 323
  - views from multiple tables, 318-319
  - views from other views, 319-320
  - views from single tables, 316-318
  - WITH CHECK OPTION, 320-321
- creating**
  - indexes, 255
  - roles, 306
  - schemas, 289-290
  - SQL with SQL, 352-353
  - synonyms, 324-325
  - system catalog, 331
  - tables, 45-47
    - existing tables, 50-51
    - from views, 322
  - triggers, 349-350
  - users, 286
    - MySQL, 289
    - Oracle, 287-288
    - SQL Server, 288-289
    - Sybase, 288-289
  - views
    - from single tables, 316-318
    - from multiple tables, 318-319
    - from other views, 319-320
    - WITH CHECK OPTION, 320-321
- cross joins, 215-217**
- current date/time function, 188**
- cursors**
  - closing, 345-346
  - current values, 344
  - declaring, 344
  - definition, 343
  - fetching data from, 345
  - opening, 345
  - overview, 344

**data****D****data**

- administration, 17
- controlling, 16
- definition, 27
- fetching from cursors, 345
- for indexes, 272
- grouping, 151
  - GROUP BY clause, 152-156
  - GROUP BY clause versus ORDER BY clause, 156-159
  - HAVING clause, 159-160
- manipulating, 16, 73
- populating tables, 74
- redundancy, 63
- retrieving from compound queries, 246
- selecting
  - statements, 16
  - multiple tables, 203
- simplifying with views, 314
- summarized data maintenance, 315-316
- system catalog, 331-332
- tables
  - deleting, 81
  - examples in book, 18-20
  - inserting, 74-75
  - inserting from another table, 76-78
  - inserting into specified columns, 75-76
  - inserting NULL values, 78-79
  - selecting from another table, 112
  - updating, 79-80
- views, updating, 321

**Data Control Language (DCL), 16****Data Definition Language (DDL), 15****data dictionaries. See system catalog****Data Manipulation Language. See DML****Data Query Language (DQL), 16****data types**

- basic, 28
- BLOB, 30
- BOOLEAN, 34
- CHAR, 29
- date and time, 32-33, 186-187
- decimal, 31-32
- definition, 27
- domains, 35
- DOUBLE PRECISION, 32
- fixed-length strings, 29
- floating-point decimals, 32
- integers, 32
- large objects, 30
- lengths, 37
- literal strings, 33-34
- NULL, 34
- numeric, 30-31
- REAL, 32
- TEXT, 30
- user-defined, 35
- VARCHAR, 29
- varying-length strings, 29

**database administrators (DBAs), 285****database management system (DBMS), 7****databases**

- client/server systems, 12
- definition, 10
- denormalizing, 69
- design information, 332
- full table scans, 254
- Internet access tools, 365

- logical, 62-63
- MySQL examples/exercises, 22
- normalizing
  - benefits, 67-68
  - disadvantages, 68
  - names, 67
  - normal forms, 61, 64-66
  - overview, 61-62
- objects
  - definition, 41
  - schemas, 42-43
- parsing, 275
- queries. *See also* subqueries
  - case sensitivity, 108
  - column aliases, 112-113
  - compound. *See* compound queries
  - conditions, 105-106
  - counting table records, 111
  - definition, 16, 101
  - examples, 109-110
  - grouping results. *See* groups, data
  - ordering output, 106-108
  - searching, 174-175
  - SELECT statement, 101-104
  - SELECT statement with case sensitivity, 108
  - SELECT statement with FROM clause, 104
  - SELECT statement with ORDER BY clause, 106-108
  - SELECT statement with WHERE clause, 105-106
  - selecting data from another table, 112
  - single, 235
- raw, 62
- relational, 11
- remote, 361-364
- security, 297-298
  - privileges. *See* privileges
  - user access control, 302-305
- structures statements, 15
- transactions
  - statements, 17
  - controlling, 88-90
  - definition, 87
  - initiating, 94
  - overview, 87
  - performance, 95
  - savepoints, 92-94
  - saving changes, 89-90
  - undoing, 90-92
- tuning, 266
- users
  - authIDs, 283
  - creating, 286-287
  - creating in MySQL, 289
  - creating in Oracle, 287-288
  - creating in SQL Server, 288-289
  - creating in Sybase, 288-289
  - deleting, 293
  - editing, 291
  - GUI tools, 293
  - managing, 285
  - roles/privileges, 285
  - schemas, 286-290
  - sessions, 292
  - types, 284
- vendors, 13-14
- web-based systems, 12-13
- date and time data types, 32-33**

