Database Concepts

8th Edition

David M. Kroenke • David J. Auer

Scott L. Vandenberg • Robert C. Yoder

Online Appendix E

Advanced SQL

VP Editorial Director: Andrew Gilfillan
Senior Portfolio Manager: Samantha Lewis
Content Development Team Lead: Laura Burgess
Program Monitor: Ann Pulido/SPi Global
Editorial Assistant: Madeline Houpt
Product Marketing Manager: Kaylee Carlson
Project Manager: Katrina Ostler/Cenveo® Publisher Services
Text Designer: Cenveo® Publisher Services
Interior design: Stock-Asso/Shutterstock; Faysal Shutterstock
Cover Designer: Brian Malloy/Cenveo® Publisher Services
Cover Art: Artwork by Donna R. Auer
Full-Service Project Management: Cenveo® Publisher Services
Composition: Cenveo® Publisher Services
Printer/Binder: Courier/Kendallville
Cover Printer: Lehigh-Phoenix Color/Hagerstown
Text Font: 10/12 Simoncini Garamond Std.

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on the appropriate page within text.

Library of Congress Cataloging-in-Publication Data

Kroenke, David M., 1948- author. | Auer, David J., author.
Database concepts / David M. Kroenke, David J. Auer, Western Washington University, Scott L. Vandenberg, Siena College, Robert C. Yoder, Siena College.
Eighth edition. | Hoboken, New Jersey : Pearson, [2017] | Includes index.
LCCN 2016048321| ISBN 013460153X | ISBN 9780134601533
LCSH: Database management. | Relational databases.
LCC QA76.9.D3 K736 2017 | DDC 005.74--dc23
LC record available at https://lccn.loc.gov/2016048321

Appendix E — 10 9 8 7 6 5 4 3 2 1

Appendix Objectives

- To understand reasons for using the SQL ALTER statement
- To use the SQL ALTER statement
- To understand the need for the SQL MERGE statement
- To use the SQL MERGE statement
- To understand the need for additional type of SQL queries
- To use SQL outer join queries
- To use SQL correlated subqueries
- To use SQL queries on recursive relationships
- To understand the reasons for using SQL set operators
- To use SQL set operators
- To understand the reasons for using SQL views
- To use SQL statements to create and use SQL views
- To understand SQL/Persistent Stored Modules (SQL/PSM)
- To use SQL statements to create and use SQL user-defined functions
- Introduce the topic of importing Microsoft Excel 2016 data into a database table
- Introduce the topic of using Microsoft Access 2016 as an application development platform

What Is the Purpose of This Appendix?

In Chapter 3, we discussed SQL in depth. We discussed two basic categories of SQL statements: data definition language (DDL) statements, which are used for creating tables, relationships, and other structures, and data manipulation language (DML) statements, which are used for querying and modifying data.

In this appendix, which should be studied immediately after Chapter 3, we:

- Describe and illustrate additional uses of the SQL ALTER statement.
- Describe and illustrate the SQL MERGE statement.
- Describe and illustrate SQL outer join queries.
- Describe and illustrate SQL correlated subqueries.
- Describe and illustrate SQL queries on recursive relationships.
- Describe and illustrate SQL set operators.
- Describe and illustrate SQL views, which extend the DML capabilities of SQL.
- Describe and illustrate SQL Persistent Stored Modules (SQL/PSM) and create user-defined functions.

Extending the WP Database

In Chapter 3, we created and used a database for Wedgewood Pacific named WP. We will continue to use that database as the basis for our discussion in this appendix. Our first step is to extend the WP database by adding some additional tables. WP manufactures and sell consumer drones, and currently offers three models: The Alpha III, the Bravo II, and the Delta IV. While sold through various retailers, these drone models are also available directly from WP through catalog and Web site sales.

WP identifies each drone model with a stock keeping unit (SKU), which is a unique identifier for each product item that WP sells. WP maintains product data (for both current and past products) in a table name PRODUCTION_ITEM, and it maintains data that identifies what products are or were available in its annual catalogs in a table named CATALOG_SKU_20##, where ## indicates the year. This table also records when the SKU was added to the Web site, and some products may be introduced in a calendar year after the catalog itself is published. Thus, such a product will be available on the Web site, but not via the catalog. The column characteristics for the PRODUCTION_ITEM table are shown in Figure E-1, and column characteristics for the CATALOG_SKU_20## tables are shown in Figure E-2. Data for the PRODUCTION_ITEM Table is shown in Figure E-3, for the CATALOG_SKU_2015 table in Figure E-4, and for the CATALOG_SKU_2016 table in Figure E-5.

PRODUCTION_ITEM

Column Name	Type	Key	Required	Remarks
SKU	Integer	Primary Key	Yes	
SKU_Description	Char (35)	No	Yes	
ProductionStartDate	Date	No	No	
ProductionEndDate	Date	No	No	
QuantityOnHand	Integer	No	No	
QuantityInProduction	Integer	No	No	

Figure E-1 — WP Database Column Characteristics for the PRODUCTION_ITEM Table

CATALOG_SKU_20##

Column Name	Type	Key	Required	Remarks
CatalogID	Integer	Primary Key	Yes	Surrogate Key
SKU	Integer	Foreign Key	Yes	REF: PRODUCTION_ITEM
CatalogDescription	Varchar (255)	No	Yes	
CatalogPage	Integer	No	No	
DateOnWebSite	Date	No	No	

Figure E-2 — WP Database Column Characteristics for the CATALOG_SKU_20## Table

SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction
150102001	Alpha II, Black	10/15/14	11/30/15	0	0
150102005	Alpha II, White	11/15/14	10/31/15	0	0
150201001	Bravo I, Black	12/15/14	11/30/15	0	0
150201005	Bravo I, White	12/15/14	11/30/15	0	0
150303001	Delta III, Black	01/15/15	01/02/16	5	0
150303005	Delta III, White	01/15/15	01/02/16	15	0
160103001	Alpha III, Black	11/15/15	NULL	100	100
160103005	Alpha III, White	11/15/15	NULL	100	0
160202001	Bravo II, Black	12/15/15	NULL	200	100
160202005	Bravo II, White	12/15/15	NULL	150	50
160304001	Delta IV, Black	01/15/16	NULL	300	200
160304005	Delta IV, White	01/15/16	NULL	200	100

Figure E-3 — WP Database Data for the PRODUCTION_ITEM Table

CatalogID	SKU	CatalogDescription	CatalogPage	DateOnWebSite
20150001	150102001	Our low price Alpha II model in black.	10	01/01/15
20150002	150102005	Our low price Alpha II model in white.	12	01/01/15
20150003	150201001	Our new Bravo I model in black.	18	01/01/15
20150004	150201005	Our new Bravo I model in white.	20	01/01/15
20150005	150303001	Our high performance Delta III model in black.	24	01/01/15
20150006	150303005	Our high performance Delta III model in white.	26	01/01/15
20150007	160103001	New, updated Alpha III model in black.	NULL	12/01/15
20150008	160103005	New, updated Alpha III model in white.	NULL	12/01/15

Figure E-4 — WP Database Data for the CATALOG_SKU_2015 Table

CatalogID	SKU	CatalogDescription	CatalogPage	DateOnWebSite
20160001	160103001	Our low price Alpha III model in black.	10	01/01/16
20160002	160103005	Our low price Alpha III model in white.	11	01/01/16
20160003	160202001	Our new Bravo II model in black.	16	01/01/16
20160004	160202005	Our new Bravo II model in white.	17	01/01/16
20160005	160304001	Our high performance Delta IV model in black.	22	01/01/16
20160006	160304005	Our high performance Delta IV model in white.	23	01/01/16

Figure E-5 — WP Database Data for the CATALOG_SKU_2016 Table

In schema format, the tables are:

PRODUCTION_ITEM (SKU, SKU_Description, ProductionStartDate, ProductionEndDate, QuantityOnHand, QuantityInProduction)

CATALOG_SKU_2015(CatalogID, *SKU*, CatalogDescription, CatalogPage, DateOnWebsite)

CATALOG_SKU_2016(CatalogID, *SKU*, CatalogDescription, CatalogPage, DateOnWebsite)

The referential integrity constraints are:

SKU in CATALOG_SKU_2015 must exist in SKU in PRODUCTION_ITEM

SKU in CATALOG_SKU_2016 must exist in SKU in PRODUCTION_ITEM

Note that while these tables have relationships between themselves, they do not currently have relationships with any of the existing DEPARTMENT, EMPLOYEE, PROJECT, or ASSIGNMENT tables in the WP database.

Using the SQL CREATE TABLE and SQL INSERT statements that we discussed in Chapter 3, we can easily add these tables to the WP database and populate them with data. The SQL CREATE TABLE statements needed to create the tables are shown in Figure E-6, and the SQL INSERT statements needed to populate the tables are shown in Figure E-7. **However, note that we are intentionally creating the tables without the foreign key constraint between CATALOG_SKU_2016 and PRODUCTION_ITEM.**

```
CREATE  TABLE PRODUCTION ITEM(
    SKU                     Int             NOT NULL,
    SKU_Description         Char(35)        NOT NULL,
    ProductionStartDate     Date            NULL,
     ProductionEndDate      Date            NULL,
    QuantityOnHand          Int             NULL,
    QuantityInProduction    Int             NULL,
    CONSTRAINT          PRODUCTION_ITEM_PK  PRIMARY KEY(SKU)
    );

CREATE  TABLE CATALOG_SKU_2015(
    CatalogID               Int             NOT NULL IDENTITY(20150001, 1),
    SKU                     Int             NOT NULL,
    CatalogDescription      Varchar(255)    NOT NULL,
    CatalogPage             Int             NULL,
    DateOnWebSite           Date            NULL,
    CONSTRAINT          CATALOG_SKU_2015_PK PRIMARY KEY(CatalogID),
    CONSTRAINT          CAT15_PROD_ITEM_FK FOREIGN KEY(SKU)
                            REFERENCES PRODUCTION_ITEM(SKU)
                                ON UPDATE NO ACTION
                                ON DELETE NO ACTION
    );

CREATE  TABLE CATALOG_SKU_2016(
    CatalogID               Int             NOT NULL IDENTITY(20160001, 1),
    SKU                     Int             NOT NULL,
    CatalogDescription      Varchar(255)    NOT NULL,
    CatalogPage             Int             NULL,
    DateOnWebSite           Date            NULL,
    CONSTRAINT          CATALOG_SKU_2016_PK PRIMARY KEY(CatalogID)
    );
```

Figure E-6 — SQL CREATE TABLE Statements to Modify the WP Database

Note that these statements illustrate the use of the SQL Server IDENTITY Keyword to create surrogate keys in SQL Server 2016. Surrogate keys must be handled differently in Oracle Database XE and MySQL 5.7. For a discussion of how to create surrogate keys in Oracle Database XE, see Appendix B, "Getting Started in Oracle Database XE." For a discussion of how to create surrogate keys in MySQL 5.7, see Appendix C, "Getting Started with MySQL 5.7 Community Server."

The CREATE TABLE CATALOG_SKU_2016 is written in a similar manner. **Again, note that we are intentionally creating the tables without the foreign key constraint between CATALOG_SKU_2016 and PRODUCTION_ITEM regardless of the syntax needs of MySQL.**

```
/*****   PRODUCTION_ITEM DATA   ************************************************/

INSERT INTO PRODUCTION_ITEM VALUES(
    150102001, 'Alpha II, Black', '15-OCT-14', '30-NOV-15', 0, 0);
INSERT INTO PRODUCTION ITEM VALUES(
    150102005, 'Alpha II, White', '15-NOV-14',   '31-OCT-15', 0, 0);
INSERT INTO PRODUCTION ITEM VALUES(
    150201001, 'Bravo I, Black', '15-DEC-14', '30-NOV-15', 0, 0);
INSERT INTO PRODUCTION_ITEM VALUES(
    150201005, 'Bravo I, White', '15-DEC-14', '30-NOV-15', 0, 0);
INSERT INTO PRODUCTION ITEM VALUES(
    150303001, 'Delta III, Black', '15-JAN-15', '02-JAN-16', 5, 0);
INSERT INTO PRODUCTION ITEM VALUES(
    150303005, 'Delta III, White', '15-JAN-15', '02-JAN-16', 15, 0);
INSERT INTO PRODUCTION_ITEM VALUES(
    160103001, 'Alpha III, Black', '15-NOV-15', NULL, 100, 100);
INSERT INTO PRODUCTION ITEM VALUES(
    160103005, 'Alpha III, White', '15-NOV-15', NULL, 100, 0);
INSERT INTO PRODUCTION ITEM VALUES(
    160202001, 'Bravo II, Black', '15-DEC-15', NULL, 200, 100);
INSERT INTO PRODUCTION_ITEM VALUES(
    160202005, 'Bravo II, White', '15-DEC-15', NULL, 150, 50);
INSERT INTO PRODUCTION_ITEM VALUES(
    160304001, 'Delta IV, Black', '15-JAN-16', NULL, 300, 200);
INSERT INTO PRODUCTION_ITEM VALUES(
    160304005, 'Delta IV, White', '15-JAN-16', NULL, 200, 100);


/*****   CATALOG_SKU_2015   ****************************************************/

INSERT INTO CATALOG SKU 2015 VALUES(
    150102001, 'Our low price Alpha II model in black.', 10, '01-JAN-15');
INSERT INTO CATALOG_SKU_2015 VALUES(
    150102005, 'Our low price Alpha II model in white.', 12, '01-JAN-15');
INSERT INTO CATALOG_SKU_2015 VALUES(
    150201001, 'Our new Bravo I model in black.', 18, '01-JAN-15');
INSERT INTO CATALOG_SKU_2015 VALUES(
    150201005, 'Our new Bravo I model in white.', 20, '01-JAN-15');
INSERT INTO CATALOG_SKU_2015 VALUES(
    150303001, 'Our high performance Delta III model in black.', 24, '01-JAN-15');
INSERT INTO CATALOG_SKU_2015 VALUES(
    150303005, 'Our high performance Delta III model in white.', 26, '01-JAN-15');
INSERT INTO CATALOG_SKU_2015 VALUES(
    160103001, 'New, updated Alpha III model in black.', NULL, '01-DEC-15');
INSERT INTO CATALOG_SKU_2015 VALUES(
    160103005, 'New, updated Alpha III model in white.', NULL, '01-DEC-15');
```

Figure E-7 — SQL INSERT Statements to Populate the New WP Database Tables

```
/*****   CATALOG_SKU_2016   ***************************************************/

INSERT INTO CATALOG_SKU_2016 VALUES(
    160103001, 'Our low price Alpha III model in black.', 10, '01-JAN-16');
INSERT INTO CATALOG_SKU_2016 VALUES(
    160103005, 'Our low price Alpha III model in white.', 11, '01-JAN-16');
INSERT INTO CATALOG_SKU_2016 VALUES(
    160202001, 'Our new Bravo II model in black.', 18, '01-JAN-16');
INSERT INTO CATALOG_SKU_2016 VALUES(
    160202005, 'Our new Bravo II model in white.', 17, '01-JAN-16');
INSERT INTO CATALOG_SKU_2016 VALUES(
    160304001, 'Our high performance Delta IV model in black.', 22, '01-JAN-16');
INSERT INTO CATALOG_SKU_2016 VALUES(
    160304005, 'Our high performance Delta IV model in white.', 23, '01-JAN-16');
```

Figure E-7 (Continued) — SQL INSERT Statements to Populate the New WP Database Tables

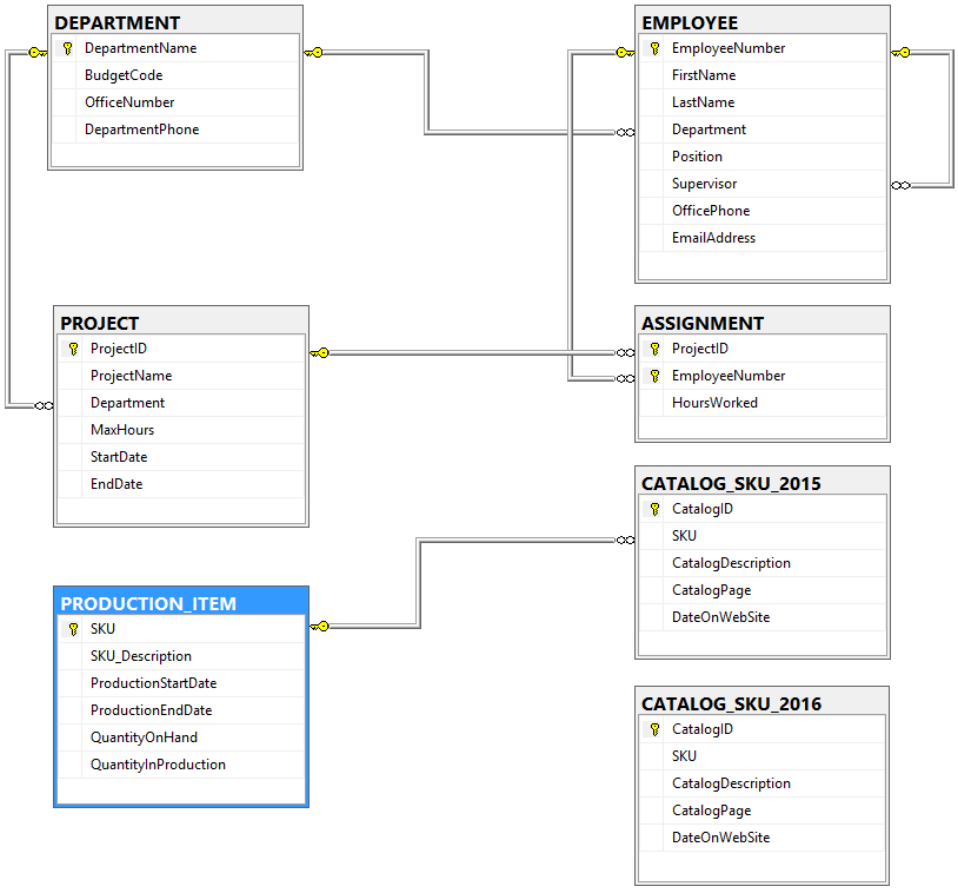


Figure E-8 — The Revised SQL Server 2016 Database Diagram for the WP Database

A revised SQL Server 2016 database diagram for the WP database is shown in Figure E-8. Note that there is no relationship between PRODUCTION_ITEM and CATALOG_SKU_2016. The data in the PRODUCTION_ITEM, CATALOG_SKU_2015, and CATALOG_SKU_2016 tables is shown in Figure E-9. Our revisions to the WP database are complete, and we will make use of these new tables in our discussion of advanced SQL features.

	SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction
1	150102001	Alpha II, Black	2014-10-15	2015-11-30	0	0
2	150102005	Alpha II, White	2014-11-15	2015-10-31	0	0
3	150201001	Bravo I, Black	2014-12-15	2015-11-30	0	0
4	150201005	Bravo I, White	2014-12-15	2015-11-30	0	0
5	150303001	Delta III, Black	2015-01-15	2016-01-02	5	0
6	150303005	Delta III, White	2015-01-15	2016-01-02	15	0
7	160103001	Alpha III, Black	2015-11-15	NULL	100	100
8	160103005	Alpha III, White	2015-11-15	NULL	100	0
9	160202001	Bravo II, Black	2015-12-15	NULL	200	100
10	160202005	Bravo II, White	2015-12-15	NULL	150	50
11	160304001	Delta IV, Black	2016-01-15	NULL	300	200
12	160304005	Delta IV, White	2016-01-15	NULL	200	100

a. Data in the New PRODUCTION_ITEM Table

	CatalogID	SKU	CatalogDescription	CatalogPage	DateOnWebSite
1	20150001	150102001	Our low price Alpha II model in black.	10	2015-01-01
2	20150002	150102005	Our low price Alpha II model in white.	12	2015-01-01
3	20150003	150201001	Our new Bravo I model in black.	18	2015-01-01
4	20150004	150201005	Our new Bravo I model in white.	20	2015-01-01
5	20150005	150303001	Our high performance Delta III model in black.	24	2015-01-01
6	20150006	150303005	Our high performance Delta III model in white.	26	2015-01-01
7	20150007	160103001	New, updated Alpha III model in black.	NULL	2015-12-01
8	20150008	160103005	New, updated Alpha III model in white.	NULL	2015-12-01

b. Data in the New CATALOG_SKU_2015 Table

	CatalogID	SKU	CatalogDescription	CatalogPage	DateOnWebSite
1	20160001	160103001	Our low price Alpha III model in black.	10	2016-01-01
2	20160002	160103005	Our low price Alpha III model in white.	11	2016-01-01
3	20160003	160202001	Our new Bravo II model in black.	18	2016-01-01
4	20160004	160202005	Our new Bravo II model in white.	17	2016-01-01
5	20160005	160304001	Our high performance Delta IV model in black.	22	2016-01-01
6	20160006	160304005	Our high performance Delta IV model in white.	23	2016-01-01

c. Data in the New CATALOG_SKU_2015 Table

Figure E-9 — Data in the New WP Database Tables

Using the SQL ALTER TABLE Statement

In Chapter 3, we introduced the **SQL ALTER TABLE statement** (see page 201), which is used to modify the structure of a table once it has been created in a database. It can be used to add, modify, and drop columns and constraints (and modify AUTO_INCREMENT in MySQL 5.7). Here we will illustrate common uses of this statement.

Adding a Column to an Existing Table

The SQL ALTER TABLE statement can be used to add a column to an existing table. We used this feature in Chapter 3's section of "The Access Workbench" (see pages 225–227), and here we will elaborate on how to do this. The basic syntax is:

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
/* *** SQL-ALTER-TABLE-AppE-01 *** */
ALTER TABLE {TableName}
   ADD {ColumnName} {DataType} {OptionalColumnConstraints};
```

Note that the **SQL COLUMN keyword** is *not* used in an **ADD {COLUMN} clause**.

If we are adding a column that allows NULL values, then we can do it with that one statement. If, however, we are adding a column with a NOT NULL column constrain, then we must use the following steps:

1. Add the column as column that allows NULL values.
2. Update all table rows with data values in the new column.
3. Modify the column to set the NOT NULL constraint.

For example, suppose that new legislation is passed requiring that any new drone being offered for sale must be approved by the FAA before going into production. The FAA checks for safety and control compliance with FAA regulations and specifications.

This means that WP will need to add a column to the PRODUCTION_ITEM table to store the date that FAA approval was received for each drone model. Further, since this approval must be received before the drone goes into production, it must be on hand when the model is added to the table, and is therefore a NOT NULL data constraint. We will assume that WP has such approvals and approval dates for all drone models currently in the table and add the column as a NOT NULL column.

Adding a NOT NULL Column to the PRODUCTION_ITEM Table

1. Add a new column named ApprovalDate as a NULL column to the PRODUCTION_ITEM table.

```
/* *** SQL-ALTER-TABLE-AppE-02 *** */
ALTER TABLE PRODUCTION_ITEM
   ADD ApprovalDate DATE NULL;
```

	SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction	ApprovalDate
1	150102001	Alpha II, Black	2014-10-15	2015-11-30	0	0	NULL
2	150102005	Alpha II, White	2014-11-15	2015-10-31	0	0	NULL
3	150201001	Bravo I, Black	2014-12-15	2015-11-30	0	0	NULL
4	150201005	Bravo I, White	2014-12-15	2015-11-30	0	0	NULL
5	150303001	Delta III, Black	2015-01-15	2016-01-02	5	0	NULL
6	150303005	Delta III, White	2015-01-15	2016-01-02	15	0	NULL
7	160103001	Alpha III, Black	2015-11-15	NULL	100	100	NULL
8	160103005	Alpha III, White	2015-11-15	NULL	100	0	NULL
9	160202001	Bravo II, Black	2015-12-15	NULL	200	100	NULL
10	160202005	Bravo II, White	2015-12-15	NULL	150	50	NULL
11	160304001	Delta IV, Black	2016-01-15	NULL	300	200	NULL
12	160304005	Delta IV, White	2016-01-15	NULL	200	100	NULL

2. Update the current rows in the PRODUCTION_ITEM table with the FAA approval dates for the existing drone models. The AprrovalDate data we will use is shown in the UPDATE statements below:

```
/* *** SQL-UPDATE-AppE-01 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-AUG-14'
   WHERE SKU = 150102001;
/* *** SQL-UPDATE-AppE-02 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-AUG-14'
   WHERE SKU = 150102005;
/* *** SQL-UPDATE-AppE-03 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-SEP-14'
   WHERE SKU = 150201001;
/* *** SQL-UPDATE-AppE-04 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-SEP-14'
   WHERE SKU = 150201005;
/* *** SQL-UPDATE-AppE-05 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-NOV-14'
   WHERE SKU = 150303001;
/* *** SQL-UPDATE-AppE-06 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-NOV-14'
   WHERE SKU = 150303005;
```

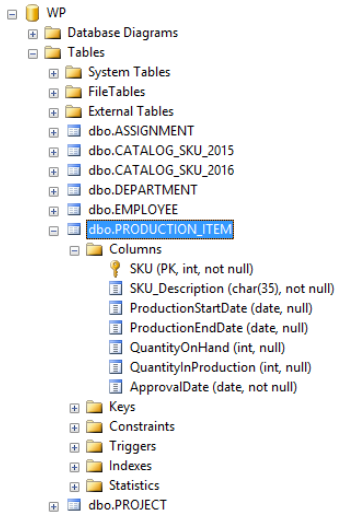
```
/* *** SQL-UPDATE-AppE-07 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-SEP-15'
   WHERE SKU = 160103001;
/* *** SQL-UPDATE-AppE-08 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-SEP-15'
   WHERE SKU = 160103005;
/* *** SQL-UPDATE-AppE-09 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-SEP-15'
   WHERE SKU = 160202001;
/* *** SQL-UPDATE-AppE-10 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-SEP-15'
   WHERE SKU = 160202005;
/* *** SQL-UPDATE-AppE-11 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-NOV-15'
   WHERE SKU = 160304001;
/* *** SQL-UPDATE-AppE-12 *** */
UPDATE   PRODUCTION_ITEM
   SET   ApprovalDate = '30-NOV-15'
   WHERE SKU = 160304005;
```

	SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction	ApprovalDate
1	150102001	Alpha II, Black	2014-10-15	2015-11-30	0	0	2014-08-30
2	150102005	Alpha II, White	2014-11-15	2015-10-31	0	0	2014-08-30
3	150201001	Bravo I, Black	2014-12-15	2015-11-30	0	0	2014-09-30
4	150201005	Bravo I, White	2014-12-15	2015-11-30	0	0	2014-09-30
5	150303001	Delta III, Black	2015-01-15	2016-01-02	5	0	2014-11-30
6	150303005	Delta III, White	2015-01-15	2016-01-02	15	0	2014-11-30
7	160103001	Alpha III, Black	2015-11-15	NULL	100	100	2015-09-30
8	160103005	Alpha III, White	2015-11-15	NULL	100	0	2015-09-30
9	160202001	Bravo II, Black	2015-12-15	NULL	200	100	2015-09-30
10	160202005	Bravo II, White	2015-12-15	NULL	150	50	2015-09-30
11	160304001	Delta IV, Black	2016-01-15	NULL	300	200	2015-11-30
12	160304005	Delta IV, White	2016-01-15	NULL	200	100	2015-11-30

3. Modify the ApprovalDate column to set the NOT NULL column constraint.