**DATEADD function****DATEADD function, 190****DATEDIFF function, 192****DATENAME function, 192****DATEPART function, 192****dates**

conversions, 192

character strings, 196-197

date pictures, 193-195

data types

implementation-specific, 187

standard, 186

date functions, 187

adding time, 190-191

comparing dates/times, 191

current, 188

miscellaneous, 192

time zones, 189

DATETIME elements, 186

parts, 194-195

pictures, 193-195

storing, 186

system, 188

**DATETIME data types, 32****DATETIME element, 186****DAYNAME function, 192****DAYOFMONTH function, 192****DAYOFWEEK function, 192****DAYOFYEAR function, 192****DBA group, 305****DBAs (database administrators), 285****dBASE, 333****DBMS (database management system), 7****DCL (Data Control Language), 16****DDL (Data Definition Language), 15****DECIMAL data type, 31****decimals, 31-32****DECODE function, 174-175****DELETE statement, 383**

subqueries, 226

table data, 81

WHERE clause, 81

**deleting**

rows into views, 321

savepoints, 94

schemas, 290

table data, 81

users, 293

**denormalization, 69****descending order, 106****differences in vendor implementations, 369-371****direct SQL, 353****DISCONNECT statement, 14****disconnecting sessions, 14****DISTINCT statement, 104, 142****division operator (/), 136****DML (Data Manipulation Language), 16**

DELETE statement

deleting table data, 81

subqueries, 226

INSERT statement

adding data from another table, 76-78

adding data to specific columns, 75-76

adding data to tables, 74-75

subqueries, 224-225

NULL values, 78-79

overview, 73

UPDATE statement

multiple columns, 80

single columns, 79-80

- subqueries, 225-226
- tables, 79
- domain data types, 35
- double pipe signs (||), 353
- DOUBLE PRECISION data type, 32
- DQL (Data Query Language), 16
- DROP statement, 51
  - indexes, 260-261
  - users, 293
- DROP INDEX statement, 383
- DROP ROLE statement, 306
- DROP SCHEMA statement, 290
- DROP TABLE statement, 383
- DROP TRIGGER statement, 351
- DROP VIEW statement, 323, 383
- dropping
  - constraints, 56
  - indexes, 260-261
  - roles, 306
  - synonyms, 325
  - tables, 51, 57
  - triggers, 351
  - views, 323
- dynamic SQL, 351-352

## E

- editing
  - columns, 49
  - tables, 47-49
  - users, 291
- embedded SQL, 353
- embedding subqueries, 227-228

- enhancements, 371
- enterprise, 359-361
- equal operator (=), 118
- equijoins, 204-206
- example extensions, 372-373
  - MySQL, 374-375
  - PL/SQL, 373-374
  - Transact-SQL, 373
- EXCEPT operator (compound queries), 241-242
- EXISTS operator, 125
- EXIT statement, 14
- exiting sessions, 14
- EXP (exponential values) function, 178
- extensions, 371-372
  - MySQL, 374-375
  - PL/SQL, 373-374
  - Transact-SQL, 373

## F

- FETCH statement, 345
- fetching data from cursors, 345
- fields (tables), 20
- firewalls, 364
- first normal forms, 64
- fixed-length strings, 29
- FLOAT data type, 32
- floating-point decimals, 32
- FLOOR function, 178
- floor values function, 178
- FOR EACH ROW syntax (triggers), 351
- foreign keys, 53-54
- forgotten passwords, 308

**formatting statements****formatting statements, 266**

- FROM clause table arrangement, 269
- join order, 269-270
- readability, 267-269
- WHERE clause condition, 270-271

**FROM clause, 385**

- SELECT statement, 104
- table arrangement, 269

**front-end applications, 360-361****front-end tools, 63****full table scans, 254, 272****functions**

- ADD\_MONTHS, 190
- aggregate
  - AVG, 146-147
  - COUNT, 142-144
  - definition, 141-142
  - GROUP BY clause, 153-156
  - MAX, 147
  - MIN, 147-148
  - SUM, 144-146
- character, 165
  - ASCII, 178
  - COALESCE, 176
  - combining, 181-182
  - concatenation, 166-168
  - DECODE, 174-175
  - IFNULL, 175-176
  - INSTR, 172
  - LENGTH, 175
  - LOWER, 170
  - LPAD, 176-177
  - LTRIM, 173
  - REPLACE, 169

RPAD, 177

RTRIM, 173-174

SUBSTR, 170-171

substrings, 166

TRANSLATE, 166-169

UPPER, 169

## conversion

- character strings to numbers, 179-180
- numeric strings to characters, 180-181

COUNT, 111

date, 187

- adding time, 190-191
- comparing dates/times, 191
- current, 188
- miscellaneous, 192
- time zones, 189

DATEADD, 190

DATEDIFF, 192

DATENAME, 192

DATEPART, 192

DAYNAME, 192

DAYOFMONTH, 192

DAYOFWEEK, 192

DAYOFYEAR, 192

definition, 141, 347

GETDATE(), 188, 192

GROUP BY clause, 152

mathematical, 178

MONTHS\_BETWEEN, 192

NEXT\_DAY, 192

NOW, 188

**G****GETDATE() function, 188, 192****GRANT statement, 383**

ADMIN OPTION, 303

GRANT OPTION, 303

privileges, 301

user access control, 302-303

**granting privileges, 301****greater than operator ( > ), 119-120****GROUP BY clause, 385**

aggregate functions, 153-156

compared to ORDER BY clause, 156-159

compound queries, 244-245

CREATE VIEW statement, 323

functions, 152

ordering column names with numbers, 156

selected data, 152

**groups**

data, 151

GROUP BY clause, 152-156

GROUP BY clause versus ORDER BY clause,  
156-159

HAVING clause, 159-160

privileges, 305

**GUI tools, 293****H - I****HAVING clause, 159-160, 275, 385****IFNULL function, 175-176****implementation-specific data types, 187****implementations**

ANSI SQL compliance, 371

cursors, 344

differences, 369-371

extensions, 371

SQL, 10

system catalog, 333-334

**implicit indexes, 257****IN operator, 123****indexes**

avoiding, 259-260

column considerations, 258

creating, 255

data for, 272

definition, 253

disabling during batch loads, 275-276

dropping, 260-261

function, 254-255

overview, 253-254

performance, 260, 275-276

types, 255

composite, 257

implicit, 257

single-column, 256

unique, 256-258

**Informix, 371****initiating transactions, 94****INSERT object privilege, 300****INSERT statement, 383**

adding data to tables, 74

from another table, 76-78

specified columns, 75-76

CUSTOMER TBL statement, 438

EMPLOYEE PAY TBL statement, 438

**INSERT statement**

EMPLOYEE TBL statement, 437

ORDERS TBL statement, 439

PRODUCTS TBL statement, 440

subqueries, 224-225

**INSERT(column\_name) object privilege, 300**

**INSERT...SELECT statement, 383**

**installing MySQL**

Linux, 388-389

Windows, 387-388

**INSTR function, 172**

**integers, 32**

**integrity constraints, 52**

check, 55-56

dropping, 56

foreign keys, 53-54

NOT NULL, 55

primary keys, 52-53

unique, 53

**interactive SQL statements, 375-376**

**International Standards Organization (ISO), 8**

**Internet**

data availability for employees/customers, 365

database access tools, 365

security, 366

worldwide availability, 364

**INTERSECT operator (compound queries), 240-241**

**intranets, 365-366**

**INX suffix, 18**

**IS NOT NULL operator, 133**

**IS NULL operator, 121-122**

**ISO (International Standards Organization), 8**

**J**

**JDBC (Java Database Connectivity), 363**

**joins**

base tables, 214-215

Cartesian product, 215-217

component locations, 204

equijoins, 204-206

multiple keys, 213-214

natural, 206-207

non-equijoins, 208-209

ordering, 269-270

outer, 210-211

self, 212-213

table aliases, 208

types, 204

**K-L**

**keys**

foreign, 53-54

joining, 213-214

primary, 21, 52-53

**large object data types, 30**

**LENGTH function, 175**

**lengths**

data types, 37

strings, 175

**less than operator (<), 119-120**

**LIKE operator, 123-124, 273**

**Linux, MySQL installation, 388-389**

**literal strings, 33-34**

**logical databases, 62-63**

**logical operators, 121**

ALL, 126  
 ANY, 126  
 BETWEEN, 122  
 EXISTS, 125  
 IN, 123  
 IS NULL, 121-122  
 LIKE, 123-124  
 SOME, 126

**LOWER function, 170****lowercase strings, 170****LPAD function, 176-177****LTRIM function, 173****M****managing users, 285**

creating users, 286-287  
     MySQL, 289  
     Oracle, 287-288  
     SQL Server, 288-289  
     Sybase, 288-289  
 deleting, 293  
 editing, 291  
 GUI tools, 293  
 schemas, 289-290  
 sessions, 292

**manipulating data, 16, 73****manual population of data, 74****mathematical functions, 178****MAX function, 147****Microsoft**