```
/* *** SQL-ALTER-TABLE-AppE-03 *** */
ALTER TABLE PRODUCTION_ITEM
   ALTER COLUMN ApprovalDate DATE NOT NULL;
```

Note that the SQL COLUMN keyword *is* used in an **ALTER {COLUMN} clause**. Also note that while SQL Server uses the ALTER COLUMN clause, Oracle Database and MySQL use the *MODIFY COLUMN clause*. Looking at the column descriptions for the PRODUCTION_ITEM table, we can see that the ApprovalDate column is now set to NOT NULL.



Modifying the MySQL AUTO_INCREMENT Settings

The SQL ALTER TABLE statement can be used to modify the MySQL AUTO_INCREMENT starting value (but not the increment value, which always has a value of 1). This modification should be immediately after an SQL CREATE TABLE statement is run, and before any data is inserted into the table so that the first value of the surrogate key will have the intended value (in this case 20150001):

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-ALTER-TABLE-AppE-04 *** */
ALTER TABLE CATALOG_SKU_2015 AUTO_INCREMENT 20150001;
```

Adding a Table Constraint to an Existing Table

The SQL ALTER TABLE statement can be used to add a table constraint to an existing table. The basic syntax is:

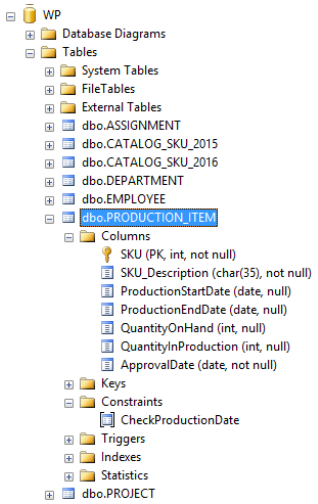
```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-ALTER-TABLE-AppE-05 *** */
ALTER TABLE {TableName}
   ADD CONSTRAINT {ConstraintName} {TableConstraint};
```

Note that the **SQL CONSTRAINT keyword** *is* used in an **ADD CONSTRAINT clause**.

A typical use here might be to add an **SQL CHECK constraint** to a table. In our WP database example, the data value in ApprovalDate in each row in the table must be less than the data value in ProductionStartDate in the same row (FAA approval must be on hand before production of a specific drone model is started). We can do that by using the SQL ALTER TABLE statement:

```
/* *** SQL-ALTER-TABLE-AppE-06 *** */
ALTER TABLE PRODUCTION_ITEM
   ADD CONSTRAINT CheckProductionDate CHECK
                     (ApprovalDate < ProductionStartDate);
```

Note that the SQL CONSTRAINT keyword *is* used in an **ALTER CONSTRAINT clause**. Looking at the constraints folder for the PRODUCTION_ITEM table, we can see that the CheckProductionDate constraint has been added.



Adding a Referential Integrity Constraint to an Existing Table

The SQL ALTER TABLE statement can be used to add a referential integrity constraint to an existing table. The basic syntax is:

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
/* *** SQL-ALTER-TABLE-AppE-07 *** */
ALTER TABLE {TableName}
   ADD CONSTRAINT {ConstraintName} FOREIGN KEY({ColumnName})
      REFERENCES {TableName}({PrimaryKeyColumn})
         {Optional ON UPDATE clause}
         {Optional ON DELETE clause};
```

Note that the SQL CONSTRAINT keyword *is* used in an **ADD CONSTRAINT clause**.

In our WP database example, we still need to add a referential integrity constraint to the CATALOG_SKU_2016 table to establish the relationship between that table and the PRODUCTION_ITEM table. We can do that by using the SQL ALTER TABLE statement:

```
/* *** SQL-ALTER-TABLE-AppE-08 *** */
ALTER TABLE CATALOG_SKU_2016
   ADD CONSTRAINT    CAT16_PROD_ITEM_FK FOREIGN KEY(SKU)
      REFERENCES PRODUCTION_ITEM(SKU)
         ON UPDATE NO ACTION
         ON DELETE NO ACTION;
```

Note that because SKU, the primary key of PRODUCTION_ITEM, is a surrogate key, it will never be changed, and therefore no update actions will be necessary. And because WP wants to retain data in the CATALOG_SKU_20## tables for historical records, we will not allow deletion of an SKU from the PRODUCTION_ITEM table.

After running the ALTER TABLE statement, the relationship between PRODUCTION_ITEM and CATALOG_SKU_2016 can be seen in Figure E-10. Note that the added ApprovalDate column also appears in the PRODUCTION_ITEM table. Compare database diagram in Figure E-10 to the one in Figure E-8.

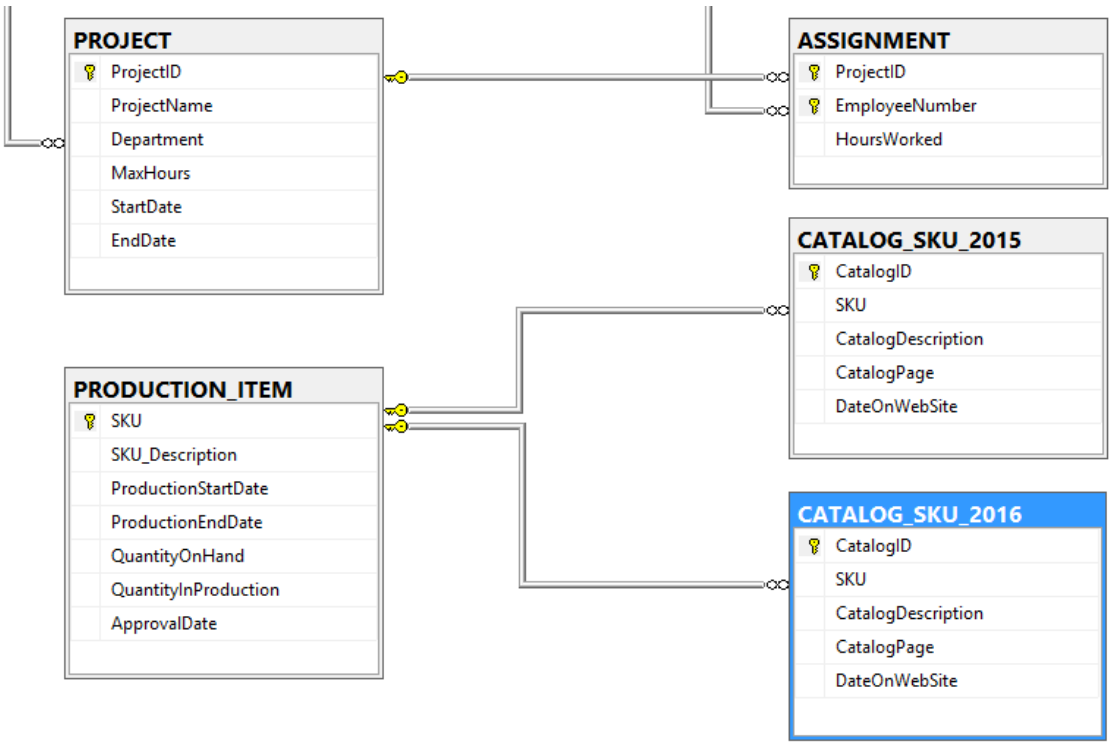


Figure E-10 — The Database Diagram for the WP Database with the Added Relationship

Removing a Table Column from an Existing Table

The SQL ALTER TABLE statement can be used to remove a column from a table. Note that the SQL COLUMN keyword *is* used in the **DROP COLUMN clause**. The basic syntax is:

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
/* *** SQL-ALTER-TABLE-AppE-09 *** */
ALTER TABLE {TableName}
   DROP COLUMN {ColumnName};
```

Removing a Table Constraint from an Existing Table

The SQL ALTER TABLE statement can be used to remove a table constraint. Note that the **SQL CONSTRAINT keyword** *is* used in the **DROP CONSTRAINT clause**. The basic syntax is:

```
/* *** EXAMPLE CODE – DO NOT RUN *** */
/* *** SQL-ALTER-TABLE-AppE-10 *** */
ALTER TABLE {TableName}
   DROP CONSTRAINT {ConstraintName};
```

Note that you *cannot* drop a table that is the parent table in a referential integrity constraint. In this case, you would use an ALTER TABLE DROP CONSTRAINT statement to drop the referential integrity constraint between tables, and then drop one or both of the tables themselves.

Using the SQL MERGE Statement

In Chapter 3, we mentioned the **SQL MERGE statement** (see page 199), which combines the capabilities of the SQL INSERT and SQL UPDATE statement into one statement that can selectively insert or update data depending upon what data is already in a table.

To use the MERGE statement, the new data is typically stored in a table of a similar structure to the table being updated. For the WP database, we will construct a table named PRODUCT_ITEM_DATA, which has the same basic structure as the PRODUCT_ITEM table in Figure E-1 as modified with the ApprovalDate column. However, in the PRODUCT_ITEM_DATA table every column except the primary key column SKU allows NULLs, since only some of these columns may be in use at any time. The SQL CREATE TABLE statement to create this table is shown in Figure E-11.

```
CREATE  TABLE PRODUCTION_ITEM_DATA(
    SKU                     Int                 NOT NULL,
    SKU_Description         Char(35)            NULL,
    ProductionStartDate     Date                NULL,
    ProductionEndDate       Date                NULL,
    QuantityOnHand          Int                 NULL,
    QuantityInProduction    Int                 NULL,
    ApprovalDate            Date                NULL,
    CONSTRAINT          PROD_ITEM_PK            PRIMARY KEY(SKU)
    );
```

Figure E-11 — SQL CREATE TABLE Statement to Create the PRODUCTION_ITEM_DATA Table

To understand why we would use this table, consider database administration permissions that may be used at Wedgewood Pacific (database administration is discussed in Chapter 6). Many people may have permissions to add data to the PRODUCTION_ITEM_DATA table, but only the actual DBA (database administrator) himself or herself may have permissions to update the PRODUCTION_ITEM table. It will be only the DBA who uses the SQL MERGE statement, and only after the data in the PRODUCTION_ITEM_DATA has been checked and verified.

For example, suppose that employees at Wedgewood Pacific have put the following data into the PRODUCTION_ITEM_DATA table:

1. SKU 160103001 (Alpha III, Black) ended its production run on October 31st, 2016.
2. SKU 160103005 (Alpha III, White) ended its production run on October 31st, 2016.
3. A new model, the Alpha IV, was put into production on November 15th, 2016. It is available in both black (SKU 170104001) and white (SKU 170104005). There is no stock on hand, but there are 200 of each SKU in production. The FAA approval was received on October 15th, 2016.

The SQL statements to populate the PRODUCTION_ITEM_DATA table are shown in Figure E-12, and the data in the PRODUCTION_ITEM_DATA table appears as:

	SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction	ApprovalDate
1	160103001	NULL	NULL	2016-10-31	NULL	NULL	NULL
2	160103005	NULL	NULL	2016-10-31	NULL	NULL	NULL
3	170104001	Alpha IV, Black	2016-11-15	NULL	0	200	2016-10-15
4	170104005	Alpha IV, White	2016-11-15	NULL	0	200	2016-10-15

Note that the rows for SKU 160103001 and SKU 160103005 hold update data, while the rows for SKU 170104001 and SKU 170104005 hold new data to be inserted as new rows into the PRODUCTION_DATA table. The SQL MERGE command tests a specified condition to see if the data in a row is to be used for an update operation or an insert operation and then acts accordingly.

```
INSERT INTO PRODUCTION_ITEM_DATA(SKU, ProductionEndDate)
    VALUES(160103001, '31-OCT-16');
INSERT INTO PRODUCTION_ITEM_DATA(SKU, ProductionEndDate)
    VALUES(160103005, '31-OCT-16');
INSERT INTO PRODUCTION_ITEM_DATA VALUES(
    170104001, 'Alpha IV, Black', '15-NOV-16', NULL, 0, 200, '15-OCT-16');
INSERT INTO PRODUCTION_ITEM_DATA VALUES(
    170104005, 'Alpha IV, White', '15-NOV-16', NULL, 0, 200, '15-OCT-16');
```

Figure E-12 — SQL Statements to Add Data to the PRODUCTION_ITEM_DATA Table

The MERGE statement the WP DBA will use is:

```
/* *** SQL-MERGE-AppE-01 *** */
MERGE INTO PRODUCTION_ITEM AS PROI USING PRODUCTION_ITEM_DATA AS PROID
      ON    PROI.SKU = PROID.SKU
      WHEN MATCHED THEN
         UPDATE   SET PROI.ProductionEndDate = PROID.ProductionEndDate
      WHEN NOT MATCHED THEN
         INSERT   (SKU, SKU_Description,
                   ProductionStartDate, ProductionEndDate,
                   QuantityOnHand, QuantityInProduction,
                   ApprovalDate)
                  VALUES (PROID.SKU, PROID.SKU_Description,
                   PROID.ProductionStartDate, PROID.ProductionEndDate,
                   PROID.QuantityOnHand, PROID.QuantityInProduction,
                   PROID.ApprovalDate);
```

Note the we check to see if there are matching SKU values in the two tables. If there are matching SKUs, we run an UPDATE because the SKU already exists in the PRODUCTION_ITEM table. In this case, the updated ProductionEndDate data will be added to the PRODUCTION_ITEM table. On the other hand, if there are unmatched SKUs, then we run an INSERT because the new SKU data must to added to the PRODUCTION_ITEM table. In this case, we use in a **bulk SQL INSERT** statement—a list of the columns that will be populated with the new data is matched with a list of the source of the data for each of those columns. Not that in a bulk SQL Insert statement, the VALUES clause can be replaced with a SELECT statement that determines the data to be inserted—see Appendix A pages A-55–A-56 for an example of this construction.

Here is the PRODUCTION_ITEM table before the MERGE statement is run:

	SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction	ApprovalDate
1	150102001	Alpha II, Black	2014-10-15	2015-11-30	0	0	2014-08-30
2	150102005	Alpha II, White	2014-11-15	2015-10-31	0	0	2014-08-30
3	150201001	Bravo I, Black	2014-12-15	2015-11-30	0	0	2014-09-30
4	150201005	Bravo I, White	2014-12-15	2015-11-30	0	0	2014-09-30
5	150303001	Delta III, Black	2015-01-15	2016-01-02	5	0	2014-11-30
6	150303005	Delta III, White	2015-01-15	2016-01-02	15	0	2014-11-30
7	160103001	Alpha III, Black	2015-11-15	NULL	100	100	2015-09-30
8	160103005	Alpha III, White	2015-11-15	NULL	100	0	2015-09-30
9	160202001	Bravo II, Black	2015-12-15	NULL	200	100	2015-09-30
10	160202005	Bravo II, White	2015-12-15	NULL	150	50	2015-09-30
11	160304001	Delta IV, Black	2016-01-15	NULL	300	200	2015-11-30
12	160304005	Delta IV, White	2016-01-15	NULL	200	100	2015-11-30

Here is the PRODUCTION_ITEM table after the MERGE statement is run:

	SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction	ApprovalDate
1	150102001	Alpha II, Black	2014-10-15	2015-11-30	0	0	2014-08-30
2	150102005	Alpha II, White	2014-11-15	2015-10-31	0	0	2014-08-30
3	150201001	Bravo I, Black	2014-12-15	2015-11-30	0	0	2014-09-30
4	150201005	Bravo I, White	2014-12-15	2015-11-30	0	0	2014-09-30
5	150303001	Delta III, Black	2015-01-15	2016-01-02	5	0	2014-11-30
6	150303005	Delta III, White	2015-01-15	2016-01-02	15	0	2014-11-30
7	160103001	Alpha III, Black	2015-11-15	2016-10-31	100	100	2015-09-30
8	160103005	Alpha III, White	2015-11-15	2016-10-31	100	0	2015-09-30
9	160202001	Bravo II, Black	2015-12-15	NULL	200	100	2015-09-30
10	160202005	Bravo II, White	2015-12-15	NULL	150	50	2015-09-30
11	160304001	Delta IV, Black	2016-01-15	NULL	300	200	2015-11-30
12	160304005	Delta IV, White	2016-01-15	NULL	200	100	2015-11-30
13	170104001	Alpha IV, Black	2016-11-15	NULL	0	200	2016-10-15
14	170104005	Alpha IV, White	2016-11-15	NULL	0	200	2016-10-15

Note that the SQL MERGE statement has done the updates and inserts correctly.

Extensions to SQL Query Techniques

In Chapter 3, we introduced, defined, and discussed **SQL queries**. We will now extend that material with some additional discussion of what can be done with an SQL query by expanding the coverage of outer joins and then introducing correlated subqueries and queries on recursive relationships.

Using Outer Join Queries

In Chapter 3, we introduced, defined, and discussed **SQL outer join** queries (see pages 193–197), and you should review the material presented there before continuing with the examples presented here. In this section, we will extend the discussion in Chapter 3 by providing examples of SQL outer joins in the WP database.

In Chapter 3, we added a new record to the WP database PROJECT table:

```
/* *** EXAMPLE CODE – DO NOT RUN – USED IN CHAPTER 3*** */
/* *** SQL-INSERT-CH03-05 *** */
INSERT INTO PROJECT
      (ProjectName, Department, MaxHours, StartDate)
      VALUES('2017 Q4 Tax Preparation', 'Accounting', 175.00,
      '10-DEC-17');
```

The added row has ProjectID 1700, and can be clearly seen by running SQL-Query-AppE-01 (which duplicates SQL-Query-CH03-52):

```
/* *** SQL-Query-AppE-01 *** */
SELECT * FROM PROJECT;
```

The result of SQL-Query-AppE-01 is:

	ProjectID	ProjectName	Department	MaxHours	StartDate	EndDate
1	1000	2017 Q3 Production Plan	Production	100.00	2017-05-10	2017-06-15
2	1100	2017 Q3 Marketing Plan	Sales and Marketing	135.00	2017-05-10	2017-06-15
3	1200	2017 Q3 Portfolio Analysis	Finance	120.00	2017-07-05	2017-07-25
4	1300	2017 Q3 Tax Preparation	Accounting	145.00	2017-08-10	2017-10-15
5	1400	2017 Q4 Production Plan	Production	100.00	2017-08-10	2017-09-15
6	1500	2017 Q4 Marketing Plan	Sales and Marketing	135.00	2017-08-10	2017-09-15
7	1600	2017 Q4 Portfolio Analysis	Finance	140.00	2017-10-05	NULL
8	1700	2017 Q4 Tax Preparation	Accounting	175.00	2017-12-10	NULL

We then used the updated PROJECT in a JOIN ON query:

```
/* *** SQL-Query-CH03-53 *** */
SELECT      ProjectName, FirstName, LastName, HoursWorked
FROM        EMPLOYEE AS E JOIN ASSIGNMENT AS A
      ON    E.EmployeeNumber = A.EmployeeNumber
            JOIN PROJECT AS P
                  ON    A.ProjectID = P.ProjectID
ORDER BY    P.ProjectID, A.EmployeeNumber;
```

The result of this query (preciously shown in Figure 3-25) is shown in Figure E-13.

	ProjectName	FirstName	LastName	HoursWorked
1	2017 Q3 Production Plan	Mary	Jacobs	30.00
2	2017 Q3 Production Plan	Ken	Evans	50.00
3	2017 Q3 Production Plan	Ken	Numoto	50.00
4	2017 Q3 Production Plan	Mary	Smith	75.00
5	2017 Q3 Production Plan	Tom	Jackson	75.00
6	2017 Q3 Marketing Plan	Mary	Jacobs	30.00
7	2017 Q3 Marketing Plan	Ken	Evans	75.00
8	2017 Q3 Marketing Plan	Ken	Numoto	55.00
9	2017 Q3 Marketing Plan	Linda	Granger	55.00
10	2017 Q3 Portfolio Analysis	Richard	Bandalone	20.00
11	2017 Q3 Portfolio Analysis	Ken	Evans	40.00
12	2017 Q3 Portfolio Analysis	Mary	Abernathy	45.00
13	2017 Q3 Portfolio Analysis	Tom	Caruthers	45.00
14	2017 Q3 Tax Preparation	Richard	Bandalone	25.00
15	2017 Q3 Tax Preparation	Ken	Evans	40.00
16	2017 Q3 Tax Preparation	Tom	Caruthers	50.00
17	2017 Q3 Tax Preparation	Heather	Jones	50.00
18	2017 Q4 Production Plan	Mary	Jacobs	30.00
19	2017 Q4 Production Plan	Ken	Evans	50.00
20	2017 Q4 Production Plan	Ken	Numoto	50.00
21	2017 Q4 Production Plan	Mary	Smith	75.00
22	2017 Q4 Production Plan	Tom	Jackson	75.00
23	2017 Q4 Marketing Plan	Mary	Jacobs	30.00
24	2017 Q4 Marketing Plan	Ken	Evans	75.00
25	2017 Q4 Marketing Plan	Ken	Numoto	55.00
26	2017 Q4 Marketing Plan	Linda	Granger	55.00
27	2017 Q4 Portfolio Analysis	Richard	Bandalone	20.00
28	2017 Q4 Portfolio Analysis	Ken	Evans	40.00
29	2017 Q4 Portfolio Analysis	Mary	Abernathy	45.00
30	2017 Q4 Portfolio Analysis	Tom	Caruthers	45.00

Figure E-13 — The Results for SQL-Query-CH03-53

The problem with this query is that there is no data about the new project with ProjectID 1700! But now consider the following SQL query statement—which simplifies SQL Query-CH03-53 by retrieving employee numbers instead of employee names and thus eliminates the need to include the EMPLOYEE table in the query—and notice the use of the JOIN ON syntax—the **SQL LEFT keyword** is simply added to the SQL query:

```
/* *** SQL-Query-AppE-02 *** */
SELECT      ProjectName, EmployeeNumber, HoursWorked
FROM        PROJECT LEFT JOIN ASSIGNMENT
      ON    PROJECT.ProjectID = ASSIGNMENT.ProjectID
ORDER BY    PROJECT.ProjectID;
```

The purpose of this join is to append rows of PROJECT to those of ASSIGNMENT, as in SQL_Query-CH03-53, except that if any row in the table on the left side of the FROM clause (in this case, PROJECT) has no match, the data from that row are now included in the results anyway. The result of this query is shown in Figure E-14. Notice that the last row of this table shows the new 2017 Q4 Tax Preparation project, and SQL appends NULL values for the EmployeeNumber and HoursWorked columns because there are no employees assigned to this new project.