Access, 333  
 SQL Server, users, 288-289

**MIN function, 147-148****MONTHS\_BETWEEN function, 192****multiplication operator (\*), 135-136****MySQL, 374-375**

cursor declaration, 344  
 examples/exercises, 22  
 installing  
     Linux, 388-389  
     Windows, 387-388  
 stored procedure syntax, 347-348  
 system catalog implementations, 334  
 system privileges, 300  
 trigger creation syntax, 350  
 users, creating, 289  
 website, 375

**N****names**

normalization, 67  
 saving points, 92  
 synonyms, 326  
 tables, 18, 47

**natural joins, 206-207****negative operators, 130**

IS NOT NULL, 133  
 NOT BETWEEN, 131-132  
 not equal, 131  
 NOT EXISTS, 134  
 NOT IN, 132  
 NOT LIKE, 133

**nesting**

queries. See subqueries  
 stored procedures, 346

**Net8****Net8, 363****NEXT\_DAY function, 192****non-equality operator (!=), 119****non-equijoins, 208-209****normal forms, 61, 64**

first, 64

second, 65

third, 66

**normalization**

benefits, 67-68

disadvantages, 68

names, 67

normal forms, 61

first, 64

second, 65

third, 66

overview, 61-62

**NOT BETWEEN operator, 131-132****NOT EXISTS operator, 134****NOT IN operator, 132****NOT LIKE operator, 133****NOT NULL constraints, 55****NOW function, 188****NULL data types, 34****NULL value checker, 175-176****NULL values**

adding to columns, 78-79

checking, 175-176

replacing, 176

tables, 22

**NUMERIC data type, 30-31****numeric strings, converting to characters, 180-181****O****object privileges, 300-301****ODBC (Open Database Connectivity), 362****Open Client/C Developers Kit, 363****opening cursors, 345****operators**

arithmetic, 134

addition, 135

combining, 136-137

division, 136

multiplication, 135-136

subtraction, 135

**BETWEEN, 222**

comparison

combining, 120-121

equal, 118

less than, greater than, 119-120

non-equality, 119

conjunctive

AND, 127-128

OR, 128-130

definition, 105, 117

**EXCEPT, 241-242****INTERSECT, 240-241****LIKE, 273**

logical

ALL, 126

ANY, 126

BETWEEN, 122

EXISTS, 125

IN, 123

IS NULL, 121-122

LIKE, 123-124

SOME, 126

- negative, 130
    - IS NOT NULL, 133
    - NOT BETWEEN, 131-132
    - not equal, 131
    - NOT EXISTS, 134
    - NOT IN, 132
    - NOT LIKE, 133
  - OR, 274-275
  - OVERLAPS, 191
  - UNION, 235-239
  - UNION ALL, 239-240
  - options**
    - ADMIN OPTION, 303
    - ALL, 104
    - CASCADE, 303
    - DISTINCT, 104
    - GRANT OPTION, 303
    - RESTRICT, 303
    - WITH CHECK, 320-321
  - OR operator, 128-130, 274-275**
  - Oracle**
    - cursor declaration, 344
    - Net8, 363
    - parameters, 376
    - PL/SQL, 373-374
    - roles, 305
    - SELECT statement syntax, 370
    - stored procedure syntax, 347-348
    - system catalog implementations, 334
    - system privileges, 299
    - trigger creation syntax, 350
    - users, creating, 287-288
  - ORDER BY clause, 385**
    - compared to GROUP BY clause, 156-159
    - compound queries, 242-244
    - SELECT statement, 106-108
    - views, 323
  - outer joins, 210-211**
  - OVERLAPS operator, 191**
  - owners (schemas), 42**
- ## P
- parameters, 375**
  - parent/child table relationships, 54**
  - parsing, 275**
  - parts of dates, 194-195**
  - passwords**
    - forgotten, 308
    - system catalog, 338
  - performance**
    - definition, 265-266
    - formatting, 266
    - FROM clause table arrangement, 269
    - full table scans, 272
    - HAVING clause, 275
    - indexes, 260, 275-276
    - join order, 269-270
    - large sort operations, 275
    - LIKE operator, 273
    - OR operator, 274-275
    - readability, 267-269
    - stored procedures, 275
    - statistics stored in system catalog, 332
    - tools, 276
    - transactional control, 95
    - WHERE clause condition, 270-271
    - wildcard placement, 273