	ProjectName	EmployeeNumber	HoursWorked
1	2017 Q3 Production Plan	1	30.00
2	2017 Q3 Production Plan	6	50.00
3	2017 Q3 Production Plan	10	50.00
4	2017 Q3 Production Plan	16	75.00
5	2017 Q3 Production Plan	17	75.00
6	2017 Q3 Marketing Plan	1	30.00
7	2017 Q3 Marketing Plan	6	75.00
8	2017 Q3 Marketing Plan	10	55.00
9	2017 Q3 Marketing Plan	11	55.00
10	2017 Q3 Portfolio Analysis	3	20.00
11	2017 Q3 Portfolio Analysis	6	40.00
12	2017 Q3 Portfolio Analysis	7	45.00
13	2017 Q3 Portfolio Analysis	8	45.00
14	2017 Q3 Tax Preparation	3	25.00
15	2017 Q3 Tax Preparation	6	40.00
16	2017 Q3 Tax Preparation	8	50.00
17	2017 Q3 Tax Preparation	9	50.00
18	2017 Q4 Production Plan	1	30.00
19	2017 Q4 Production Plan	6	50.00
20	2017 Q4 Production Plan	10	50.00
21	2017 Q4 Production Plan	16	75.00
22	2017 Q4 Production Plan	17	75.00
23	2017 Q4 Marketing Plan	1	30.00
24	2017 Q4 Marketing Plan	6	75.00
25	2017 Q4 Marketing Plan	10	55.00
26	2017 Q4 Marketing Plan	11	55.00
27	2017 Q4 Portfolio Analysis	3	20.00
28	2017 Q4 Portfolio Analysis	6	40.00
29	2017 Q4 Portfolio Analysis	7	45.00
30	2017 Q4 Portfolio Analysis	8	45.00
31	2017 Q4 Tax Preparation	NULL	NULL

Figure E-14 — The Results for SQL-Query-AppE-02

Right outer joins operate similarly, except that the **SQL RIGHT keyword** is used, and rows in the table on the right-hand side of the FROM clause are included. For example, you could join all three tables together with the following modification of SQL-Query-CH03-53, which includes a right outer join:

```
/* *** SQL-Query-AppE-03 *** */
SELECT      ProjectName, FirstName, LastName, HoursWorked
FROM        (PROJECT AS P JOIN ASSIGNMENT AS A
      ON    P.ProjectID = A.ProjectID)
            RIGHT JOIN EMPLOYEE AS E
                  ON    A.EmployeeNumber = E.EmployeeNumber
ORDER BY    P.ProjectID, A.EmployeeNumber;
```

The result of this join, which now shows not only the employees assigned to projects but also those employees who are *not* assigned to any projects, is shown in Figure E-15.

	ProjectName	FirstName	LastName	HoursWorked
1	NULL	Rosalie	Jackson	NULL
2	NULL	George	Smith	NULL
3	NULL	Alan	Adams	NULL
4	NULL	James	Nestor	NULL
5	NULL	Rick	Brown	NULL
6	NULL	Mike	Nguyen	NULL
7	NULL	Jason	Sleeman	NULL
8	NULL	George	Jones	NULL
9	NULL	Julia	Hayakawa	NULL
10	NULL	Sam	Stewart	NULL
11	2017 Q3 Production Plan	Mary	Jacobs	30.00
12	2017 Q3 Production Plan	Ken	Evans	50.00
13	2017 Q3 Production Plan	Ken	Numoto	50.00
14	2017 Q3 Production Plan	Mary	Smith	75.00
15	2017 Q3 Production Plan	Tom	Jackson	75.00
16	2017 Q3 Marketing Plan	Mary	Jacobs	30.00
17	2017 Q3 Marketing Plan	Ken	Evans	75.00
18	2017 Q3 Marketing Plan	Ken	Numoto	55.00
19	2017 Q3 Marketing Plan	Linda	Granger	55.00
20	2017 Q3 Portfolio Analysis	Richard	Bandalone	20.00
21	2017 Q3 Portfolio Analysis	Ken	Evans	40.00
22	2017 Q3 Portfolio Analysis	Mary	Abernathy	45.00
23	2017 Q3 Portfolio Analysis	Tom	Caruthers	45.00
24	2017 Q3 Tax Preparation	Richard	Bandalone	25.00
25	2017 Q3 Tax Preparation	Ken	Evans	40.00
26	2017 Q3 Tax Preparation	Tom	Caruthers	50.00
27	2017 Q3 Tax Preparation	Heather	Jones	50.00
28	2017 Q4 Production Plan	Mary	Jacobs	30.00
29	2017 Q4 Production Plan	Ken	Evans	50.00
30	2017 Q4 Production Plan	Ken	Numoto	50.00
31	2017 Q4 Production Plan	Mary	Smith	75.00
32	2017 Q4 Production Plan	Tom	Jackson	75.00
33	2017 Q4 Marketing Plan	Mary	Jacobs	30.00
34	2017 Q4 Marketing Plan	Ken	Evans	75.00
35	2017 Q4 Marketing Plan	Ken	Numoto	55.00
36	2017 Q4 Marketing Plan	Linda	Granger	55.00
37	2017 Q4 Portfolio Analysis	Richard	Bandalone	20.00
38	2017 Q4 Portfolio Analysis	Ken	Evans	40.00
39	2017 Q4 Portfolio Analysis	Mary	Abernathy	45.00
40	2017 Q4 Portfolio Analysis	Tom	Caruthers	45.00

Figure E-15 — The Results for SQL-Query-AppE-03

Does Not Work with Microsoft Access ANSI-89 SQL

Even with the following syntax, which is what worked before in Microsoft Access, the error message *Join expression not supported* is returned when the query is run.

```
/* *** SQL-Query-AppE-03-Access *** */
SELECT      ProjectName, FirstName, LastName, HoursWorked
FROM        (PROJECT AS P JOIN ASSIGNMENT AS A
      ON    P.ProjectID = A.ProjectID)
               RIGHT JOIN EMPLOYEE AS E
                  ON    A.EmployeeNumber = E.EmployeeNumber
ORDER BY    P.ProjectID, A.EmployeeNumber;
```

Solution: Build an equivalent query or set of queries using the Microsoft Access Query by Example (QBE). Query by Example (QBE) is discussed in Chapter 3's section of "The Access Workbench."

Using Correlated Subqueries

In Chapter 3, we introduced and used SQL subqueries (see pages 183–185 and 93) and mentioned the **SQL correlated subquery**, which can do work that is not possible with join queries. A correlated subquery looks very much like the noncorrelated subqueries we discussed in Chapter 3, but, in actuality, correlated subqueries are very different. To understand the difference, consider the following noncorrelated subquery, which is like those in Chapter 3, and lists the FirstName and LastName of all WP employees assigned to work on the project with ProjectID 1000:

```
/* *** SQL-Query-AppE-04 *** */
SELECT   E.FirstName, E.lastName
FROM     EMPLOYEE AS E
WHERE    EmployeeNumber IN
         (SELECT  A.EmployeeNumber
         FROM     ASSIGNMENT AS A
         WHERE    A.ProjectID = 1000);
```

The DBMS can process such subqueries from the bottom up—that is, it can first find all of the values of EmployeeNumber in ASSIGNMENT that have the ProjectID of 1000 and then process the upper query using that set of values. There is no need to move back and forth between the two SELECT statements. The result of this query is:

	FirstName	lastName
1	Mary	Jacobs
2	Ken	Evans
3	Ken	Numoto
4	Mary	Smith
5	Tom	Jackson

Searching for Multiple Rows with a Given Value

Now, to introduce correlated subqueries, suppose that someone at Wedgewood Pacific proposes that the LastName column of EMPLOYEE be used as a candidate key which can be used in place of the primary key EmployeeNumber to uniquely identify each row in the table (see the discussion of candidate and primary keys in Chapter 2 on pages 74–79). If you look at the data in the EMPLOYEE table shown here, you can easily see that there are two occurrences in LastName for *Jackson*, *Jones* and *Smith*. Therefore, LastName cannot be a candidate key, and we can determine this by simply looking at the dataset.

	EmployeeNumber	FirstName	LastName	Department	Position	Supervisor	OfficePhone	EmailAddress
1	1	Mary	Jacobs	Administration	CEO	NULL	360-285-8110	Mary.Jacobs@WP.com
2	2	Rosalie	Jackson	Administration	Admin Assistant	1	360-285-8120	Rosalie.Jackson@WP.com
3	3	Richard	Bandalone	Legal	Attorney	1	360-285-8210	Richard.Bandalone@WP.com
4	4	George	Smith	Human Resources	HR3	1	360-285-8310	George.Smith@WP.com
5	5	Alan	Adams	Human Resources	HR1	4	360-285-8320	Alan.Adams@WP.com
6	6	Ken	Evans	Finance	CFO	1	360-285-8410	Ken.Evans@WP.com
7	7	Mary	Abernathy	Finance	FA3	6	360-285-8420	Mary.Abernathy@WP.com
8	8	Tom	Caruthers	Accounting	FA2	6	360-285-8430	Tom.Caruthers@WP.com
9	9	Heather	Jones	Accounting	FA2	6	360-285-8440	Heather.Jones@WP.com
10	10	Ken	Numoto	Sales and Marketing	SM3	1	360-285-8510	Ken.Numoto@WP.com
11	11	Linda	Granger	Sales and Marketing	SM2	10	360-285-8520	Linda.Granger@WP.com
12	12	James	Nestor	InfoSystems	CIO	1	360-285-8610	James.Nestor@WP.com
13	13	Rick	Brown	InfoSystems	IS2	12	360-285-8620	Rick.Brown@WP.com
14	14	Mike	Nguyen	Research and Development	CTO	1	360-285-8710	Mike.Nguyen@WP.com
15	15	Jason	Sleeman	Research and Development	RD3	14	360-285-8720	Jason.Sleeman@WP.com
16	16	Mary	Smith	Production	OPS3	1	360-285-8810	Mary.Smith@WP.com
17	17	Tom	Jackson	Production	OPS2	14	360-285-8820	Tom.Jackson@WP.com
18	18	George	Jones	Production	OPS2	15	360-285-8830	George.Jones@WP.com
19	19	Julia	Hayakawa	Production	OPS1	15	NULL	Julia.Hayakawa@WP.com
20	20	Sam	Stewart	Production	OPS1	15	NULL	Sam.Stewart@WP.com

However, if the EMPLOYEE table had 10,000 or more rows, this would be difficult to determine. In that case, we need a query that examines the EMPLOYEE table and displays the Employee Number, LastName, and FirstName of any employees that share the same last name.

If we were asked to write a program to perform such a query, our logic would be as follows: Take the value of LastName from the first row in EMPLOYEE and examine all of the other rows in the table. If we find a row that has the same last name as the one in the first row, we know there are duplicates, so we print the EmployeeNumber, LastName, and FirstName of the first row. We continue searching for duplicate LastName values until we come to the end of the EMPLOYEE table.

Next we take the value of LastName in the second row and compare it with all other rows in the EMPLOYEE table, printing out the EmployeeNumber, LastName, and FirstName of any duplicate names. We proceed in this way until all rows of EMPLOYEE have been examined.

A Correlated Subquery That Finds Rows with the Same Value

The following correlated subquery performs the action just described:

```
/* *** SQL-Query-AppE-05 *** */
SELECT      E1.EmployeeNumber, E1.FirstName, E1.LastName
FROM        EMPLOYEE AS E1
WHERE       E1.LastName IN
            (SELECT     E2.LastName
             FROM       EMPLOYEE AS E2
             WHERE      E1.LastName = E2.LastName
                AND     E1.EmployeeNumber <> E2.EmployeeNumber);
```

The result of this query for the data in the EMPLOYEE table is:

	EmployeeNumber	FirstName	LastName
1	2	Rosalie	Jackson
2	4	George	Smith
3	9	Heather	Jones
4	16	Mary	Smith
5	17	Tom	Jackson
6	18	George	Jones

Looking at these results, it is easy to see the nonunique, duplicated LastName data that prevents LastName from being used as an alternate key. This subquery, which is a *correlated subquery*, looks deceptively similar to a regular, *non-correlated subquery*. To the surprise of many students, this subquery and the one above are drastically different. Their similarity is only superficial.

Before learning why, first notice the notation in the correlated subquery. The EMPLOYEE table is used in both the upper and the lower SELECT statements. In the upper statement, it is given the alias E1; in the lower SELECT statement, it is given the alias E2. In essence, when we use this notation, it is as if we have made *two copies* of the EMPLOYEE table. One copy is called E1, and the second copy is called E2. Therefore, in the last two lines of the correlated subquery, values in the E1 copy of EMPLOYEE are compared with values in the E2 copy.

Now consider what makes this subquery so different. Unlike with a regular, noncorrelated subquery, the DBMS cannot run the bottom SELECT by itself, obtain a set of LastNames, and then use that set to execute the upper query. The reason for this appears in the last two lines of the query:

```
WHERE     E1.LastName = E2.LastName
   AND    E1.EmployeeNumber <> E2.EmployeeNumber);
```

In these expressions, E1.LastName (from the top SELECT statement) is being compared with E2.LastName (from the bottom SELECT statement). The same is true for E1.EmployeeNumber and E2.EmployeeNumber. Because of this fact, the DBMS cannot process the subquery portion independently of the upper SELECT.

Instead, the DBMS must process this statement as a subquery that is nested within the main query. The logic is as follows: Take the first row from E1. Using that row, evaluate the second query. To do that, for each row in E2, compare E1.LastName with E2.LastName and E1.EmployeeNumber with E2.EmployeeNumber. If the last names are equal and the values of EmployeeNumber are *not* equal, return the value of E2.LastName to the upper query. Do this for every row in E2.

Once all of the rows in E2 have been evaluated for the first row in E1, move to the second row in E1 and evaluate it against all the rows in E2. Continue in this way until all rows of E1 have been compared with all of the rows of E2.

If this is not clear to you, write out two copies of the EMPLOYEE data on a piece of scratch paper. Label one of them E1 and the second E2, and then work through the logic as described. From this, you will see that correlated subqueries always require nested processing.

A Common Trap

By the way, do not fall into the following common trap:

```
/* *** SQL-Query-AppE-06 *** */
SELECT      E1.EmployeeNumber, E1.FirstName, E1.LastName
FROM        EMPLOYEE AS E1
WHERE       E1.EmployeeNumber IN
            (SELECT  E2.EmployeeNumber
             FROM       EMPLOYEE AS E2
             WHERE      E1.LastName = E2.LastName
                AND     E1.EmployeeNumber <> E2.EmployeeNumber);
```

The logic here seems correct, but it is not. Compare SQL-Query-AppE-06 to SQL-Query-AppE-05, and note the differences between the two SQL statements. The result of SQL-Query-CH08-06 when run on the WP data is an empty set:

	EmployeeNumber	FirstName	LastName

In fact, no row will ever be displayed by this query, regardless of the underlying data (see if you can figure out why this is so before continuing to the next paragraph).

The bottom query will indeed find all rows that have the same title and different EmployeeNumbers. If one is found, it will produce the E2.EmployeeNumber of that row. But that value will then be compared with E1.EmployeeNumber. These two values will always be different because of the condition

```
E1.EmployeeNumber <> E2.EmployeeNumber
```

No rows are returned because the values of the two unequal EmployeeNumbers are used in the IN clause instead of the values of the two equal LastNames.

SQL Correlated Subqueries Using the EXISTS and NOT EXISTS Comparison Operators

In Chapter 3, we discussed a set of SQL comparison operators, and these are summarized in Figure 3-16. To this set we will now add the **SQL EXISTS comparison operator** and the **SQL NOT EXISTS comparison operator**, as shown in Figure E-16. When we use the EXIST or NOT EXISTS operator in a query, we are creating another form of correlated subquery.

These operators simply test whether there are any values returned by the subquery, which indicates there are values meeting the conditions of the subquery. If one or more values are returned, then values from the subquery are used to run the top-level query. If there are no values returned, the top-level query produces an empty set as the result.

For example, we can rewrite the SQL-Query-AppE-05 correlated subquery using the SQL EXISTS keyword as follows:

```
/* *** SQL-Query-AppE-07 *** */
SELECT      E1.EmployeeNumber, E1.FirstName, E1.LastName
FROM        EMPLOYEE AS E1
WHERE       EXISTS
            (SELECT     E2.LastName
             FROM       EMPLOYEE AS E2
             WHERE      E1.LastName = E2.LastName
                AND     E1.EmployeeNumber <> E2.EmployeeNumber);
```

Because using EXISTS creates a form of a correlated subquery, the processing of the SELECT statements is nested. The first row of E1 is input to the subquery. If the subquery finds any row in E2 for which the last names are the same and the employee numbers are different, then the EXISTS is true (returns a non-empty set of values) and the FirstName, LastName, and EmployeeNumber for the first row are selected. Next, the second row of E1 is input to the subquery, the SELECT is processed, and the EXISTS is evaluated. If true, the FirstName, LastName, and EmployeeNumber of the second row are selected. This process is repeated for all of the rows in E1.

SQL Comparison Operators	
Operator	**Meaning**
EXISTS	Is a non-empty set of values
NOT EXISTS	Is an empty set

Figure E-16 — SQL EXISTS and NOT EXISTS Comparison Operators

The results of SQL-Query-AppE-07 are identical to the previous results from SQL-Query-AppE-05:

	EmployeeNumber	FirstName	LastName
1	2	Rosalie	Jackson
2	4	George	Smith
3	9	Heather	Jones
4	16	Mary	Smith
5	17	Tom	Jackson
6	18	George	Jones

The SQL EXISTS operator will be true (will return a non-empty set of values) if *any* row in the subquery *meets the condition*. The SQL NOT EXISTS operator will be true (will return an empty set) only if *all* rows in the subquery fail to meet the condition. Consequently, the double use of NOT EXISTS can be used to find rows that do not *not match a condition*. And, yes, the word *not* is supposed to be there *twice*—this is a *double negative*. Because of the logic of a double negative, if a row does not *not match any row*, then it matches *every row*!

This leads to some interesting uses of NOT EXISTS, but that discussion is beyond the scope of the book. If you are interested, see David M. Kroenke and David J. Auer, *Database Processing: Fundamentals, Design, and Implementation*, 14th edition (Upper Saddle River, NJ: Prentice Hall, 2016), pages 398–403. The discussion in that book includes additional uses of correlated subqueries.

SQL Queries on Recursive Relationships

In Chapters 4 (see page 281) and 5 (see pages 336–340), we discussed the logic of recursive relationships, where data in a table is related to other data in the same table. We will now discuss and illustrate how to create SQL queries on a recursive relationship. As discussed in Chapter 5, 1:1 and 1:N recursive queries require only the table that is referenced by the recursive relationship, while N:M recursive queries require an intersection table (just as any other N:M relationship does).

As illustrated in Chapter 5, recursive queries use the same technique we just used in correlated subqueries of using the same table under two different aliases. This gives us two virtual tables, and the queries using them are written just as we would write any other SQL query.

For example, our WP database EMPLOYEE table has a 1:N recursive relationship between Supervisor and EmployeeNumber. One person, the Supervisor, can supervise many other employees. We can visually examine these relationships in the EMPLOYEE table by using SQL-Query-AppE-08, but to specifically list who supervises whom, we can use query SQL-Query-AppE-09.

```
/* *** SQL-Query-AppE-08 *** */
SELECT      EmployeeNumber, FirstName, LastName, Supervisor
FROM        EMPLOYEE
ORDER BY    EmployeeNumber;
```

	EmployeeNumber	FirstName	LastName	Supervisor
1	1	Mary	Jacobs	NULL
2	2	Rosalie	Jackson	1
3	3	Richard	Bandalone	1
4	4	George	Smith	1
5	5	Alan	Adams	4
6	6	Ken	Evans	1
7	7	Mary	Abernathy	6
8	8	Tom	Caruthers	6
9	9	Heather	Jones	6
10	10	Ken	Numoto	1
11	11	Linda	Granger	10
12	12	James	Nestor	1
13	13	Rick	Brown	12
14	14	Mike	Nguyen	1
15	15	Jason	Sleeman	14
16	16	Mary	Smith	1
17	17	Tom	Jackson	14
18	18	George	Jones	15
19	19	Julia	Hayakawa	15
20	20	Sam	Stewart	15

```
/* *** SQL-Query-AppE-09 *** */
SELECT      S.FirstName AS SupervisorFirstName,
            S.LastName AS SupervisorLastName,
            E.FirstName AS EmployeeFirstName,
            E.LastName AS EmployeeLastName
FROM        EMPLOYEE S JOIN EMPLOYEE E
   ON       S.EmployeeNumber = E.Supervisor
ORDER BY    S.EmployeeNumber;
```

The result for SQL-Query-AppE-09 clearly shows which employees are supervisors and which employees they supervise:

	SupervisorFirstName	SupervisorLastName	EmployeeFirstName	EmployeeLastName
1	Mary	Jacobs	Rosalie	Jackson
2	Mary	Jacobs	Richard	Bandalone
3	Mary	Jacobs	George	Smith
4	Mary	Jacobs	Ken	Evans
5	Mary	Jacobs	Ken	Numoto
6	Mary	Jacobs	James	Nestor
7	Mary	Jacobs	Mary	Smith
8	Mary	Jacobs	Mike	Nguyen
9	George	Smith	Alan	Adams
10	Ken	Evans	Mary	Abernathy
11	Ken	Evans	Tom	Caruthers
12	Ken	Evans	Heather	Jones
13	Ken	Numoto	Linda	Granger
14	James	Nestor	Rick	Brown
15	Mike	Nguyen	Tom	Jackson
16	Mike	Nguyen	Jason	Sleeman
17	Jason	Sleeman	George	Jones
18	Jason	Sleeman	Julia	Hayakawa
19	Jason	Sleeman	Sam	Stewart

Using SQL Set Operators

Mathematicians use the term set theory to describe mathematical operations on *sets*, where a **set** is defined as a group of distinct items. A relational database table meets the definition of a set, so it is little wonder that SQL includes a group of set operators for use with SQL queries. **Venn diagrams** are the standard method of visualizing sets and their relationships, and Venn diagrams are illustrated in Figure E-17.

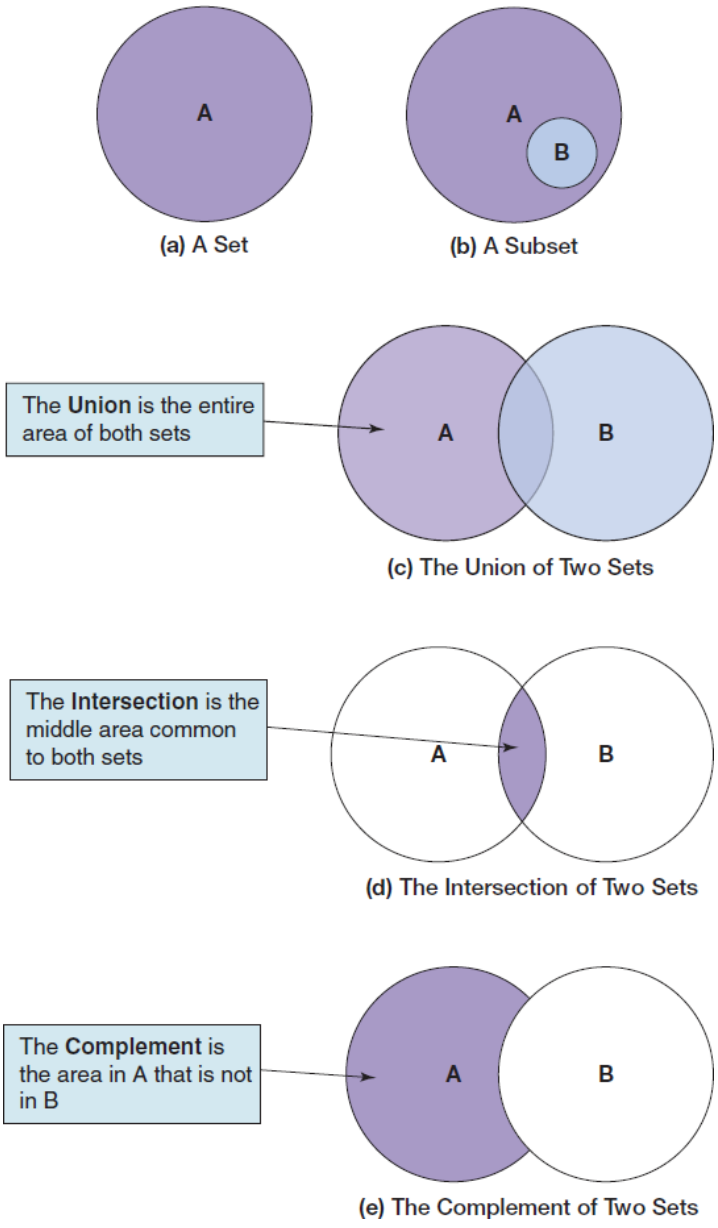


Figure E-17 — Venn Diagrams

As shown in Figure E-17:

- A **set** is represented by a labeled circle, as shown in Figure E-17(a).
- A **subset** is a portion of a set that is contained entirely within the set, as shown in Figure E-17(b).
- The **union** of two sets is shown in Figure E-17(c), and represents the two sets together to get a set that contains all values in both sets. This is equivalent to an OR logical operation (A OR B).
- The **intersection** of two sets is shown in Figure E-17(d), and represents the area common to both sets. This is equivalent to an AND logical operation (A AND B).
- The **complement** of set B in set A is shown in Figure E-17(e), and represents everything in set A that is not in set B. This is equivalent to a logical operation using A AND (NOT B). It is equivalent to the set different (A – B).

SQL provides **SQL set operators** for each of these set operations, and these are shown in Figure E-18. However, not all DBMS products support all of these operators, and to confuse things even more, Oracle Database use the set operator MINUS in place of EXPECT. The one operator that is always supported is the **SQL UNION operator**, and that is the set operator we will use here. Note that in order to use SQL set operators, the table columns involved in the operations must be the same number in each SELECT component, and corresponding columns must have the same or compatible (e.g., CHAR and VARCHAR) data types!

To illustrate SQL set operations, imagine that your boss asks you the question: "What products were available for sale (by either catalog or Web site) in 2015 and 2016?" Looking at Figures E-4 and E-5, we can see that to answer this question we must combine all the data in the CATALOG_SKU_2015 and CATALOG_SKU_2016 tables. We do this using the SQL UNION operator, as shown in SQL-Query-AppE-10:

```
/* *** SQL-Query-AppE-10 *** */
SELECT SKU, CatalogDescription, CatalogPage, DateOnWebSite
FROM CATALOG_SKU_2015
UNION
SELECT SKU, CatalogDescription, CatalogPage, DateOnWebSite
FROM CATALOG_SKU_2016;
```

SQL Set Operators	
Operator	**Meaning**
UNION	The result is all the row values in one or both tables
INTERSECT	The result is all the row values common to both tables
EXCEPT	The result is all the row values in the first table but not the second

Figure E-18 — SQL Set Operators

The result combines all the rows of CATALOG_SKU_2015 and CATALOG_SKU_2015. The result shown below is certainly sorted by SKU, but there is no guarantee that UNION results will be sorted—they often seem that way because of the algorithms used for the UNION operation, but the only way to control sort order is to an ORDER BY clause to the query. Note that the result shows two rows for both SKU 16103001 and SKU 16103005 because these SKUs appeared in both years. Note that even though these SKUs had a different CatalogDescription in the two years, this did not affect the result—there would have been two rows for each SKU even if the CatalogDescription was identical in both years.

	SKU	CatalogDescription	CatalogPage	DateOnWebSite
1	150102001	Our low price Alpha II model in black.	10	2015-01-01
2	150102005	Our low price Alpha II model in white.	12	2015-01-01
3	150201001	Our new Bravo I model in black.	18	2015-01-01
4	150201005	Our new Bravo I model in white.	20	2015-01-01
5	150303001	Our high performance Delta III model in black.	24	2015-01-01
6	150303005	Our high performance Delta III model in white.	26	2015-01-01
7	160103001	New, updated Alpha III model in black.	NULL	2015-12-01
8	160103001	Our low price Alpha III model in black.	10	2016-01-01
9	160103005	New, updated Alpha III model in white.	NULL	2015-12-01
10	160103005	Our low price Alpha III model in white.	11	2016-01-01
11	160202001	Our new Bravo II model in black.	18	2016-01-01
12	160202005	Our new Bravo II model in white.	17	2016-01-01
13	160304001	Our high performance Delta IV model in black.	22	2016-01-01
14	160304005	Our high performance Delta IV model in white.	23	2016-01-01

Creating and Working with SQL Views

An **SQL view** is a virtual table that is constructed from other tables or views. A view has no data of its own but uses data stored in tables or other views. Views are created using SQL SELECT statements and then used in other SELECT statements as just another table. The only limitation on the SQL statements that create the views is that they cannot contain ORDER BY clauses.[1] If the results of a query using a view need to be sorted, the sort order must be provided by the SELECT statement that processes the view.

Does Not Work with Microsoft Access ANSI-89 SQL

Unfortunately, Microsoft Access does not support views. However, Access allows you to create a query, name it, and then save it, which is not supported in a standard SQL implementation. You can then process Access queries in the same ways that you process views in the following discussion.

Solution: Create Microsoft Access view–equivalent queries, as discussed in the "The Access Workbench" section at the end of this appendix.

[1] This limitation appears in the SQL-92 standard. Some DBMSs modify this limitation in their implementation of SQL. For example, Oracle Database allows views to include ORDER BY, and SQL Server allows ORDER BY in very limited circumstances.

We'll use the WP database that we created in Chapter 3 as the example database for this discussion of views. You use the **SQL CREATE VIEW statement** to create view structures. The syntax of this statement is:

```
/* *** EXAMPLE CODE - DO NOT RUN *** */
/* *** SQL-CREATE-VIEW-AppE-01 *** */
CREATE VIEW {ViewName} AS
   {SQL SELECT statement};
```

The following statement defines a view named EmployeePhoneView based on the EMPLOYEE table:

```
/* *** SQL-CREATE-VIEW-AppE-02 *** */
CREATE VIEW EmployeePhoneView AS
   SELECT   FirstName, LastName, OfficePhone AS EmployeePhone
   FROM     EMPLOYEE;
```

Figure E-19 shows the view being created in the SQL Server Management Studio, Figure E-20 shows the view being created in the Oracle SQL Developer, and Figure E-21 shows the view being created in the MySQL Workbench.

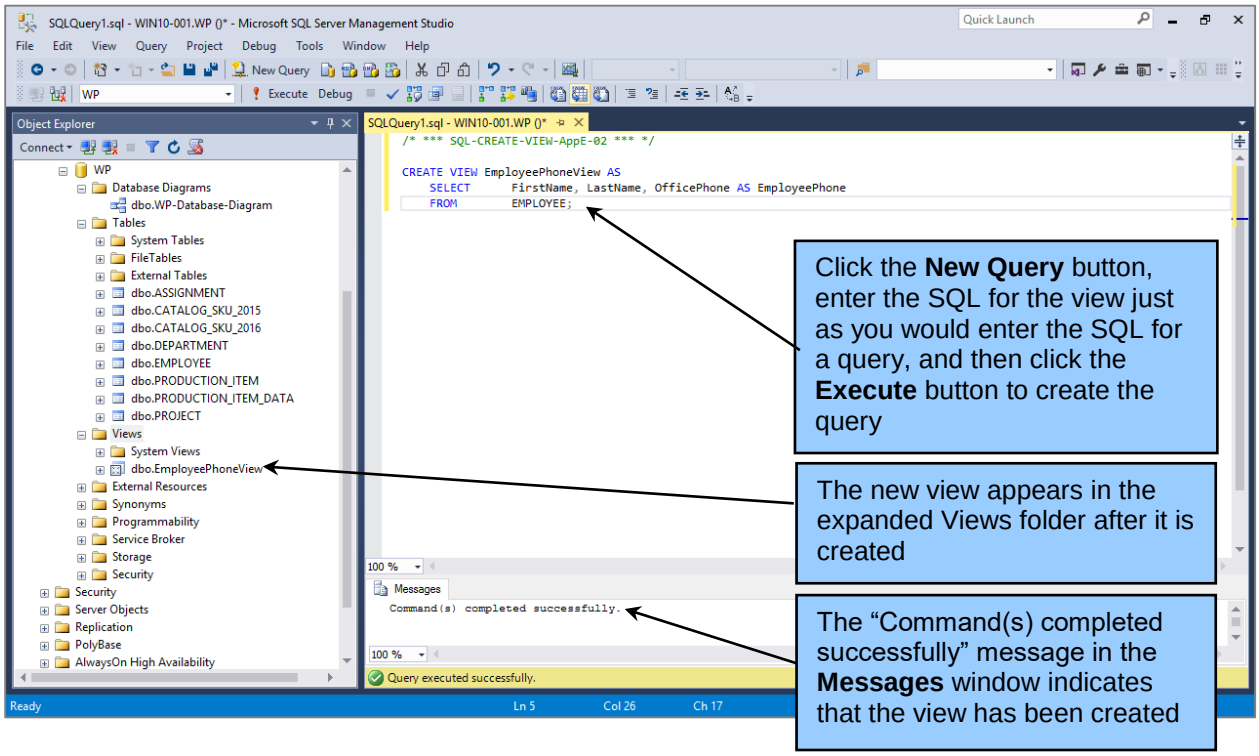


Figure E-19 — Creating a View in the Microsoft SQL Server Management Studio

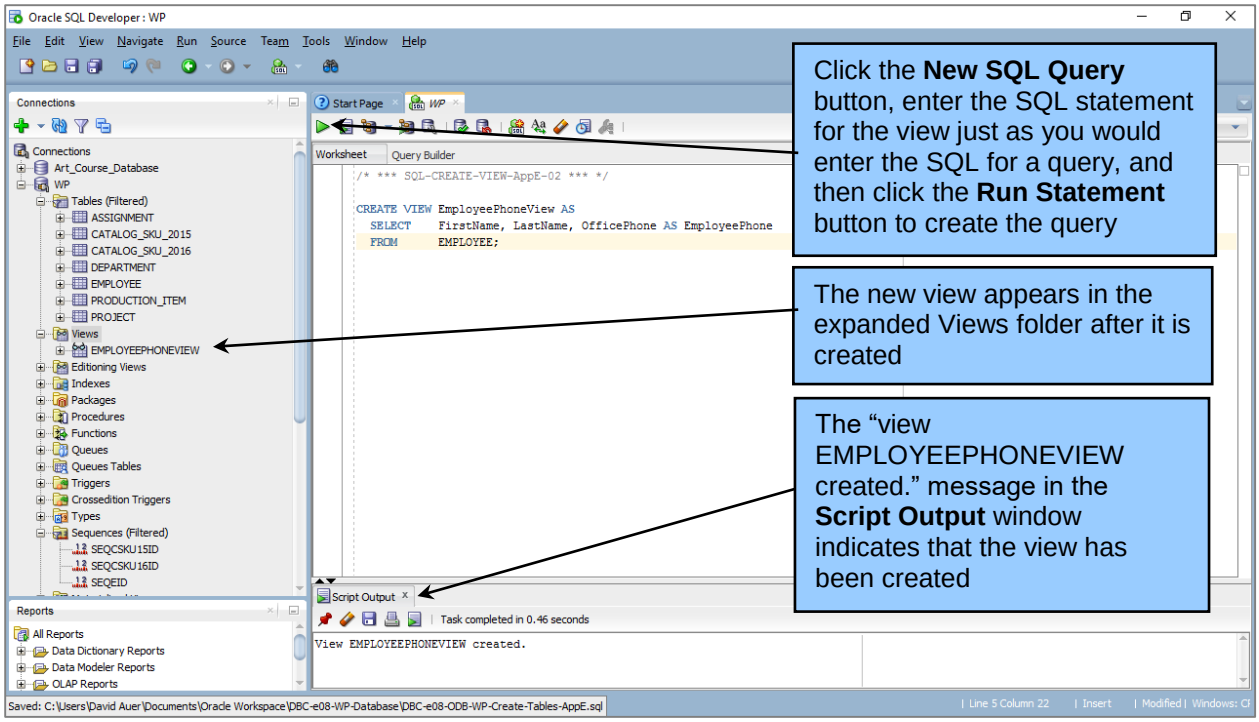


Figure E-20 — Creating a View in the Oracle SQL Developer

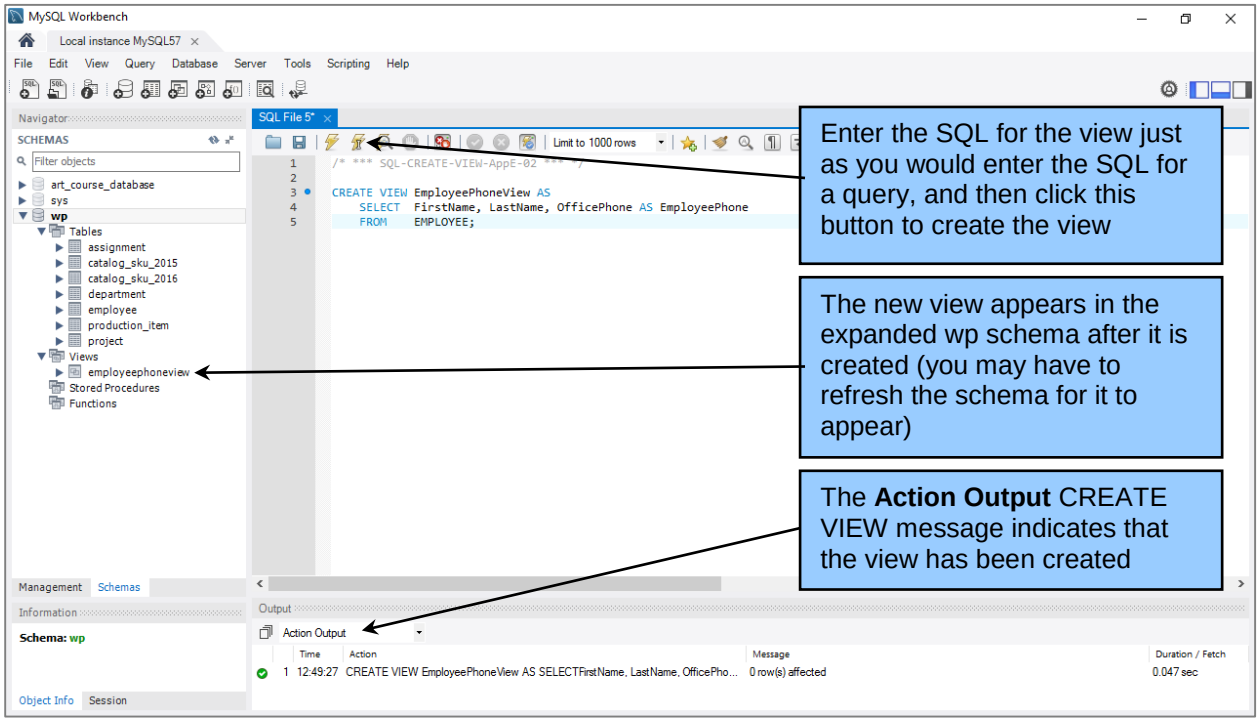


Figure E-21 — Creating a View in the MySQL Workbench

By The Way

SQL Server 2005, SQL Server 2008, SQL Server 2008 R2, SQL Server 2012, SQL Server 2014, SQL Server 2016, and most other DBMSs process the CREATE VIEW statements as written here without difficulty. However, SQL Server 2000 will not run such statements unless you *remove the semicolon* at the end of the CREATE VIEW statement. We have no idea why SQL Server 2000 works this way, but be aware of this peculiarity if you are using SQL Server 2000 (which you really shouldn't be because it is no longer being supported by Microsoft).

After we create the view, we can use it in the FROM clause of SELECT statements just as you'd use a table. The following obtains a list of employee names and phone numbers, sorted first by employee last name and then by employee first name:

```
/* *** SQL-Query-AppE-11 *** */
SELECT      *
FROM        EmployeePhoneView
ORDER BY    LastName, FirstName;
```

Figure E-22 shows this SQL statement run in the SQL Server Management Studio, Figure E-23 shows it run in the Oracle SQL Developer, and Figure E-24 shows it run in the MySQL Workbench.

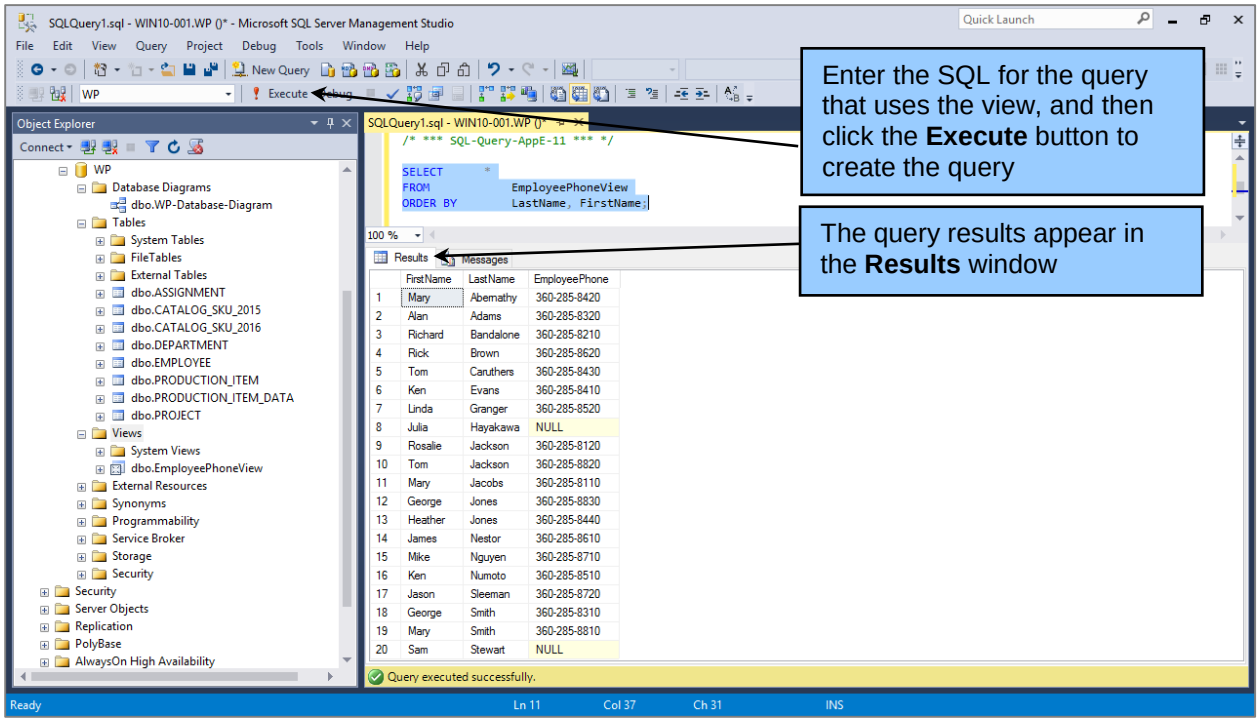


Figure E-22 — Using EmployeePhoneView in the Microsoft SQL Server Management Studio

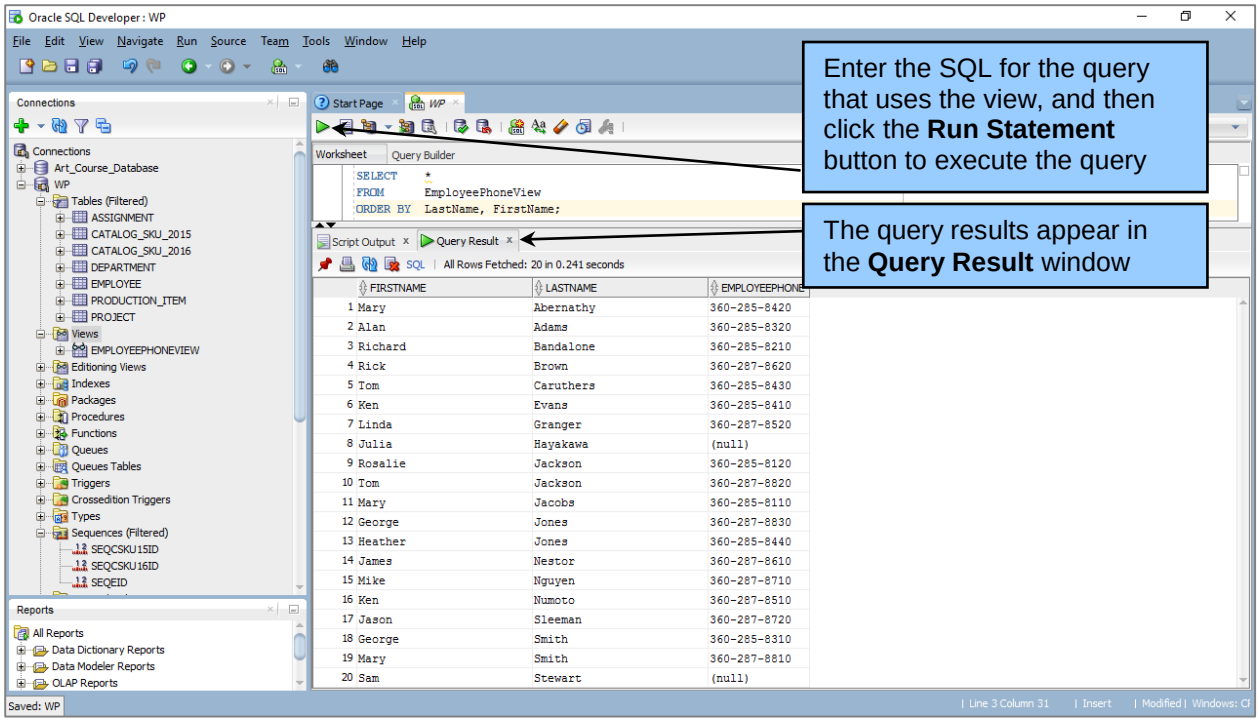


Figure E-23 — Using EmployeePhoneView in the Oracle SQL Developer

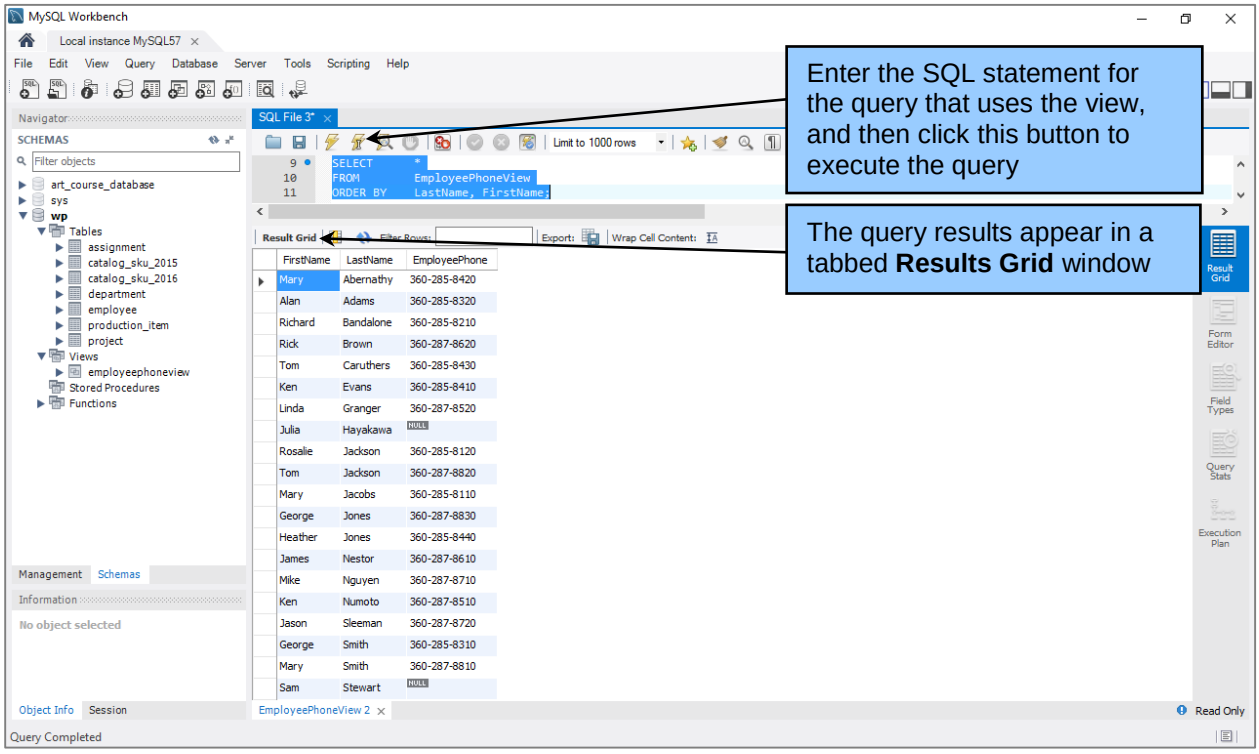


Figure E-24 — Using EmployeePhoneView in the MySQL Workbench

Note that the number of columns returned depends on the number of columns in the view, not on the number of columns in the underlying table. In this example, SELECT * produces just three columns because the view has just three columns. Also notice that the column Name in the EMPLOYEE table has been renamed EmployeePhone in the view, so the DBMS uses the label EmployeePhone when producing results.

By The Way

If you ever need to modify an SQL view you have created, use the **SQL ALTER VIEW {ViewName} statement**. This works exactly the same as the CREATE VIEW {ViewName} statement except that it replaces the existing view definition with the new one. This statement is very useful when you are trying to fine-tune your view definitions. And If you ever want to delete a view, simply use the **SQL DROP VIEW {ViewName} statement**.

Using SQL Views

In general, SQL views are used to prepare data for use in an information system application, which may or may not be a Web based application. While applications commonly use a Web interface (via a Web browser such as Microsoft Internet Explorer [IE], Google Chrome, or Mozilla Firefox), there are still many applications that run in their own application window.

In Appendix F, "Getting Started with Systems Analysis and Development," we define **data** as recorded facts and numbers. Based on this definition, we can now define[2] **information** as:

- Knowledge derived from data.
- Data presented in a meaningful context.
- Data processed by summing, ordering, averaging, grouping, comparing, or other similar operations.