**PL/SQL****PL/SQL, 373-374**

plus (+) symbol, 210

**populating tables with data, 74-75**

from another table, 76-78

NULL values, 78-79

into specified columns, 75-76

**positioning characters, 172****POWER function, 178**

precision, 31

primary keys, 21, 52-53

**PRIVATE synonyms, 324****privileges, 298**

abandoned, 304

controlling with roles, 305-307

granting/revoking, 301

groups, 305

object, 300-301

system, 299-300

**pseudocolumns, 188****PUBLIC database account, 304****PUBLIC synonyms, 324**

UNION ALL operator, 239-240

UNION operator, 237-239

conditions, 105-106

counting table records, 111

definition, 16, 101

examples, 109-110

grouping results, 151

GROUP BY clause, 152-156

GROUP BY clause versus ORDER BY clause,  
156-159

HAVING clause, 159-160

ordering output, 106-108

searching, 174-175

SELECT statement, 101

case sensitivity, 108

FROM clauses, 104

ORDER BY clauses, 106-108

selecting data, 102-104

WHERE clauses, 105-106

selecting data from another table, 112

single, 235

system catalog, 335-336

**Q****qualifying columns, 205****queries. See also subqueries, 221**

case sensitivity, 108

column aliases, 112-113

compound, 235

data retrieval, 246

EXCEPT operator, 241-242

GROUP BY clause, 244-245

INTERSECT operator, 240-241

ORDER BY clause, 242-244

**R****raw databases, 62****RDBMS (relational database management system), 7****readability of statements, 267-269****REAL data type, 32**

records (tables), 21, 111

redundancy (data), 63

REFERENCES object privilege, 301

**REFERENCES(column\_name) object privilege, 301**

referential integrity, 68

relational database management system  
(RDBMS), 7

relational databases, 11

**RELEASE SAVEPOINT statement, 94**

remote databases, accessing, 361

JDBC, 363

ODBC, 362

vendor connectivity tools, 363

web interface, 363-364

**REPLACE function, 169**

replacing

characters, 169

NULL values, 176

**RESOURCE group, 305**

**RESTRICT keyword**

DROP SCHEMA statement, 290

REVOKE statement, 303

**REVOKE statement, 384**

privileges, 301

user access control, 303-304

users, 293

**revoking privileges, 301**

roles

creating, 306

dropping, 306

Oracle, 305

setting, 307

**ROLLBACK statement, 90-92, 384**

**ROLLBACK TO SAVEPOINT statement, 92-94**

rolling back savepoints, 92-94

**ROUND function, 178**

**rows, 21, 45**

averaging values, 146-147

counting, 142-144

maximum values, 147

minimum values, 147-148

totaling values, 144-146

views, 321

**RPAD function, 177**

**RTRIM function, 173-174**

## S

**SAVEPOINT statement, 92, 384**

savepoints

deleting, 94

names, 92

rolling back, 92-94

schemas

creating, 289-290

definition, 42

deleting, 290

overview, 42-43

owners, 42

users, compared, 286

**searching queries, 174-175**

**second normal forms, 65**

security

databases, 297-298

firewalls, 364

information stored in system catalog, 332

Internet, 366

privileges, 298

abandoned, 304

controlling with roles, 305-307

**security**

- granting/revoking, 301
- groups, 305
- object, 300-301
- system, 299-300
- roles, 305
  - creating, 306
  - dropping, 306
  - setting, 307
- user access
  - columns, 304
  - GRANT statement, 302-303
  - groups of privileges, 305
  - PUBLIC database account, 304
  - REVOKE statement, 303-304
- views, 315
- security officers, 285**
- SELECT statement, 384**
  - clauses, 102
  - column aliases, 112-113
  - COUNT function, 111
  - EXCEPT operator, 241-242
  - GROUP BY clause, 244-245
    - aggregate functions, 153-156
    - compared to ORDER BY clause, 156-159
    - functions, 152
    - ordering column names with numbers, 156
    - selected data, 152
  - HAVING clause, 159-160
  - implementation differences, 370
  - INTERSECT operator, 240-241
  - ORDER BY clause, 242-244
  - queries, 101
    - ALL option, 104
    - case sensitivity, 108
    - DISTINCT option, 104
    - FROM clause, 104
    - ORDER BY clause, 106-108
    - selecting data, 102-104
    - WHERE clause, 105-106
  - selecting data from another table, 112
  - single queries, 235
  - subqueries, 223-224
  - UNION ALL operator, 239-240
  - UNION operator, 237-239
- SELECT object privilege, 300**
- selecting data**
  - from another table, 112
  - statements, 16
  - multiple tables, 203
- self joins, 212-213**
- semicolons (;), 46**
- sessions**
  - connecting, 14
  - definition, 14
  - disconnecting, 14
  - exiting, 14
  - users, 292
- SET ROLE statement, 307**
- SET TRANSACTION statement, 94**
- SIGN function, 178**
- sign values function (SIGN), 178**
- single queries, 235**
- single quotation marks ("), 353**
- single-column indexes, 256**
- SOME operator, 126**
- sort operations, 275**
- SQL (Structured Query Language), 8**
  - definition, 8
  - generation with SQL, 352-353
  - implementation, 10