In general, application programmers prefer that the work of transforming *database data* into the *information* that will be used in and presented by the application be done by the DBMS itself. SQL views are the main DBMS tool for this work. The basic principle is that all *summing, averaging, grouping, comparing, and similar operations* should be done in SQL views, and that it is the final result as it appears in the SQL view that is passed to the application program for use. This is illustrated in Figure E-25.

For a specific example, let's consider a Web page that we'll build in Chapter 7's section of "The Access Workbench." We are building a customer relationship management (CRM) Web application for Wallingford Motors. As shown in Figure E-26, one part of the Web CRM application displays a report

[2] These definitions are from David M. Kroenke and Randall J Boyles's books: *Using MIS* (9th ed.) (Upper Saddle River, NJ: Prentice Hall, 2017) and *Experiencing MIS* (7th ed.) (Upper Saddle River, NJ: Prentice Hall, 2017). See these books for a full discussion of these definitions, as well as a discussion of a fourth definition, "a difference that makes a difference."

named *The Wallingford Motors CRM Customer Contacts List*, which shows all contacts between WM salespeople (identified by NickName) and customers (identified by LastName and FirstName). This report is based on a view named *viewCustomerContacts*, which combines data from the both the CUSTOMER table and the CONTACT table in the WM database. This example clearly illustrates the principle of combing and processing data into a view that becomes the basis of the data sent to the Web application for display in a Web page.

Figure E-27 lists some of the specific uses for views.[3] They can hide columns or rows. They also can be used to display the results of computed columns, to hide complicated SQL syntax, and to layer the use of built-in functions to create results that are not possible with a single SQL statement. We will give examples of each of these uses.

Using Views to Hide Columns or Rows

Views can be used to hide columns to simplify results or to prevent the display of sensitive data. For example, suppose the users at WP want a simplified list of departments that has just the department names and phone numbers. One use for such a view would be to populate a Web page. The following statement defines a view, BasicDepartmentDataView, that will produce that list:

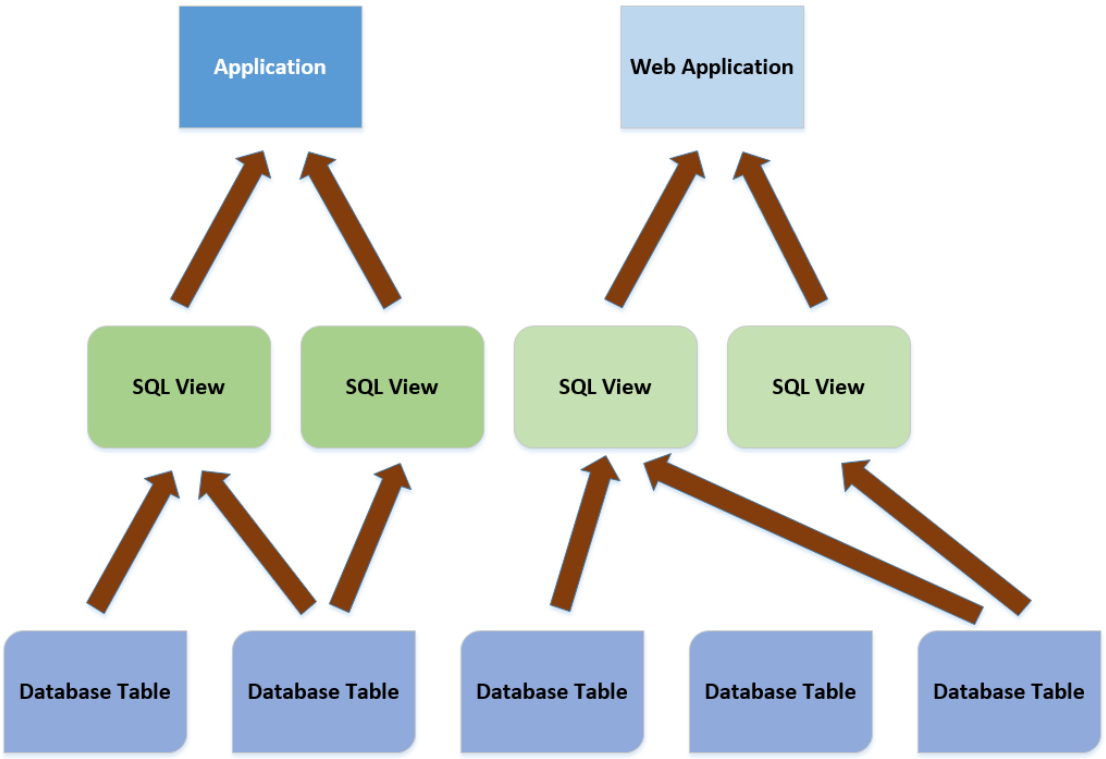


Figure E-25 — SQL Views as the Basis for Application Reports

[3] Additional uses of SQL views are discussed in David M. Kroenke and David J. Auer, *Database Processing: Fundamentals, Design, and Implementation*, 14th edition (Upper Saddle River, NJ: Prentice Hall, 2016), Chapter 7.

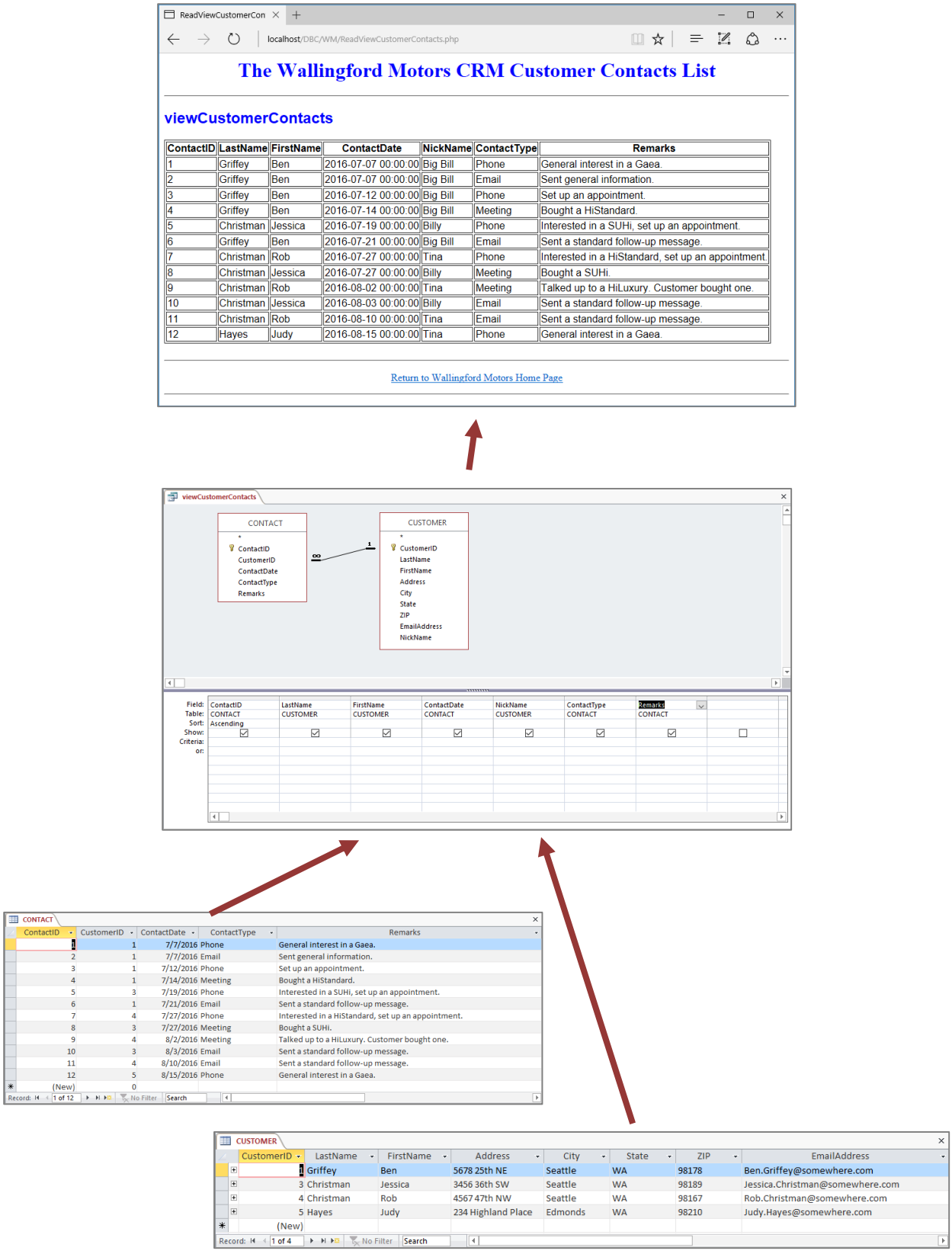


Figure E-26— The Wallingford Motors CRM Web Application Customer Contacts List

Uses of SQL Views
Hide columns or rows.
Display results of computations.
Hide complicated SQL syntax.
Layer built-in functions.

Figure E-27 — Some Uses for SQL Views

```
/* *** SQL-CREATE-VIEW-AppE-03 *** */
CREATE VIEW BasicDepartmentDataView AS
   SELECT      DepartmentName, DepartmentPhone
   FROM        DEPARTMENT;
```

The following SELECT statement obtains a list of department names and phone numbers sorted by the DepartmentName:

```
/* *** SQL-Query-AppE-12 *** */
SELECT      *
FROM        BasicDepartmentDataView
ORDER BY    DepartmentName;
```

The results of a SELECT sorted by DepartmentName on this view are:

	DepartmentName	DepartmentPhone
1	Accounting	360-285-8405
2	Administration	360-285-8100
3	Finance	360-285-8400
4	Human Resources	360-285-8300
5	InfoSystems	360-287-8600
6	Legal	360-285-8200
7	Production	360-287-8800
8	Research and Development	360-287-8700
9	Sales and Marketing	360-287-8500

Views can also hide rows by providing a WHERE clause in the view definition. The next SQL statement defines a view of WP projects in the Sales and Marketing department:

```
/* *** SQL-CREATE-VIEW-AppE-04 *** */
CREATE VIEW MarketingDepartmentProjectView AS
   SELECT      ProjectID, ProjectName, MaxHours,
               StartDate, EndDate
   FROM        PROJECT
   WHERE       Department = 'Sales and Marketing';
```

The following SELECT statement obtains a list of projects managed by the Sales and Marketing department, sorted by the ProjectID number:

```
/* *** SQL-Query-AppE-13 *** */
SELECT      *
FROM        MarketingDepartmentProjectView
ORDER BY    ProjectID;
```

The results of a SELECT sorted by ProjectID on this view are:

	ProjectID	ProjectName	MaxHours	StartDate	EndDate
1	1100	2017 Q3 Marketing Plan	135.00	2017-05-10	2017-06-15
2	1500	2017 Q4 Marketing Plan	135.00	2017-08-10	2017-09-15

As desired, only the Sales and Marketing department projects are shown in this view. This limitation is not obvious from the results because Department is not included in the view. This characteristic can be good or bad, depending on the use of the view. It is good if this view is used in a setting in which only marketing department projects matter; it is bad if the view indicates that these projects are the only WP projects currently under way.

Using Views to Display Results of Computed Columns

Another use of views is to show the results of computed columns without requiring the user to enter the computation expression. For example, the following view allows the user to compare the maximum hours allocated for each WP project to the total hours worked to date on the project:

```
/* *** SQL-CREATE-VIEW-AppE-05 *** */
CREATE VIEW ProjectHoursToDateView AS
   SELECT      PROJECT.ProjectID, ProjectName,
               MaxHours AS ProjectMaxHours,
               SUM(HoursWorked) AS ProjectHoursWorkedToDate
   FROM        PROJECT JOIN ASSIGNMENT
      ON       PROJECT.ProjectID = ASSIGNMENT.ProjectID
   GROUP BY    PROJECT.ProjectID;
```

Note, however, that Microsoft SQL Server and Oracle Database require that any column specified in the SELECT phrase be used in either an SQL built-in function or the GROUP BY phrase. The previous SQL

statement is correct SQL-92 syntax and will run in MySQL as written. However, Microsoft SQL Server and Oracle Database require you to write:

```
/* *** SQL-CREATE-VIEW-AppE-05-MSSQL *** */
CREATE VIEW ProjectHoursToDateView AS
   SELECT      PROJECT.ProjectID, ProjectName,
               MaxHours AS ProjectMaxHours,
               SUM(HoursWorked) AS ProjectHoursWorkedToDate
   FROM        PROJECT JOIN ASSIGNMENT
         ON    PROJECT.ProjectID = ASSIGNMENT.ProjectID
   GROUP BY    PROJECT.ProjectID, ProjectName, MaxHours;
```

Note the use of the extra column names in the GROUP BY clause. These are necessary to create the view but have no practical effect on the results. When the view user enters:

```
/* *** SQL-Query-AppE-14 *** */
SELECT      *
FROM        ProjectHoursToDateView
ORDER BY    ProjectID;
```

these results are displayed:

	ProjectID	ProjectName	ProjectMaxHours	ProjectHoursWorkedToDate
1	1000	2017 Q3 Production Plan	100.00	280.00
2	1100	2017 Q3 Marketing Plan	135.00	215.00
3	1200	2017 Q3 Portfolio Analysis	120.00	150.00
4	1300	2017 Q3 Tax Preparation	145.00	165.00
5	1400	2017 Q4 Production Plan	100.00	280.00
6	1500	2017 Q4 Marketing Plan	135.00	215.00
7	1600	2017 Q4 Portfolio Analysis	140.00	150.00

Placing computations in views has two major advantages. First, it saves users from having to know or remember how to write an expression to get the results they want. Second, it ensures consistent results. If developers who use a computation write their own SQL expressions, they may write the expression differently and obtain inconsistent results.

Using Views to Hide Complicated SQL Syntax

Another use of views is to hide complicated SQL syntax. By using views, developers do not need to enter complex SQL statements when they want particular results. Also, such views allow developers who do not know how to write complicated SQL statements to enjoy the benefits of such statements. This use of views also ensures consistency.

Suppose that WP users need to know which employees are assigned to which projects and how many hours each employee has worked on that project. To display these interests, two joins are necessary: one to join EMPLOYEE to ASSIGNMENT and another to join that result to PROJECT. You saw the SQL statement to do this in Chapter 3:

```
/* *** SQL-Query-CH03-51 *** */
SELECT      ProjectName, FirstName, LastName, HoursWorked
FROM        EMPLOYEE AS E JOIN ASSIGNMENT AS A
      ON    E.EmployeeNumber = A.EmployeeNumber
            JOIN PROJECT AS P
                  ON    A.ProjectID = P.ProjectID
ORDER BY    P.ProjectID, A.EmployeeNumber;
```

Now we need to make it into a view named EmployeeProjectHoursWorkedView. Remember that we cannot include the ORDER BY clause in the view. If we want to sort the output, we'll need to do this when we use the view:

```
/* *** SQL-CREATE-VIEW-AppE-06 *** */
CREATE VIEW EmployeeProjectHoursWorkedView AS
   SELECT      ProjectName, FirstName, LastName, HoursWorked
   FROM        EMPLOYEE AS E JOIN ASSIGNMENT AS A
         ON    E.EmployeeNumber = A.EmployeeNumber
               JOIN PROJECT AS P
                  ON    A.ProjectID = P.ProjectID;
```

This is a complicated SQL statement to write, but after the view is created the results of this statement can be obtained with a simple SELECT statement. Now the user simply uses the SQL query (note the sorting by employee name—we *cannot* sort on ProjectID and EmployeeNumber because these columns were *not included* in the view):

```
/* *** SQL-Query-AppE-15 *** */
SELECT      *
FROM        EmployeeProjectHoursWorkedView
ORDER BY    LastName, FirstName;
```

The result of SQL-Query-AppE-15 is shown in Figure E-28. Clearly, using the view is much simpler than constructing the join syntax. Even developers who know SQL well will appreciate having a simpler view with which to work.

Layering Computations and Built-In Functions

Recall from Chapter 3 that you cannot use a computation or a built-in function as part of a WHERE clause. You can, however, construct a view that computes a variable and then write an SQL statement on that view that uses the computed variable in a WHERE clause. To understand this, consider the

	ProjectName	FirstName	LastName	HoursWorked
1	2017 Q3 Portfolio Analysis	Mary	Abernathy	45.00
2	2017 Q4 Portfolio Analysis	Mary	Abernathy	45.00
3	2017 Q4 Portfolio Analysis	Richard	Bandalone	20.00
4	2017 Q3 Portfolio Analysis	Richard	Bandalone	20.00
5	2017 Q3 Tax Preparation	Richard	Bandalone	25.00
6	2017 Q3 Tax Preparation	Tom	Caruthers	50.00
7	2017 Q3 Portfolio Analysis	Tom	Caruthers	45.00
8	2017 Q4 Portfolio Analysis	Tom	Caruthers	45.00
9	2017 Q4 Portfolio Analysis	Ken	Evans	40.00
10	2017 Q4 Production Plan	Ken	Evans	50.00
11	2017 Q4 Marketing Plan	Ken	Evans	75.00
12	2017 Q3 Portfolio Analysis	Ken	Evans	40.00
13	2017 Q3 Tax Preparation	Ken	Evans	40.00
14	2017 Q3 Production Plan	Ken	Evans	50.00
15	2017 Q3 Marketing Plan	Ken	Evans	75.00
16	2017 Q3 Marketing Plan	Linda	Granger	55.00
17	2017 Q4 Marketing Plan	Linda	Granger	55.00
18	2017 Q4 Production Plan	Tom	Jackson	75.00
19	2017 Q3 Production Plan	Tom	Jackson	75.00
20	2017 Q3 Marketing Plan	Mary	Jacobs	30.00
21	2017 Q3 Production Plan	Mary	Jacobs	30.00
22	2017 Q4 Marketing Plan	Mary	Jacobs	30.00
23	2017 Q4 Production Plan	Mary	Jacobs	30.00
24	2017 Q3 Tax Preparation	Heather	Jones	50.00
25	2017 Q3 Marketing Plan	Ken	Numoto	55.00
26	2017 Q3 Production Plan	Ken	Numoto	50.00
27	2017 Q4 Marketing Plan	Ken	Numoto	55.00
28	2017 Q4 Production Plan	Ken	Numoto	50.00
29	2017 Q4 Production Plan	Mary	Smith	75.00
30	2017 Q3 Production Plan	Mary	Smith	75.00

Figure E-28 — The Result for SQL-Query-AppE-15

ProjectHoursToDateView definition created previously as SQL-CREATE-VIEW-AppE-05 with the modified GROUP BY clause needed for SQL Server2016:

```
/* *** SQL-CREATE-VIEW-AppE-05-MSSQL *** */
CREATE VIEW ProjectHoursToDateView AS
   SELECT      PROJECT.ProjectID,
               ProjectName,
               MaxHours AS ProjectMaxHours,
               SUM(HoursWorked) AS ProjectHoursWorkedToDate
   FROM        PROJECT JOIN ASSIGNMENT
         ON    PROJECT.ProjectID = ASSIGNMENT.ProjectID
   GROUP BY    PROJECT.ProjectID, ProjectName, MaxHours;
```

The view definition contains the maximum allocated hours for each project and the total hours actually worked on the project to date as ProjectHoursWorkedToDate. Now we can use ProjectHoursWorkedToDate in both an additional calculation and the WHERE clause, as follows:

```
/* *** SQL-Query-AppE-16 *** */
SELECT      ProjectID, ProjectName, ProjectMaxHours,
            ProjectHoursWorkedToDate
FROM        ProjectHoursToDateView
WHERE       ProjectHoursWorkedToDate > ProjectMaxHours
ORDER BY    ProjectID;
```

Here, we are using the result of a computation in a WHERE clause, something that is not allowed in a single SQL statement. This allows users to determine which projects have exceeded the number of hours allocated to them by producing the result—and it looks like WP management needs to seriously rein in those project hours!

	ProjectID	ProjectName	ProjectMaxHours	ProjectHoursWorkedToDate
1	1000	2017 Q3 Production Plan	100.00	280.00
2	1100	2017 Q3 Marketing Plan	135.00	215.00
3	1200	2017 Q3 Portfolio Analysis	120.00	150.00
4	1300	2017 Q3 Tax Preparation	145.00	165.00
5	1400	2017 Q4 Production Plan	100.00	280.00
6	1500	2017 Q4 Marketing Plan	135.00	215.00
7	1600	2017 Q4 Portfolio Analysis	140.00	150.00

Such layering can be continued over many levels. We can turn this SELECT statement into another view named ProjectsOverAllocatedMaxHoursView (again without the ORDER BY clause):

```
/* *** SQL-CREATE-VIEW-AppE-07 *** */
CREATE VIEW ProjectsOverAllocatedMaxHoursView AS
   SELECT      ProjectID, ProjectName, ProjectMaxHours,
               ProjectHoursWorkedToDate
   FROM        ProjectHoursToDateView
   WHERE       ProjectHoursWorkedToDate > ProjectMaxHours;
```

Now we can use ProjectsOverAllocatedMaxHoursView in a further calculation—this time to find the number of hours by which each project has overrun its allocated hours:

```
/* *** SQL-Query-AppE-17 *** */
SELECT   ProjectID, ProjectName, ProjectMaxHours,
         ProjectHoursWorkedToDate,
         (ProjectHoursWorkedToDate - ProjectMaxHours)
               AS HoursOverMaxAllocated
FROM     ProjectsOverAllocatedMaxHoursView
ORDER BY ProjectID;
```

Here are the results, and now we can see just how far over ProjectMaxHours each project has gone:

	ProjectID	ProjectName	ProjectMaxHours	ProjectHoursWorkedToDate	HoursOverMaxAllocated
1	1000	2017 Q3 Production Plan	100.00	280.00	180.00
2	1100	2017 Q3 Marketing Plan	135.00	215.00	80.00
3	1200	2017 Q3 Portfolio Analysis	120.00	150.00	30.00
4	1300	2017 Q3 Tax Preparation	145.00	165.00	20.00
5	1400	2017 Q4 Production Plan	100.00	280.00	180.00
6	1500	2017 Q4 Marketing Plan	135.00	215.00	80.00
7	1600	2017 Q4 Portfolio Analysis	140.00	150.00	10.00

SQL views are very useful tools for database developers (who will define the views) and application programmers (who will use the views in applications).

SQL/Persistent Stored Modules (SQL/PSM)

Each DBMS product has its own variant or extension of SQL, including features that allow SQL to function similarly to a procedural programming language. The ANSI/ISO standard refers to these as **SQL/Persistent Stored Modules (SQL/PSM)**. Microsoft SQL Server calls its version of SQL *Transact-SQL (T-SQL)*, and Oracle Database calls its version of SQL *Procedural Language/SQL (PL/SQL)*. The MySQL variant also includes SQL/PSM components, but it has no special name and is just called *SQL* in the MySQL documentation.

SQL/PSM provides the program variables and cursor functionality discussed in Chapter 6. It also includes control-of-flow language such as BEGIN…END blocks, IF…THEN…ELSE logic structures, and LOOPs, as well as the ability to provide usable output to users.

The most important feature of SQL/PSM, however, is that it allows the code that implements these features in a database to be contained in that database. Thus the name: *Persistent*—the code remains available for use over time—*Stored*—the code is stored for reuse in the database—*Modules*—the code is written as a self-contained block of code. The SQL code can be written as one of three module types: user-defined functions, triggers, and stored procedures.

SQL/PSM User-Defined Functions

A **user-defined function** (also known as a *stored function*) is a stored set of SQL statements that:

- is *called by name* from another SQL statement,
- may have *input parameters* passed to it by the calling SQL statement, and
- *returns an output value* to the SQL statement that called the function.

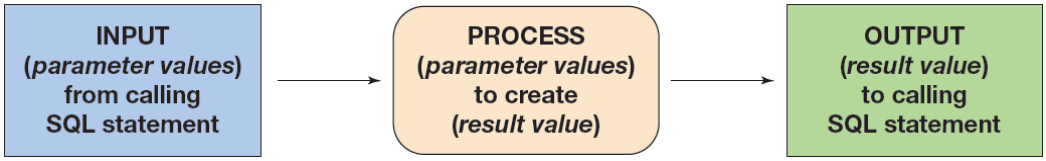


Figure E-29 — User-Defined Functions Logical Process Flow

The logical process flow of a user-defined function is illustrated in Figure E-29. SQL/PSM user-defined functions are very similar to the SQL built-in functions (COUNT, SUM, AVG, MAX, and MIN) that we discussed and used in Chapter 3, except that, as the name implies, we create them ourselves to perform specific tasks that we need to do.

A common problem is needing a name in the format *LastName, FirstName* (including the comma!) in a report when the database stores the basic data in two fields named *FirstName* and *LastName*. Using the data in the WP database, we could, of course, simply include the code to do this in an SQL statement using a concatenation operator:

```
/* *** SQL-Query-AppE-18 *** */
SELECT      RTRIM(LastName)+', '+RTRIM(FirstName) AS EmployeeName,
            Department, OfficePhone, EmailAddress
FROM        EMPLOYEE
ORDER BY    EmployeeName;
```

This produces the desired results, as shown in Figure E-30, but at the expense of working out some cumbersome coding.

SQL-Query-AppE-18 is written for SQL Server 2016 using SQL Server T-SQL. As usual, SQL syntax varies from DBMS to DBMS. Oracle Database uses a double vertical bar [||] as the concatenation operator, and SQL-Query-AppE-18 is written for Oracle Database as:

```
/* *** SQL-Query-AppE-18-Oracle *** */
SELECT      RTRIM(LastName)||', '||RTRIM(FirstName) AS EmployeeName,
            Department, Phone, Email
FROM        EMPLOYEE
ORDER BY    EmployeeName;
```

MySQL 5.7 uses the concatenation string function CONCAT() as the concatenation operator, and SQL-Query-AppE-18 is written for MySQL 5.7 as:

```
/* *** SQL-Query-AppE-18-MySQL *** */
SELECT      CONCAT(RTRIM(LastName),', ',RTRIM(FirstName))
               AS EmployeeName,
            Department, Phone, Email
FROM        EMPLOYEE
ORDER BY    EmployeeName;
```

	EmployeeName	Department	OfficePhone	EmailAddress
1	Abernathy, Mary	Finance	360-285-8420	Mary.Abernathy@WP.com
2	Adams, Alan	Human Resources	360-285-8320	Alan.Adams@WP.com
3	Bandalone, Richard	Legal	360-285-8210	Richard.Bandalone@WP.com
4	Brown, Rick	InfoSystems	360-285-8620	Rick.Brown@WP.com
5	Caruthers, Tom	Accounting	360-285-8430	Tom.Caruthers@WP.com
6	Evans, Ken	Finance	360-285-8410	Ken.Evans@WP.com
7	Granger, Linda	Sales and Marketing	360-285-8520	Linda.Granger@WP.com
8	Hayakawa, Julia	Production	NULL	Julia.Hayakawa@WP.com
9	Jackson, Rosalie	Administration	360-285-8120	Rosalie.Jackson@WP.com
10	Jackson, Tom	Production	360-285-8820	Tom.Jackson@WP.com
11	Jacobs, Mary	Administration	360-285-8110	Mary.Jacobs@WP.com
12	Jones, George	Production	360-285-8830	George.Jones@WP.com
13	Jones, Heather	Accounting	360-285-8440	Heather.Jones@WP.com
14	Nestor, James	InfoSystems	360-285-8610	James.Nestor@WP.com
15	Nguyen, Mike	Research and De...	360-285-8710	Mike.Nguyen@WP.com
16	Numoto, Ken	Sales and Marketing	360-285-8510	Ken.Numoto@WP.com
17	Sleeman, Jason	Research and De...	360-285-8720	Jason.Sleeman@WP.com
18	Smith, George	Human Resources	360-285-8310	George.Smith@WP.com
19	Smith, Mary	Production	360-285-8810	Mary.Smith@WP.com
20	Stewart, Sam	Production	NULL	Sam.Stewart@WP.com

Figure E-30 — The Result of SQL-Query-AppE-18

The alternative is to create a user-defined function to store this code. Not only does this make it easier to use, but it also makes it available for use in other SQL statements. Figure E-31 shows a user-defined function written in T-SQL for use with Microsoft SQL Server 2016, and the SQL code for the function uses, as we would expect, specific syntax requirements for Microsoft SQL Server's T-SQL 2016:

```
CREATE FUNCTION dbo.NameConcatenation
-- These are the input parameters
    (
    @FirstName CHAR(25),
    @LastName CHAR(25)
    )
RETURNS VARCHAR(60)
AS
BEGIN
    -- This is the variable that will hold the value to be returned
    DECLARE @FullName VARCHAR(60);
    -- SQL statements to concatenate the names in the proper order
    SELECT @FullName = RTRIM(@LastName) + ', ' + RTRIM(@FirstName);
    -- Return the concatenated name
    RETURN @FullName;
END
```

Figure E-31 — SQL Server 2016 User-Defined Function Code to Concatenate FirstName and LastName

- The function is created and stored in the database by using the T-SQL CREATE FUNCTION statement.
- The function name starts with *dbo*, which is a Microsoft SQL Server *schema* name.

This use of a schema name appended to a database object name is common in Microsoft SQL Server.

- The variable names of both the input parameters and the returned output value start with @.
- The concatenation syntax is T-SQL syntax.

Now that we have created and stored the user-defined function, we can use it in SQL-Query-AppE-19:

```
/* *** SQL-Query-AppE-19 *** */
SELECT      dbo.NameConcatenation(FirstName, LastName) AS EmployeeName,
            Department, OfficePhone, EmailAddress
FROM        EMPLOYEE
ORDER BY    EmployeeName;
```

The advantage of having a user-defined function is that we can now use it whenever we need to without having to recreate the code. Now we have a function that produces the results we want, which of course are identical to the results for SQL-Query-AppE-18 above, and which are shown in Figure E-32.

	EmployeeName	Department	OfficePhone	EmailAddress
1	Abernathy, Mary	Finance	360-285-8420	Mary.Abernathy@WP.com
2	Adams, Alan	Human Resources	360-285-8320	Alan.Adams@WP.com
3	Bandalone, Richard	Legal	360-285-8210	Richard.Bandalone@WP.com
4	Brown, Rick	InfoSystems	360-285-8620	Rick.Brown@WP.com
5	Caruthers, Tom	Accounting	360-285-8430	Tom.Caruthers@WP.com
6	Evans, Ken	Finance	360-285-8410	Ken.Evans@WP.com
7	Granger, Linda	Sales and Marketing	360-285-8520	Linda.Granger@WP.com
8	Hayakawa, Julia	Production	NULL	Julia.Hayakawa@WP.com
9	Jackson, Rosalie	Administration	360-285-8120	Rosalie.Jackson@WP.com
10	Jackson, Tom	Production	360-285-8820	Tom.Jackson@WP.com
11	Jacobs, Mary	Administration	360-285-8110	Mary.Jacobs@WP.com
12	Jones, George	Production	360-285-8830	George.Jones@WP.com
13	Jones, Heather	Accounting	360-285-8440	Heather.Jones@WP.com
14	Nestor, James	InfoSystems	360-285-8610	James.Nestor@WP.com
15	Nguyen, Mike	Research and De...	360-285-8710	Mike.Nguyen@WP.com
16	Numoto, Ken	Sales and Marketing	360-285-8510	Ken.Numoto@WP.com
17	Sleeman, Jason	Research and De...	360-285-8720	Jason.Sleeman@WP.com
18	Smith, George	Human Resources	360-285-8310	George.Smith@WP.com
19	Smith, Mary	Production	360-285-8810	Mary.Smith@WP.com
20	Stewart, Sam	Production	NULL	Sam.Stewart@WP.com

Figure E-32 — The Result of SQL-Query-AppE-19

By The Way

The user-defined function NameConcatenation as shown above is written for SQL Server 2016 using SQL Server T-SQL. As usual, SQL syntax varies from DBMS to DBMS.

The Oracle Database XE version is written as:

```
CREATE OR REPLACE FUNCTION NameConcatenation
    -- These are the input parameters
       (
        varFirstName    IN Char,
        varLastName     IN Char
       )
    -- This is the variable that will hold the returned value
       RETURN          Varchar
       IS varFullName   Varchar(60);
BEGIN
    -- SQL statemente to concatentate tne names in the proper order
    varFullName := (RTRIM(varLastName)||', '||RTRIM(varFirstName));
    -- Return the concatenated name
    RETURN varFullName;
END;
/
```

The MySQL 5.7 version is written as:

```
DELIMITER //

CREATE FUNCTION NameConcatenation
-- These are the input parameters
    (
    varFirstName CHAR(25),
    varLastName CHAR(25)
    )
RETURNS VARCHAR(60) DETERMINISTIC
BEGIN
    -- This is the variable that will hold the value to be returned
    DECLARE varFullName VARCHAR(60);
    -- SQL statements to concatenate the names in the proper order
    SET varFullName = CONCAT(varLastName, ', ', varFirstName);
    -- Return the concatenated name
    RETURN varFullName;
END
//

DELIMITER ;
```

SQL/PSM Triggers

A **trigger** is a stored program that is executed by the DBMS whenever a specified event occurs. Triggers for Oracle Database are written in Java or in Oracle's PL/SQL. SQL Server triggers are written in Microsoft .NET Common Language Runtime (CLR) languages, such as Visual Basic .NET, or Microsoft's T-SQL. MySQL triggers are written in MySQL's variant of SQL. In this chapter, we will discuss triggers in a generic manner without considering the particulars of those languages.

A trigger is attached to a table or a view. A table or a view may have many triggers, but a trigger is associated with just one table or view. A trigger is invoked by an SQL DML INSERT, UPDATE, or DELETE request on the table or view to which it is attached. Figure E-33 summarizes the triggers available for SQL Server 2016, Oracle Database XE, and MySQL 5.7.

Oracle Database XE supports three kinds of triggers: BEFORE, INSTEAD OF, and AFTER. As you would expect, BEFORE triggers are executed before the DBMS processes the insert, update, or delete request. INSTEAD OF triggers are executed in place of any DBMS processing of the insert, update, or delete request. AFTER triggers are executed after the insert, update, or delete request has been processed. Taken all together, nine trigger types are possible: BEFORE (INSERT, UPDATE, DELETE); INSTEAD OF (INSERT, UPDATE, DELETE); and AFTER (INSERT, UPDATE, DELETE).

Since SQL Server 2005, SQL Server supports DDL triggers (triggers on such SQL DDL statements as CREATE, ALTER, and DROP) as well as DML triggers. We will only deal with the DML triggers here, which for SQL Server 2016 are INSTEAD OF and AFTER triggers on INSERT, UPDATE, and DELETE. (Microsoft includes the FOR keyword, but this is a synonym for AFTER in Microsoft syntax.) Thus, we have six possible trigger types for use in SQL Server 2016.

Oracle Database XE supports all nine trigger types. MySQL 5.7 supports only BEFORE and AFTER triggers—thus, it supports only six trigger types. Other DBMS products support triggers differently. See the documentation of your product to determine which trigger types it supports.

Trigger Type / DML Action	BEFORE	INSTEAD OF	AFTER
INSERT	Oracle Database MySQL	Oracle Database SQL Server	Oracle Database SQL Server MySQL
UPDATE	Oracle Database MySQL	Oracle Database SQL Server	Oracle Database SQL Server MySQL
DELETE	Oracle Database MySQL	Oracle Database SQL Server	Oracle Database SQL Server MySQL

Figure E-33 — Summary of SQL Triggers by DBMS Product

When a trigger is invoked, the DBMS makes the data involved in the requested action available to the trigger code. For an insert, the DBMS will supply the values of columns for the row that is being inserted. For deletions, the DBMS will supply the values of columns for the row that is being deleted. For updates, it will supply both the old and the new values. The way in which this is done depends on the DBMS product.

While a full discussion of triggers is beyond the scope of this book, we will note that triggers have many uses, and four common uses for triggers are

- Providing default values.
- Enforcing data constraints.
- Updating SQL views.
- Performing referential integrity actions.

For more information about SQL triggers and how to use them, see David M. Kroenke and David J. Auer, *Database Processing: Fundamentals, Design, and Implementation*, 14th edition (Upper Saddle River, NJ: Pearson, 2016).

SQL/PSM Stored Procedures

A **stored procedure** is a program that is stored within the database and compiled when used. In Oracle Database, stored procedures can be written in PL/SQL or in Java. With SQL Server 2012, stored procedures are written in T-SQL or a .NET CLR language, such as Visual Basic.NET, C#.NET, or C++.NET. With MySQL, stored procedures are written in MySQL's variant of SQL.

Stored procedures can receive input parameters and return results. Unlike triggers, which are attached to a given table or view, stored procedures are attached to the database. They can be executed by any process using the database that has permission to use the procedure. Differences between triggers and stored procedures are summarized in Figure E-34.

Triggers Versus Stored Procedures
Trigger
Module of code that is called by the DBMS when INSERT, UPDATE, or DELETE commands are issued.
Assigned to a table or view.
Depending on the DBMS, may have more than one trigger per table or view.
Triggers may issue INSERT, UPDATE, and DELETE commands and thereby may cause the invocation of other triggers.
Stored Procedure
Module of code that is called by a user or database administrator.
Assigned to a database, but not to a table or a view.
Can issue INSERT, UPDATE, DELETE, and MERGE commands.
Used for repetitive administration tasks or as part of an application.

Figure E-34 — Triggers Versus Stored Procedures

Stored procedures are used for many purposes. Although database administrators use them to perform common administration tasks, their primary use is within database applications. They can be invoked from application programs written in languages such as COBOL, C, Java, C#, or C++. They also can be invoked from Web pages using VBScript, JavaScript, or PHP. Ad hoc users can run them from DBMS management products such as SQL*Plus or SQL Developer in Oracle Database, SQL Server Management Studio in SQL Server, or the MySQL Workbench in MySQL.

While a full discussion of stored procedures is beyond the scope of this book, we will note that there are many advantages of using stored procedures. These are listed in Figure E-35.

Unlike application code, stored procedures are never distributed to client computers. They always reside in the database and are processed by the DBMS on the database server. Thus, they are more secure than distributed application code, and they also reduce network traffic. Increasingly, stored procedures are the preferred mode of processing application logic over the Internet or corporate intranets. Another advantage of stored procedures is that their SQL statements can be optimized by the DBMS compiler.

When application logic is placed in a stored procedure, many different application programmers can use that code. This sharing results not only in less work but also in standardized processing. Further, the developers best suited for database work can create the stored procedures while other developers, say, those who specialize in Web-tier programming, can do other work. Because of these advantages, it is likely that stored procedures will see increased use in the future.

For more information about SQL stored procedures and how to use them, see David M. Kroenke and David J. Auer, *Database Processing: Fundamentals, Design, and Implementation*, 14th edition (Upper Saddle River, NJ: Pearson, 2016).

Advantages of Stored Procedures
Greater security.
Decreased network traffic.
SQL can be optimized.
Code sharing.
Less work.
Standardized processing.
Specialization among developers.

Figure E-35 — Advantages of Stored Procedures

Importing Microsoft Excel Data into a Database Table

Because most users don't understand databases, they will typically build a Microsoft Excel worksheet (called a *spreadsheet* in other electronic **spreadsheet** products such as Apache OpenOffice Calc) to store the data that they work with. This often results in a database developer needing to import the Microsoft Excel data into one or more tables in a database.

The techniques for importing Microsoft Excel data vary considerably from DBMS to DBMS. For a discussion of how to import Microsoft Excel 2016 data into Microsoft SQL Server 2016, Oracle Database XE, and MySQL 5.7 refer to:

- For Microsoft SQL Server 2016, see Appendix A, "Getting Started with Microsoft SQL Server 2016."
- For Oracle Database XE, see Appendix B, "Getting Started with Oracle Database XE."
- For MySQL 5.7, see Appendix C, "Getting Started with MySQL 5.7 Community Server."

For a discussion of how to import Microsoft Excel 2016 data into Microsoft Access 2016, see the section of "The Access Workbench" later in this appendix.

Using Microsoft Access 2016 as a Development Platform

While we often prefer an enterprise-class DBMS product such as Microsoft SQL Server 2016, Oracle Database XE, or MySQL 5.7 for creating databases rather than a personal database product such as Microsoft Access 2016, none of the enterprise class DBMSs provides application development tools (at least not as part of the DBMS product itself). This means that application developers need some tool or tools to develop forms, reports, and other components of an application front end to a database.

One of the advantages of Microsoft Access 2016 is that it *does* include application development tools. In fact, we have used these tools to create forms, reports, and other application components (see Appendix H, "The Access Workbench—Section H—Microsoft Access 2016 Switchboards") for the WMCRM database we've built in the various sections of "The Access Workbench."

Is it possible to use the Microsoft Access 2016 application development tools with a database that exists in another DBMS such as Microsoft SQL Server 2016, Oracle Database XE, or MySQL 5.7? The answer is *yes*: We can connect a Microsoft Access 2016 database as a development platform to databases in these DBMS products using a tool known as **Open Database Connectivity (ODBC)**. We will discuss ODBC in detail in Chapter 7, where we will use it to construct Web-based database applications.

For a discussion of how to use ODBC to link Microsoft Access 2016 to Microsoft SQL Server 2016, Oracle Database XE, and MySQL 5.7, refer to:

- For Microsoft SQL Server 2016, see Appendix A, "Getting Started with Microsoft SQL Server 2016."
- For Oracle Database XE, see Appendix B, "Getting Started with Oracle Database XE."
- For MySQL 5.7, see Appendix C, "Getting Started with MySQL 5.7 Community Server."

Section E

Advanced SQL in Microsoft Access and Importing Microsoft Excel Data

In Chapter 3's section of "The Access Workbench," you learned to work with Microsoft Access SQL and QBE. In this section, you'll learn how to use some of the advanced SQL statements discussed in the Appendix including:

- How to use the SQL ALTER TABLE statement.
- How to create and use a query on a recursive relationship.
- How to create and use the Microsoft Access equivalent of SQL views.

In addition, we will describe:

- How to import Microsoft Excel 2016 data into a Microsoft Access Table.

We'll continue to use the WMCRM database we have been using. At this point, we have created and populated (which means we've inserted the data into) the CUSTOMER, PHONE_NUMBER, CONTACT, and SALESPERSON tables and have set the referential integrity constraints between them.

Working with the SQL ALTER TABLE Statement in Microsoft Access

We introduced the SQL ALTER TABLE statement in Chapter 3 and then used it in that chapter's section of "The Access Workbench." There we used it both to add the NOT NULL column NickName to the CUSTOMER table and to add a CHECK constraint to the CONTACT table. We simply walked through the steps, and only in this appendix did we cover these uses of the ALTER TABLE statement in depth. Nonetheless, if you worked through all of Chapter 3's section of "The Access Workbench," you have used the ALTER TABLE statement before, and it should seem familiar to you.

Here we will use it to add a column to the CUSTOMER table that we will need in order to create a query on a recursive relationship. At Wallingford Motors, one way the new customers are found is by the recommendation of existing customers. When a sale is made to a customer, that customer is asked to recommend one (and only one) potential customer. If the customer makes such a recommendation, he or she is rewarded with a 3 percent reduction in the sales price of the car he or she is buying. Note that this recommendation is not a requirement, and not all customers are comfortable giving Wallingford Motors the name of someone they know.

Column Name	Type	Key	Required	Remarks
ReferredBy	Integer	Foreign Key	No	UNIQUE REF: CustomerID in CUSTOMER

Figure AW-E-1 — WWCRM Database Column Characteristics for the CUSTOMER ReferredBy Column

CustomerID	LastName	FirstName	...	ReferredBy
1	Griffey	Ben	...	NULL
3	Christman	Jessica	...	NULL
4	Christman	Rob	...	3
5	Hayes	Judy	...	1

Figure AW-E-2 — WMCRM CUSTOMER Table ReferredBy Data

To record this data, we must add a ReferredBy column to the CUSTOMER table. As shown in Figure AW-E-1, the column will be integer data (because CustomerID is integer) and not required (or NULL) (because not all customers were recommended by other customers). Because customers can only refer one other person, the data values of ReferredBy must be unique (each CustomerID value can only appear once in the ReferredBy column). Further, this column will have a recursive relationship within the CUSTOMER table because ReferredBy will be a foreign key referencing CustomerID. Figure AW-E-2 shows the data for the ReferredBY column that will need to be added after the column is created.

Let's take this one step at time. First, we add the NULL column ReferredBy. To do this, we will use the SQL ALTER TABLE statement:

```
/* *** SQL-ALTER-TABLE-AppE-AW-01 *** */
ALTER TABLE CUSTOMER
   ADD ReferredBy INTEGER NULL;
```

Then we use SQL UPDATE statements to add new data to the column:

```
/* *** SQL-UPDATE-AppE-AW-01 *** */
UPDATE CUSTOMER
      SET      ReferredBy = 3
      WHERE    CustomerID = 4;
/* *** SQL-UPDATE-AppE-AW-02 *** */
UPDATE CUSTOMER
      SET      ReferredBy = 1
      WHERE    CustomerID = 5;
```

Creating and Populating the ReferredBy Column in the CUSTOMER Table by using Microsoft Access SQL

1. Start Microsoft Access 2016, and open the **WMCRM.accdb** database.
2. Open an Access Query window in SQL View as discussed in Chapter 3.
3. Type the SQL-ALTER-Table-AppE-AW-01 statement shown above into the query window.
4. Click the **Run** button.
   - **NOTE:** The only indication that the command has run successfully is the fact that *no* error message is displayed.
5. Type the SQL-UPDATE-AppE-AW-01 statement shown above into the query window.
6. Click the **Run** button. When the dialog box appears asking you to confirm that you want to insert the data, click the **Yes** button in the dialog box. The data are inserted into the table.
7. Type in the SQL-UPDATE-AppE-AW-02 statement shown above into the query window.
8. Click the **Run** button. When the dialog box appears asking you to confirm that you want to insert the data, click the **Yes** button in the dialog box. The data are inserted into the table.
9. Close the Query1 window. A dialog box appears asking if you want to save the query. Click the **No** button—there is no need to save this SQL statement.
10. Open the CUSTOMER table.
11. Click the **Shutter Bar Open/Close** button, if needed, to minimize the Navigation Pane and then scroll to the right so that the added ReferredBy column and the data in it are displayed. The table looks as shown in Figure AW-E-3.
12. Click the **Shutter Bar Open/Close** button to expand the Navigation Pane if necessary, and then click the **Design View** button to switch the CUSTOMER table into Design view.
13. Click the **ReferredBy** field name to select it.
14. Set the **Indexed** field property setting to **Yes (no Duplicates)**, as shown in Figure AW-E-4.
15. Note that the Required field property setting is set to *No*—this is the Microsoft Access equivalent of *NULL*.
16. Close the CUSTOMER table. If a dialog box appears asking "Do you want to save changes to the layout of table CUSTOMER?" click the **Yes** button.

Why did we set the Indexed property for the ReferredBy column to Yes (No Duplicates)? We did this because this is how we enforce the requirement that values in this column be unique. Further, as discussed in Chapter 5's section of "The Access Workbench" (see pages 348–354), this setting is necessary to create a 1:1 relationship instead of a 1:N relationship. We need this because the recursive relationship we are about to create needs to be a 1:1 relationship.