- on the Internet
  - data availability for employees/customers, 365
  - database access tools, 365
  - worldwide availability, 364
- optimizer, 267
- SQL Server**
  - cursor declaration, 344
  - stored procedure syntax, 347-348
  - system catalog implementations, 333
  - Transact-SQL, 373
  - trigger creation syntax, 350
  - users, creating, 288-289
- SQL-2003, 9-10**
- SQLBase**
  - authority levels, 305
  - SELECT statement syntax, 370
- SQRT (square root) function, 178**
- standard data types, 186**
- standards**
  - ANSI SQL, 8
  - SQL-2003, 9-10
  - table-naming, 18
- statements**
  - ALTER TABLE, 47-48, 381
  - COMMIT, 89-90, 381
  - CONNECT, 14
  - CREATE DOMAIN, 381
  - CREATE INDEX, 255, 381
  - CREATE ROLE, 306, 382
  - CREATE SCHEMA, 289-290
  - CREATE TABLE, 45-47, 50-51, 382
    - CUSTOMER TBL, 436
    - EMPLOYEE PAY TBL, 435
    - EMPLOYEE TBL, 435
    - ORDERS TBL, 436
    - PRODUCTS TBL, 436
  - CREATE TABLE AS, 382
  - CREATE TRIGGER, 349-350
  - CREATE TYPE, 382
  - CREATE VIEW, 382
    - GROUP BY clause, 323
    - views from multiple tables, 318-319
    - views from other views, 319-320
    - views from single tables, 316-318
    - WITH CHECK OPTION, 320-321
  - DELETE, 383
    - subqueries, 226
    - table data, 81
    - WHERE clause, 81
  - DISCONNECT, 14
  - DISTINCT, 142
  - DROP, 51
    - indexes, 260-261
    - users, 293
  - DROP INDEX, 383
  - DROP ROLE, 306
  - DROP SCHEMA, 290
  - DROP TABLE, 383
  - DROP TRIGGER, 351
  - DROP VIEW, 323, 383
  - EXIT, 14
  - FETCH, 345
  - formatting, 266
  - GRANT, 383
    - ADMIN OPTION, 303
    - GRANT OPTION, 303
    - privileges, 301
    - user access control, 302-303

## statements

- INSERT, 383
  - adding data to columns, 75-76
  - adding data to tables, 74-78
  - CUSTOMER TBL, 438
  - EMPLOYEE PAY TBL, 438
  - EMPLOYEE TBL, 437
  - ORDERS TBL, 439
  - PRODUCTS TBL, 440
  - subqueries, 224-225
- INSERT...SELECT, 383
- interactive, 375-376
- RELEASE SAVEPOINT, 94
- REVOKE, 384
  - privileges, 301
  - user access control, 303-304
  - users, 293
- ROLLBACK, 90-92, 384
- ROLLBACK TO SAVEPOINT, 92-94
- SAVEPOINT, 92, 384
- SELECT, 384
  - ALL option, 104
  - case sensitivity, 108
  - clauses, 102
  - column aliases, 112-113
  - COUNT function, 111
  - DISTINCT option, 104
  - EXCEPT operator, 241-242
  - FROM clause, 104
  - GROUP BY clause, 152-159, 244-245
  - HAVING clause, 159-160
  - implementation differences, 370
  - INTERSECT operator, 240-241
  - ORDER BY clause, 106-108, 242-244
  - queries, 101-104
  - selecting data from another table, 112
  - single queries, 235
  - subqueries, 223-224
  - UNION ALL operator, 239-240
  - UNION operator, 237-239
  - WHERE clause, 105-106
- SET ROLE, 307
- SET TRANSACTION, 94
- tuning, 265-266
  - formatting, 266
  - FROM clause table arrangement, 269
  - full table scans, 272
  - HAVING clause, 275
  - indexes, 275-276
  - join order, 269-270
  - large sort operations, 275
  - LIKE operator, 273
  - OR operator, 274-275
  - readability, 267-269
  - stored procedures, 275
  - tools, 276
  - WHERE clause condition, 270-271
  - wildcard placement, 273
- types
  - data administration, 17
  - data control, 16
  - defining database structures, 15
  - manipulating data, 16
  - selecting data, 16
  - transactional control, 17
- UPDATE, 384
  - multiple columns, 80
  - single columns, 79-80
  - subqueries, 225-226
  - table data, 79