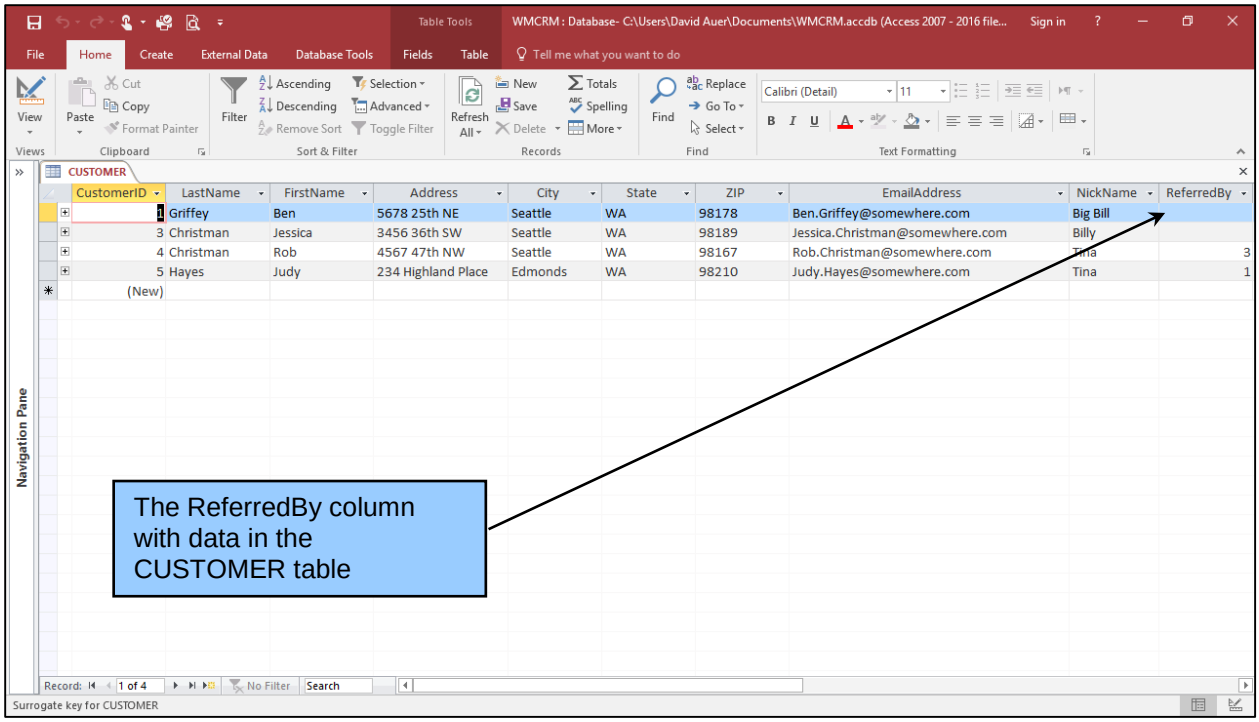


Figure AW-E-3 — The ReferredBy Column in the CUSTOMER Table

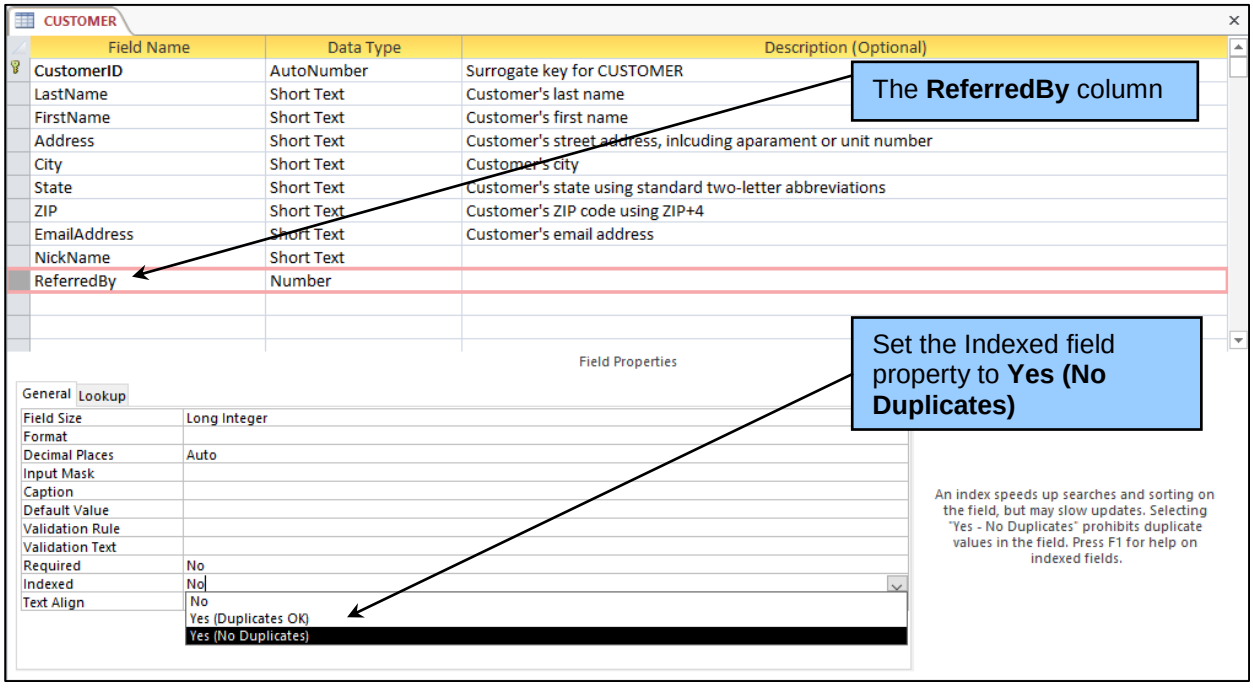


Figure AW-E-4 — Setting the Indexed Field Property to Yes (No Duplicates)

To create this recursive relationship, we will again use an SQL ALTER TABLE statement:

```
/* *** SQL-ALTER-TABLE-AppE-AW-02 *** */
ALTER TABLE CUSTOMER
      ADD CONSTRAINT REF_BY_CUSTOMER_FK FOREIGN KEY(ReferredBy)
            REFERENCES CUSTOMER(CustomerID);
```

Creating a 1:1 Recursive Relationship Within the CUSTOMER Table

1. Open an Access query window in SQL View.
2. Type the SQL code for the SQL-ALTER-TABLE-AppE-AW-02 statement into the query window.
3. Click the **Run** button.
   - **NOTE:** As before, the only indication that the command has run successfully is the fact that *no* error message is displayed.
4. Close the Query1 window. A dialog box appears asking if you want to save the query. Click the **No** button—there is no need to save this SQL statement.
5. Click the **Shutter Bar Open/Close** button to minimize the Navigation Pane.
6. Click the **Database Tools** command tab.
7. Click the **Relationships** button in the Relationships group. The Relationships tabbed document window appears, as shown in Figure AW-E-5.
8. In Figure AW-E-5, note how Microsoft Access displays recursive relationships—a new table named CUSTOMER_1 has been introduced. This is a virtual table, used to allow Microsoft Access to create a relationship between two tables!
9. Also notice that the relationship line shows a “many” symbol where it should show a “1” symbol because this is a 1:1 relationship.
10. To verify that you do in fact have the correct 1:1 relationship, **right-click** the relationship line between CUSTOMER and CUSTOMER_1, and then click **Edit Relationship** in the shortcut menu that appears. The Edit Relationships dialog box appears.
11. Note that the correct 1:1 relationship between the tables is confirmed in the Relationships window, as shown in Figure AW-E-6.
12. Click the **OK** button in the Edit Relationships dialog box to close it.
13. Click the **Save** button to save any changes.
14. Close the Relationships window.

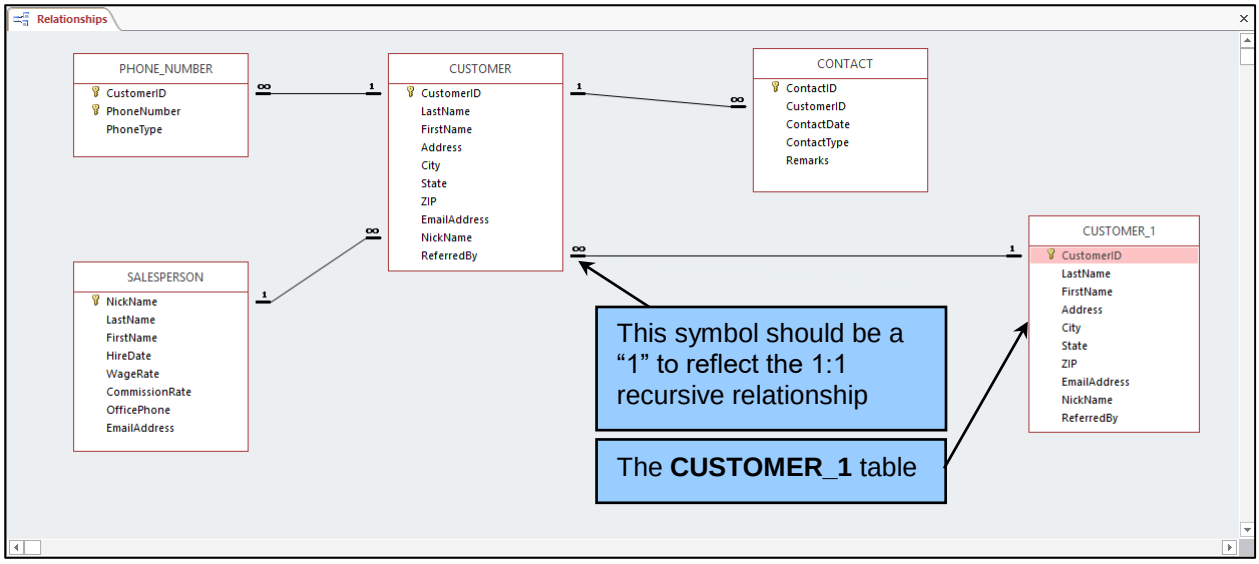


Figure AW-E-5 — The 1:1 Recursive Relationship Within the CUSTOMER Table

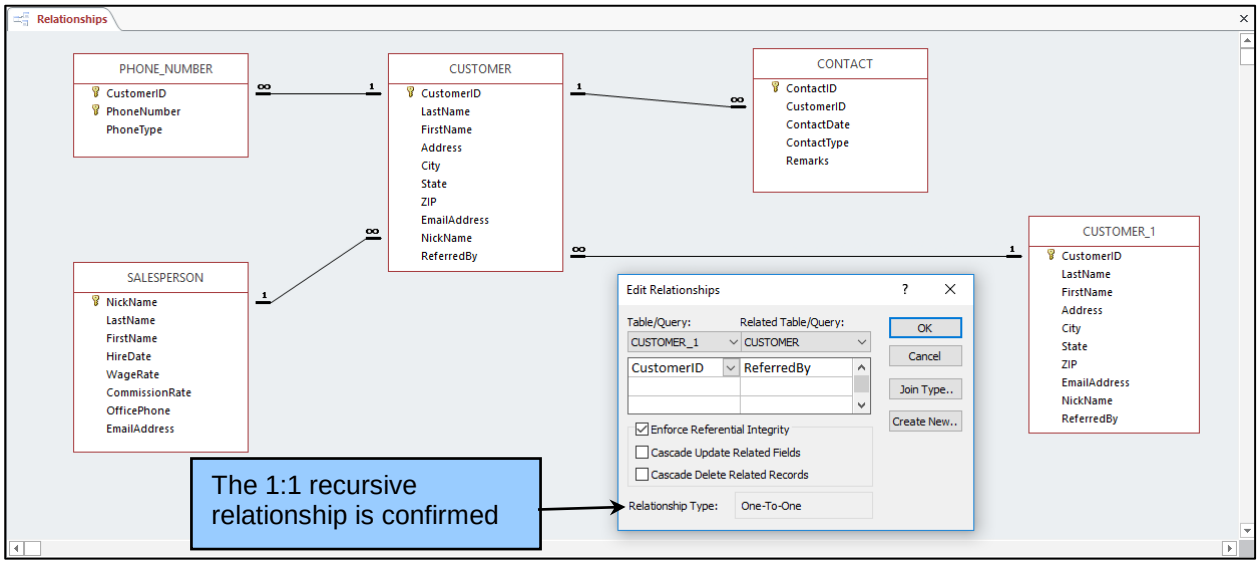


Figure AW-E-6 — The 1:1 Recursive Relationship Is Confirmed

Working with Queries on Recursive Relationships in Microsoft Access

Now that we have added the ReferredBy column to the CUSTOMER table, populated the column with data, and established the recursive relationship and referential integrity constraint within the CUSTOMER table, we can create a query on the recursive relationship in the CUSTOMER table.

We'll do this with an SQL query, and then take a look at the QBE graphical interpretation of it. In our WMCRM database, the CUSTOMER table has a 1:1 recursive relationship between ReferredBy and

CustomerID. One customer can refer only one other person as a potential customer. We can visually examine these relationships in the CUSTOMER table by using SQL-Query-AppE-AW-01, which we will save under that name in the WMCRM database.

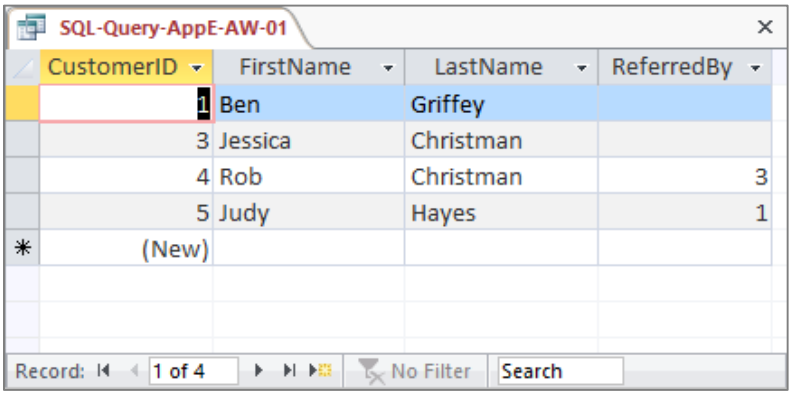
```
/* *** SQL-Query-AppE-AW-01 *** */
SELECT      CustomerID, FirstName, LastName, ReferredBy
FROM        CUSTOMER
ORDER BY    CustomerID;
```

Creating and Running the SQL-Query-AppE-AW-01 SQL Query

1. Click the **Create** command tab to display the Create command groups
2. Click the **Query Design** button.
3. The Query1 tabbed document window is displayed in Design view, along with the Show Table dialog box.
4. Click the **Close** button on the Show Table dialog box.
5. Click the **SQL View** button in the Results group on the Design tab. The Query1 window switches to the SQL view.
6. Edit the SQL SELECT command to read:

```
SELECT      CustomerID, FirstName, LastName, ReferredBy
FROM        CUSTOMER
ORDER BY    CustomerID;
```

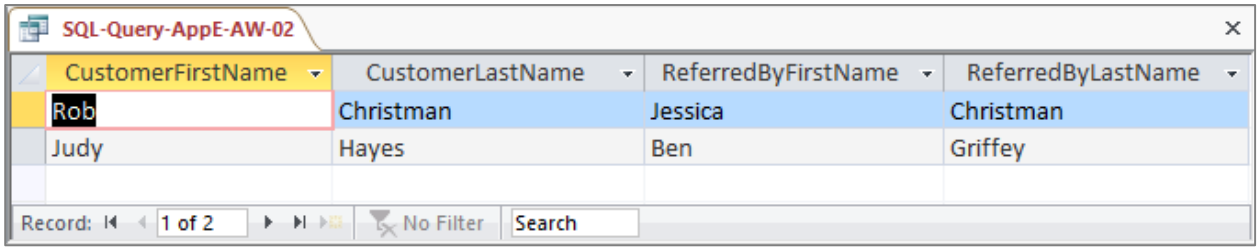
7. Click the **Save** button on the Quick Access Toolbar. The Save As dialog box appears.
8. Type in the query name SQL-Query-AppE-AW-01, and then click the **OK** button. The query is saved, and the query window is renamed with the query name.
9. Click the **Run** button on the Design tab. The query result is:



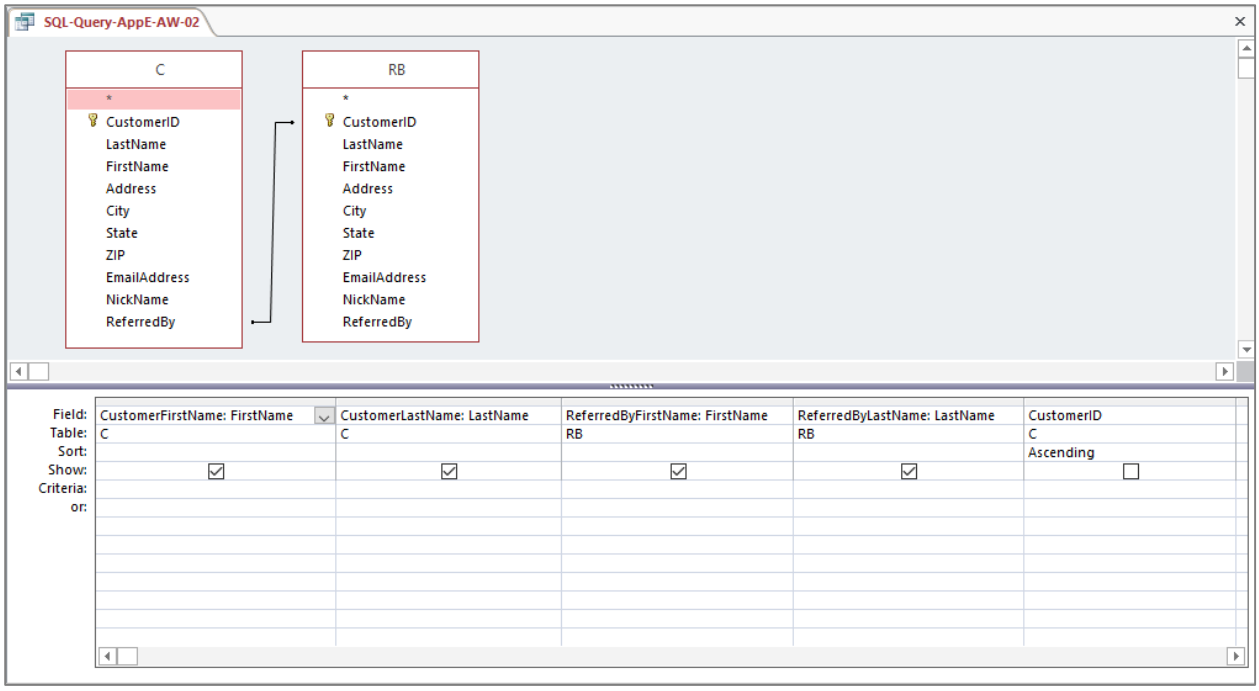
Given the result, we can easily see that Ben Griffey referred Judy Hayes to Wallingford Motors, while Rob Christman was referred by Jessica Christman. To specifically list who referred whom, we can use query SQL-Query-AppE-AW-02.

```
/* *** SQL-Query-AppE-AW-02 *** */
SELECT      C.FirstName AS CustomerFirstName,
            C.LastName AS CustomerLastName,
            RB.FirstName AS ReferredByFirstName,
            RB.LastName AS ReferredByLastName
FROM        CUSTOMER AS C INNER JOIN CUSTOMER AS RB
   ON       C.ReferredBy = RB.CustomerID
ORDER BY    C.CustomerID;
```

The process for creating, saving, and running this query is basically the same as the one we used for SQL-Query-AppE-AW01 above. The result clearly shows which customers were referred by other customers:



While we created SQL-Query-AppE-AW-02 as an SQL query, we can look at how it appears in Microsoft Access QBE. Switching the SQL-Query-AppE-AW-02 window to Design View shows us how that looks:



Working with SQL Views in Microsoft Access

Although a view is a virtual table, it can also be represented as a stored query. Although most DBMSs do not allow queries to be saved in a database, Access does. Access allows us to run queries against tables or against saved queries. This gives us a way to implement a **Microsoft Access equivalent of an SQL view**: We simply save the SELECT query that would be used to create the SQL view and use it as we would an SQL view in other queries.

We'll use a view to solve a very real problem in the WMCRM database: the fact that customer's phone numbers are stored in a separate table (PHONE_NUMBER) from the rest of the customer data (which is in CUSTOMER). We created the PHONE_NUMBER table in Chapter 2's section of "The Access Workbench" to resolve a **multivalue, multicolumn problem** that existed in the original CUSTOMER table.

Now, having fixed the multivalue, multicolumn problem, we are faced with the problem of recombining the data in the two tables when a user wants to see the customer's phone data together with other customer data at the sametime. Of course, this is easy to do with an SQL query if you know SQL:

```
/* *** SQL-Query-AppE-AW-03 *** */
SELECT      FirstName, LastName, EmailAddress,
            PhoneNumber AS CustomerPhoneNumber,
            PhoneType
FROM        CUSTOMER INNER JOIN PHONE_NUMBER
      ON    CUSTOMER.CustomerID = PHONE_NUMBER.CustomerID
ORDER BY    LastName, FirstName;
```

SQL-Query-AppE-AW-03 produces the desired result—note that there are multiple rows in the table for customers with more than one phone number:

SQL-Query-AppE-AW-03

FirstName	LastName	EmailAddress	CustomerPhoneNumber	PhoneType
Jessica	Christman	Jessica.Christman@somewhere.com	206-467-3456	
Rob	Christman	Rob.Christman@somewhere.com	206-478-9998	Fax
Rob	Christman	Rob.Christman@somewhere.com	206-478-4567	
Ben	Griffey	Ben.Griffey@somewhere.com	206-765-5678	Cell
Ben	Griffey	Ben.Griffey@somewhere.com	206-456-2345	Home
Judy	Hayes	Judy.Hayes@somewhere.com	425-354-8765	

Record: 1 of 6 No Filter Search

However, this is not an SQL query that we would want a typical user to construct. One of the uses of SQL views is to "hide complicated SQL syntax" from users, and this situation is a great place to use an SQL view for just that purpose.

Here is an SQL CREATE VIEW statement that would be used to list this customer data from the WMCRM database if we were creating a standard SQL view (note that we have removed the ORDER BY clause):

```
/* *** SQL-CREATE-VIEW-AppE-AW-01 *** */
CREATE VIEW CustomerPhoneView AS
   SELECT      FirstName, LastName, EmailAddress,
               PhoneNumber AS CustomerPhoneNumber,
               PhoneType
   FROM        CUSTOMER INNER JOIN PHONE_NUMBER
         ON    CUSTOMER.CustomerID = PHONE_NUMBER.CustomerID
```

Since we cannot create SQL Views *directly as SQL Views* in Access, we instead create a query—using either Access SQL or QBE—based on the SELECT portion of this statement. Note that because this is just another query as far as Microsoft Access 2016 is concerned, we can and do include the ORDER BY clause! Note that this query is identical to SQL-Query-AppE-AW-03.

```
/* *** SQL-Query-AppE-AW-04 *** */
SELECT      FirstName, LastName, EmailAddress,
            PhoneNumber AS CustomerPhoneNumber,
            PhoneType
FROM        CUSTOMER INNER JOIN PHONE_NUMBER
      ON    CUSTOMER.CustomerID = PHONE_NUMBER.CustomerID
ORDER BY    LastName, FirstName;
```

After we create the query, we save it using a query name that indicates that *this query is intended to be used as an SQL view*. We can use the naming convention of putting the word view at the beginning of any such query name. Thus, we can name this query *viewCustomerPhone*.

Although we can include the ORDER BY clause, it is a little tricky because Microsoft Access QBE sorts in the left-to-right order of the columns as listed. In our case, if we create the query directly in QBE and request sorting on the first two columns (FirstName and LastName), it would sort by FirstName and then LastName instead of LastName and then First Name. Fortunately, there is a way to get around this in QBE that we will demonstrate (this problem does not occur if we just use an SQL statement).

Creating a Microsoft Access Query as a View Equivalent

1. Click the **Create** command tab to display the Create command groups.
2. Click the **Query Design** button.
3. The Query1 tabbed document window is displayed in Design view, along with the Show Table dialog box.
4. Using either the SQL or QBE technique of creating queries, as described in Chapter 3's section of "The Access Workbench," create the query, and then click on Design View if needed to complete the query in Design View:

```
SELECT      FirstName, LastName, EmailAddress,
            PhoneNumber AS CustomerPhoneNumber,
            PhoneType
FROM        CUSTOMER INNER JOIN PHONE_NUMBER
      ON    CUSTOMER.CustomerID = PHONE_NUMBER.CustomerID;
```

- **NOTE:** As shown in Figure AW-E-7, to create the QBE equivalent of **`EmailAddress AS CustomerEmailAddress`**, enter the *alias name* followed by a *colon [:]* and then followed by the *column name* that is being aliased.

- **NOTE:** As shown in Figure AW-E-7, to create the correct sort order on QBE, include another instance of **FirstName**, and set the sort order here. Then set this column *to not appear in the query results* by unchecking the **Show** checkbox.

7. To save the query, click the **Save** button on the Quick Access Toolbar. The Save As dialog box appears.

8. Type in the query name viewCustomerPhone, and then click the **OK** button. The query is saved, the document window is renamed with the query name, and a newly created viewCustomerPhone query object appears in the Queries section of the Navigation Pane, as shown in Figure AW-E-7. Note that Figure AW-E-7 shows the query created in Access QBE.

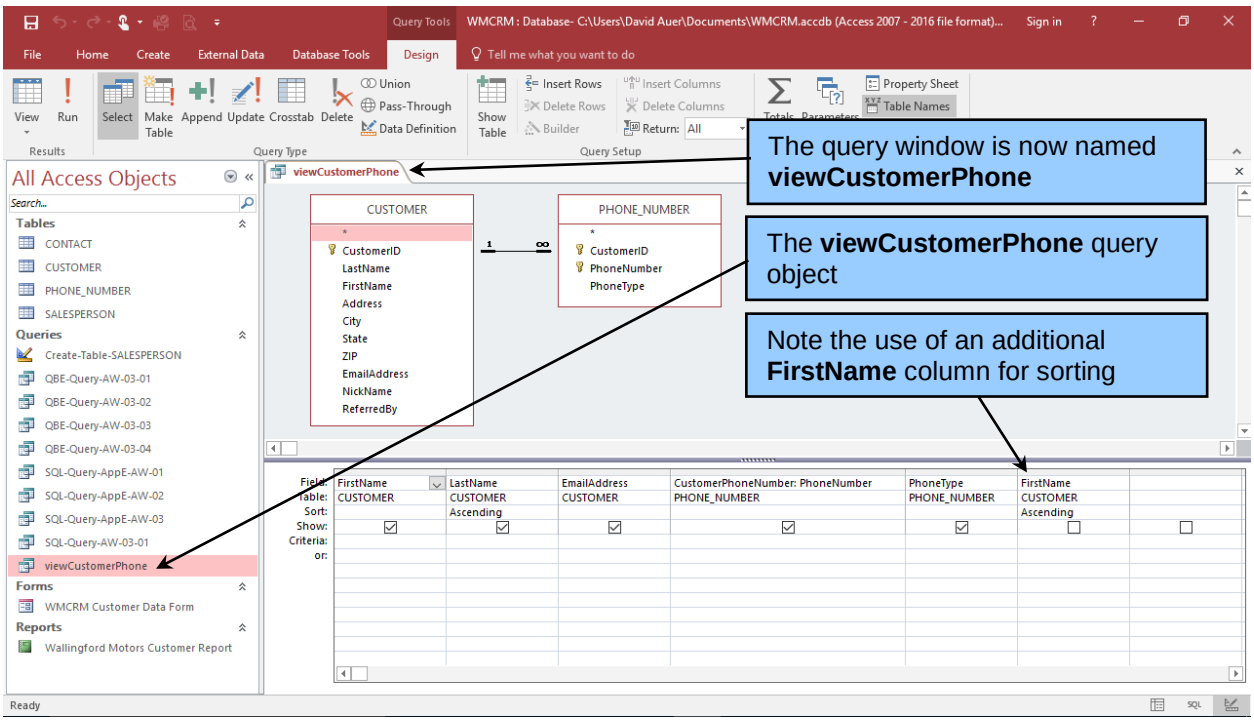


Figure AW-E-7 — The viewCustomerPhone Query in the Queries Pane

9. Close the viewCustomerPhone window by clicking the document window's **Close** button.
10. If Access displays a dialog box that asks whether you want to save changes to the design of query viewCustomerPhone, click the **Yes** button.