**static SQL, 351****stored procedures**

- advantages, 348
- definition, 346
- MySQL syntax, 347-348
- nesting, 346
- Oracle syntax, 347-348
- overview, 347
- performance, 275
- SQL Server syntax, 347-348

**storing dates/times, 186**

- DATETIME elements, 186
- implementation-specific data types, 187
- standard data types, 186

**strings**

- characters
  - adding, 176-177
  - ASCII, 178
  - date conversions, 196-197
  - functions, 165
  - positions, 172
  - replacing, 169
- concatenation, 166-168
- conversions
  - character to numbers, 179-180
  - numeric to characters, 180-181
- fixed-length, 29
- lengths, 175
- literal, 33-34
- lowercases, 170
- NULL values, 175-176
- query searches, 174-175
- substrings, 166, 170-171
- translating, 166-169
- trimming, 173-174

- uppercase, 169
- varying-length, 29

**Structured Query Language. See SQL****subqueries**

- BETWEEN operator, 222
- correlated, 229-230
- definition, 221
- DELETE statement, 226
- embedded, 227-228
- INSERT statement, 224-225
- overview, 221-222
- rules, 222
- SELECT statement, 223-224
- syntax, 222
- UPDATE statement, 225-226

**SUBSTR function, 170-171****substrings, 166, 170-171****subtraction operator (-), 135****SUM function, 144-146****summarized data maintenance, 315-316****Sybase**

- Open Client/C Developers Kit, 363
- parameters, 376
- system catalog implementations, 334
- system privileges, 299
- users, creating, 288-289

**synonyms**

- creating, 324-325
- definition, 324
- dropping, 325
- names, 326
- overview, 324
- PRIVATE, 324
- PUBLIC, 324

**system catalog****system catalog**

- creating, 331
- data, 331-332
- definition, 329
- implementations, 333-334
- maintenance, 332
- overview, 330
- passwords, 338
- querying, 335-336
- table queries, 338
- updating, 337

**systems**

- analysts, 285
- client/server, 12
- date, 188
- privileges, 299-300
- web-based database, 12-13

**T****tables**

- aliases, 208
- arranging in FROM clauses, 269
- base, 214-215
- columns, 21, 44-45
  - adding, 48
  - adding data, 75-76
  - aliases, 112-113
  - attributes, editing, 48
  - auto-incrementing, adding, 48
  - averaging values, 146-147
  - cardinality, 260
  - check constraints, 55-56
  - counting values, 142-144

- dropping constraints, 56
- editing, 49
- foreign keys, 53-54
- index considerations, 258
- maximum values, 147
- minimum values, 147-148
- NOT NULL constraints, 55
- NULL values, 78-79
- ordering with numbers, 156
- primary keys, 52-53
- qualifying, 205
- totaling values, 144, 146
- unique constraints, 53
- updating, 79-80
- user access control, 304

**creating, 45-47**

- existing table, 50-51
- views, 322

**data**

- deleting, 81
- inserting, 74-75
- inserting from another table, 76-78
- inserting into specified columns, 75-76
- inserting NULL values, 78-79
- populating, 74
- selecting from another table, 112
- updating, 79-80

**data examples in book, 18, 20****dropping, 51, 57****editing, 47-49****fields, 20****joins**

- base tables, 214-215
- Cartesian product, 215-217
- component locations, 204

- equijoins, 204-206
  - multiple keys, 213-214
  - natural, 206-207
  - non-equijoins, 208-209
  - outer, 210-211
  - self, 212-213
  - table aliases, 208
  - types, 204
- names, 18, 47
- NULL values, 22
- parent/child relationships, 54
- primary keys, 21
- records, 21, 111
- relational databases, 11
- rows, 21, 45
  - averaging values, 146-147
  - counting, 142-144
  - maximum values, 147
  - minimum values, 147-148
  - totaling values, 144-146
- selecting data from multiple, 203
- system catalog, 338
- windowed table functions, 354
- TBL suffix, 18**
- TEXT data type, 30**
- third normal forms, 66**
- time zone function, 189**
- times**
  - adding to dates, 190-191
  - data types
    - implementation-specific, 187
    - standard, 186
  - date functions, 187
    - adding time, 190-191
    - comparing dates/times, 191
    - current, 188
    - miscellaneous, 192
    - time zones, 189
- DATETIME elements, 186
- storing, 186
- tools**
  - front-end, 63
  - GUI, 293
  - performance, 276
  - web database access, 365
- Transact-SQL, 373**
- transactions**
  - controlling, 88-90
  - databases, 17
  - definition, 87
  - initiating, 94
  - overview, 87
  - savepoints
    - deleting, 94
    - names, 92
    - performance, 95
    - rolling back, 92-94
  - saving changes, 89-90
  - undoing, 90-92
- TRANSLATE function, 166-169**
- translating strings, 166-169**
- triggers**
  - creating, 349-350
  - definition, 349
  - dropping, 351
  - FOR EACH ROW syntax, 351
- trimming strings, 173-174**
- troubleshooting passwords, 308**

**tuning****tuning**

- databases, 266

- SQL statements

- definition, 265-266

- formatting, 266

- FROM clause table arrangement, 269

- full table scans, 272

- HAVING clause, 275

- indexes, 275-276

- join order, 269-270

- large sort operations, 275

- LIKE operator, 273

- OR operator, 274-275

- readability, 267-269

- stored procedures, 275

- tools, 276

- WHERE clause condition, 270-271

- wildcard placement, 273

**types**

- statements

- data administration, 17

- data control, 16

- defining database structures, 15

- manipulating data, 16

- selecting data, 16

- transactional control, 17

- data

- basic, 28

- BLOB, 30

- BOOLEAN, 34

- CHAR, 29

- date and time, 32-33, 186-187

- DECIMAL, 31-32

- definition, 27

- domains, 35

- DOUBLE PRECISION, 32

- fixed-length strings, 29

- FLOAT, 32

- floating-point decimals, 32

- integers, 32

- large objects, 30

- lengths, 37

- literal strings, 33-34

- NULL, 34

- numeric, 30-31

- REAL, 32

- TEXT, 30

- user-defined, 35

- VARCHAR, 29

- varying-length strings, 29

- indexes, 255

- composite, 257

- implicit, 257

- single-column, 256

- unique, 256-258

- joins

- equijoins, 204-206

- natural, 206-207

- non-equijoins, 208-209

- outer, 210-211

- self, 212-213

- users, 284

**U**

- undoing transactions, 90-92

- UNION ALL operator, 239-240

- UNION operator, 235-239

- unique column constraints, 53

**unique indexes, 256-258**

**UPDATE object privilege, 301**

**UPDATE statement, 384**

subqueries, 225-226

table data, 79-80

**UPDATE(column\_name) object privilege, 301**

**updating**

system catalog, 337

table data, 79-80

view data, 321

**UPPER function, 169**

**uppercase strings, 169**

**USAGE object privilege, 300**

**user-defined data types, 35**

**users**

access, controlling

columns, 304

GRANT statement, 302-303

groups of privileges, 305

PUBLIC database account, 304

REVOKE statement, 303-304

authIDs, 283

creating, 286-287

MySQL, 289

Oracle, 287-288

SQL Server, 288-289

Sybase, 288-289

data, system catalog, 332

deleting, 293

editing, 291

GUI tools, 293

logical database design considerations, 63

managing, 285, 298

roles/privileges, 285

schemas, 289-290

schemas, compared, 286

sessions, 292

types, 284

## V

**values**

ceiling and floor function, 178

exponential function, 178

NULL

adding to columns, 78-79

checking, 175-176

replacing, 176

tables, 22

**VARCHAR data type, 29**

**varying-length strings, 29**

**vendors**

databases, 13-14

implementations, 369-371

**views**

creating, 316

multiple tables, 318-319

other views, 319-320

single tables, 316-318

WITH CHECK OPTION, 320-321

creating tables from, 322

data updates, 321

definition, 313

dependencies, 320

dropped tables, 326

dropping, 323

ORDER BY clause, 323

476

## views

- overview, 314
- rows, 321
- security, 315
- simplifying data, 314
- summarized data maintenance, 315-316

## W

**web interfaces, 363-364**

**web-based database systems, 12-13**

### websites

- ASCII chart, 178
- book, 9
- Center for Internet Security, 298
- MySQL, 375

**WHERE clause, 385**

- DELETE statement, 81
- restrictive condition, 270-271
- SELECT statement, 105-106

**wildcard performance, 273**

**windowed table functions, 354**

**WITH CHECK OPTION (CREATE VIEW statement),  
320-321**

## X-Z

**XML, 354-355**