Now we can use the viewCustomerPhone query just as we would any other SQL view (or Access saved query). For example, we can implement the following SQL statement:

```
/* *** SQL-Query-AppE-AW-05 *** */
SELECT      FirstName, LastName, EmailAddress, CustomerPhoneNuber,
            PhoneType
FROM        viewCustomerPhone;
```

We'll use Access QBE in this example.

Using an Access Query in Another Access Query

1. Click the **Create** command tab to display the Create command groups.
2. Click the **Query Design** button.
3. The Query1 tabbed document window is displayed in Design view, along with the Show Table dialog box.
4. In the Show Table dialog box, click the **Queries** tab to select it. The list of all saved queries appears, as shown in Figure AW-E-8.
5. Click **viewCustomerPhone** to select the viewCustomerPhone query. Click the **Add** button to add the viewCustomerPhone query to the new query.

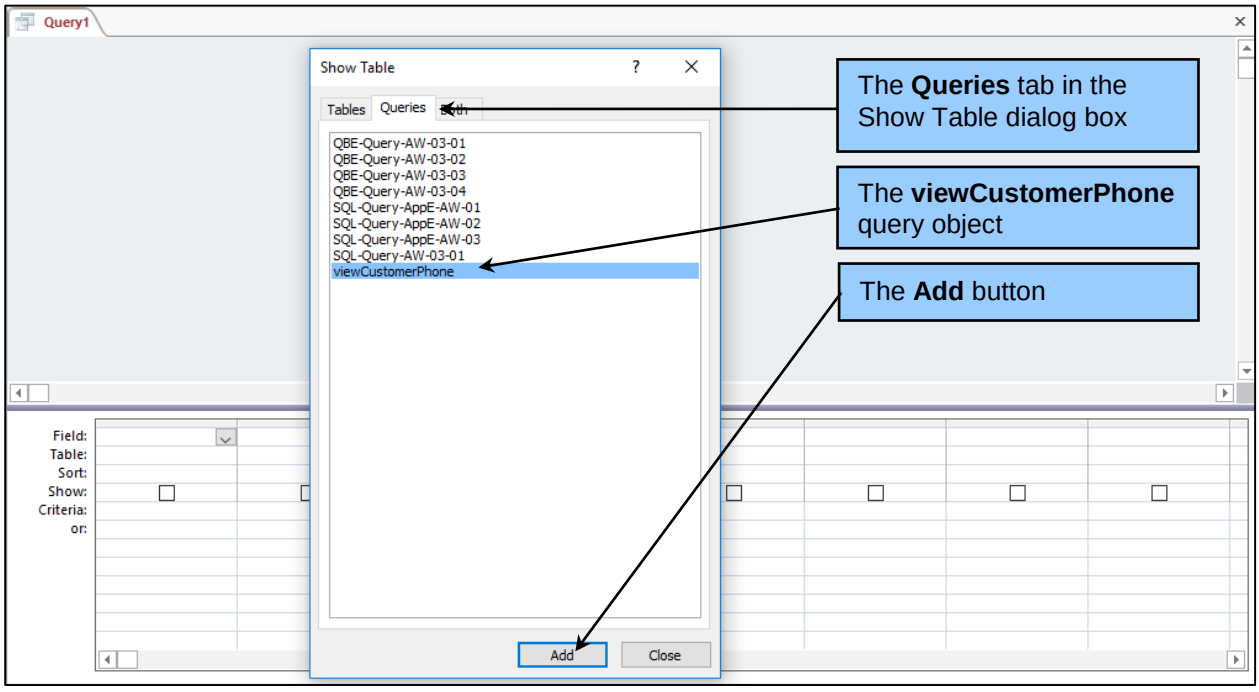


Figure AW-E-8 — Queries Displayed in the Show Table Dialog Box

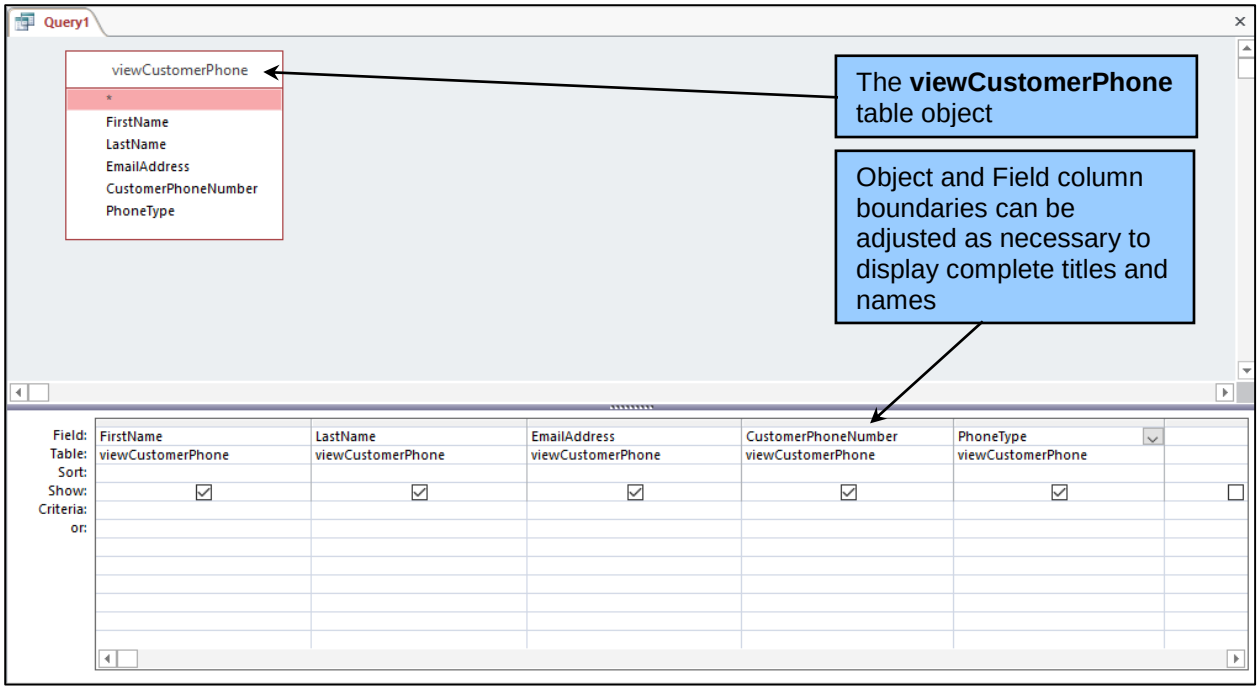


Figure AW-E-9 — The Completed QBE Query

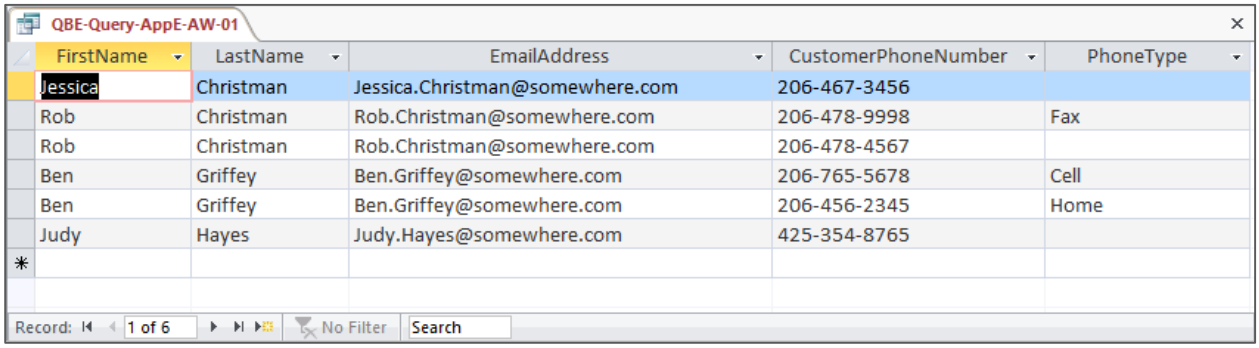


Figure AW-E-10 — The Completed Query Result

6. Click the **Close** button to close the Show Table dialog box.

7. From the viewCustomerPhone query, click and drag the **LastName**, **FirstName**, **EmailAddress**, **CustomerPhoneNumber,** and **PhoneType** column names to the first five field columns in the lower pane.

8. We do not need to set any Sort settings because they are included in viewCustomerPhone. The completed QBE query looks as shown in Figure AW-E-9. If necessary, resize the table object and the Field columns so that complete labels are displayed.

9. Save the query as **QBE-Query-AppE-AW-01**.

10. Click the **Run** button on the Design command tab. The query results appear as shown in Figure AW-E-10.

11. Close the QBE-Query-AppE-AW-01 query window.

Now we can use the equivalent of SQL views in Microsoft Access by using one query as the source for additional queries. That completes the work on SQL views that we'll do in this section of "The Access Workbench."

Working with SQL/PSM in Microsoft Access

Microsoft Access 2016 does not implement SQL/PSM as such. Corresponding capabilities can be found in Microsoft Access Visual Basic for Applications (VBA), but that topic is beyond the scope of this book.

Importing Microsoft Excel Data into in Microsoft Access 2016

To illustrate importing Microsoft Excel data into Microsoft Access 2016, we'll need a Microsoft Excel 2016 worksheet. Fortunately, the sales force at Wallingford Motors has been keeping details about Gaea model specifications in just such a worksheet, as shown in Figure AW-E-11.

However, this worksheet is problematic because it contains more than just the column names and data we will want to import. Therefore, we create an edited version as shown in Figure AW-E-12. Now we need to import this data into Microsoft Access 2016.

Importing Microsoft Excel Data into a Microsoft Access Table

1. Click the **EXTERNAL DATA** command tab to display the EXTERNAL DATA command groups.
2. As shown in Figure AW-E-13, click the **Excel** button to display the Get External Data – Excel Worksheet dialog box.

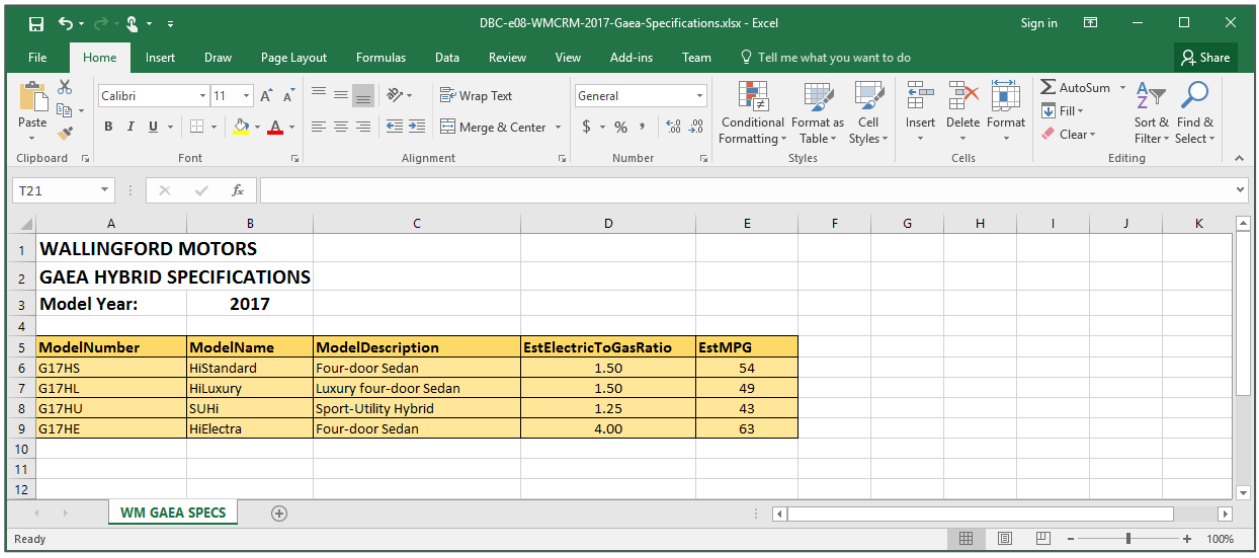


Figure AW-E-11 — The Wallingford Motors Gaea Specifications Workbook

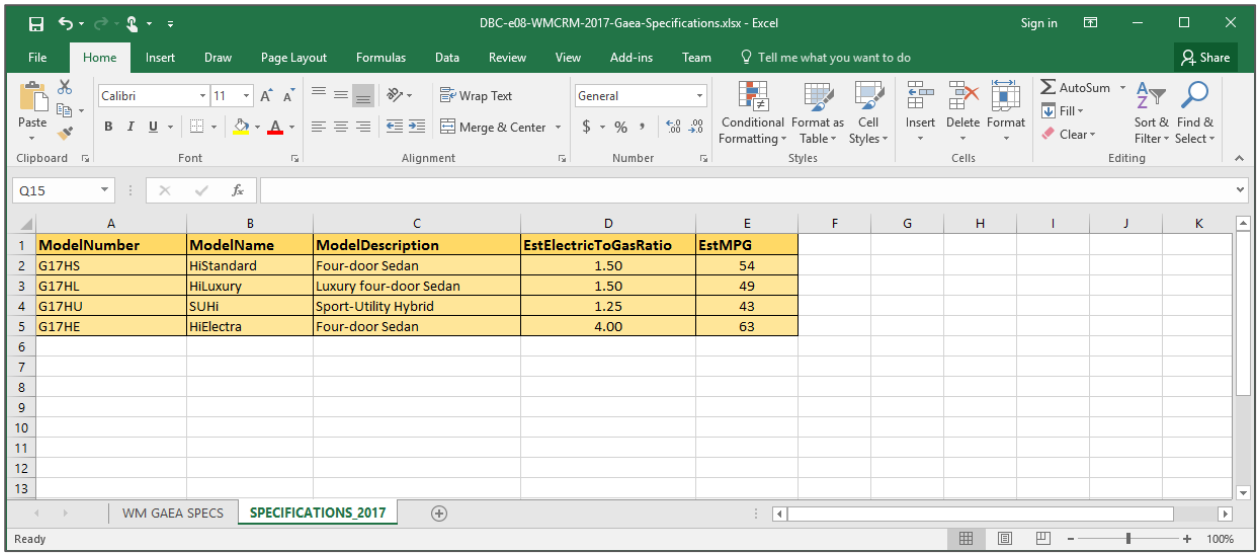


Figure AW-E-12 — The Revised Wallingford Motors Gaea Specifications Worksheet

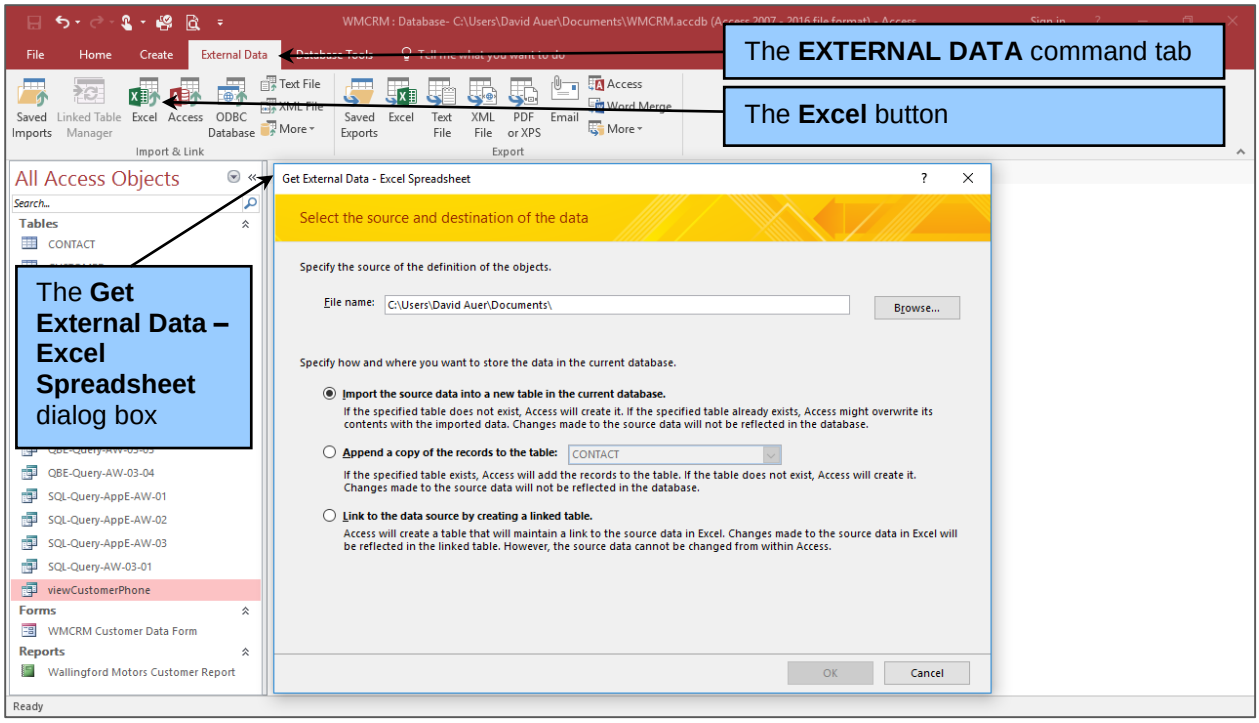


Figure AW-E-13 — The Get External Data – Excel Spreadsheet Dialog Box

3. Browse to the **DBC-e08-WMCRM-2017-Gaea-Specificiations.xlsx Microsoft Excel 2016** workbook, as shown in Figure AW-E-14. Leave the **Import the source data into a new table in the current database** radio button selected.

4. Click the **OK** button.

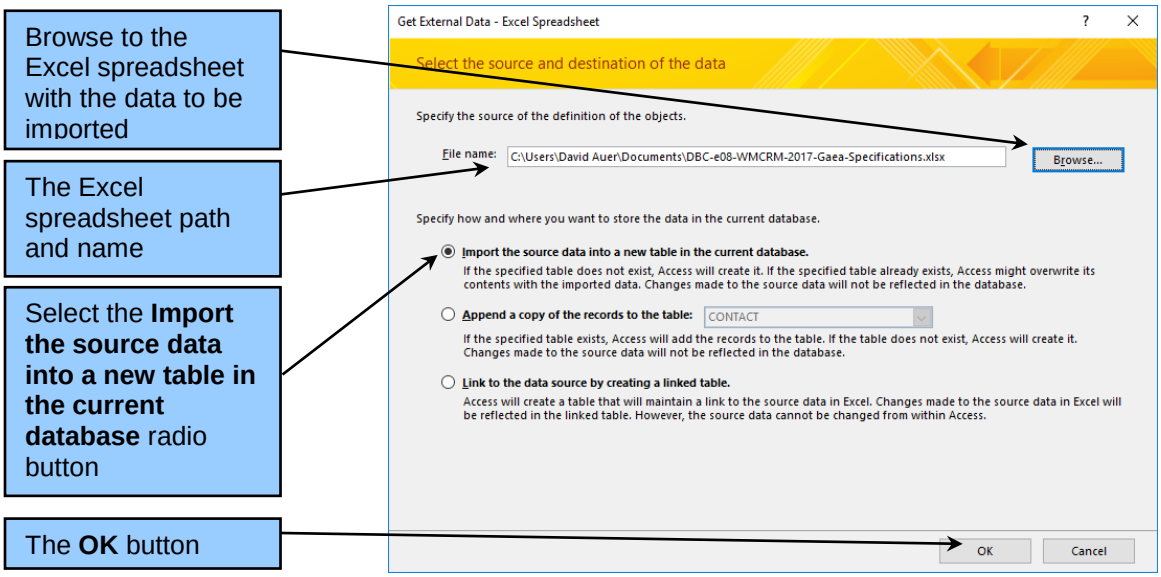


Figure AW-E-14 — The Get External Data – Excel Spreadsheet Dialog Box

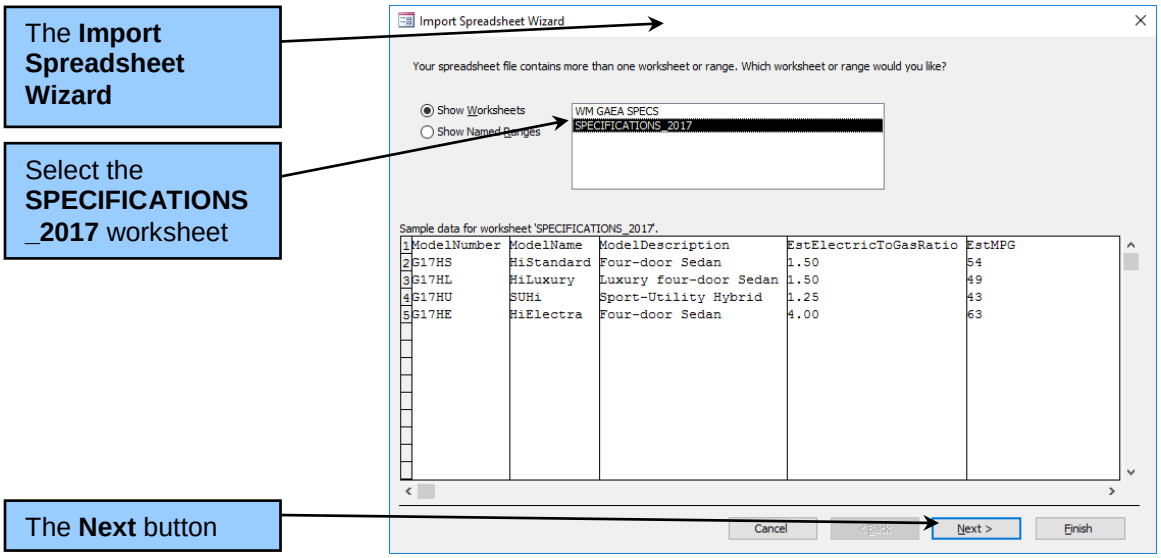


Figure AW-E-15 — The Import Spreadsheet Wizard Dialog Box

5. The Import Spreadsheet Wizard is launched. On the first page, select the **SPECIFICATIONS_2017** worksheet as shown in Figure AW-E-15.

6. Click the **Next** button.

7. On the next page of the Import Spreadsheet Wizard, make sure the **First Row Contains Column Headings** checkbox is checked, as shown if Figure AW-E-16.

8. Click the **Next** button.

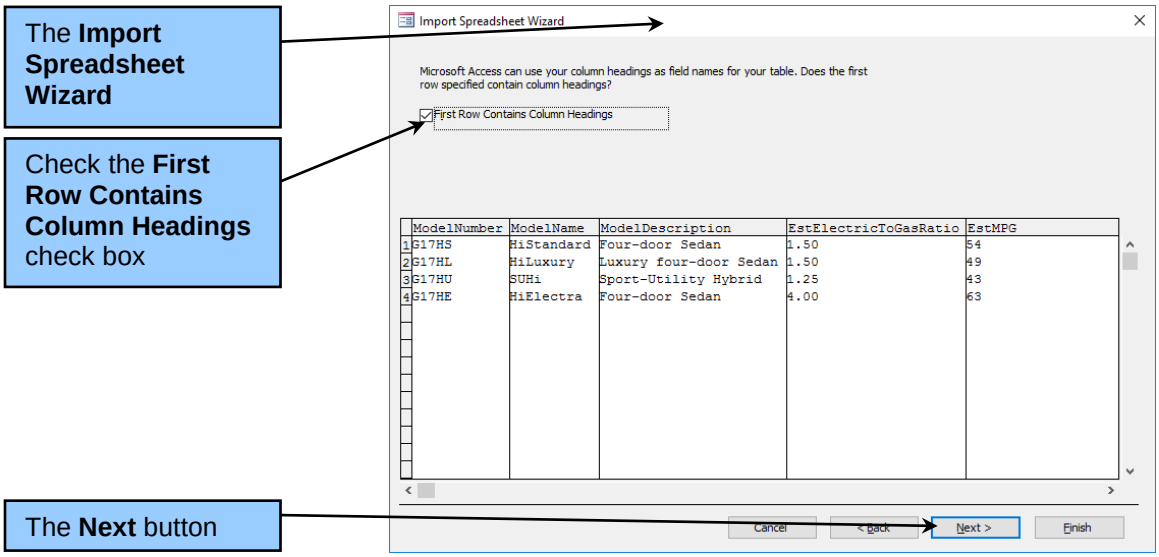


Figure AW-E-16 — The First Row Contains Column Headings Check Box

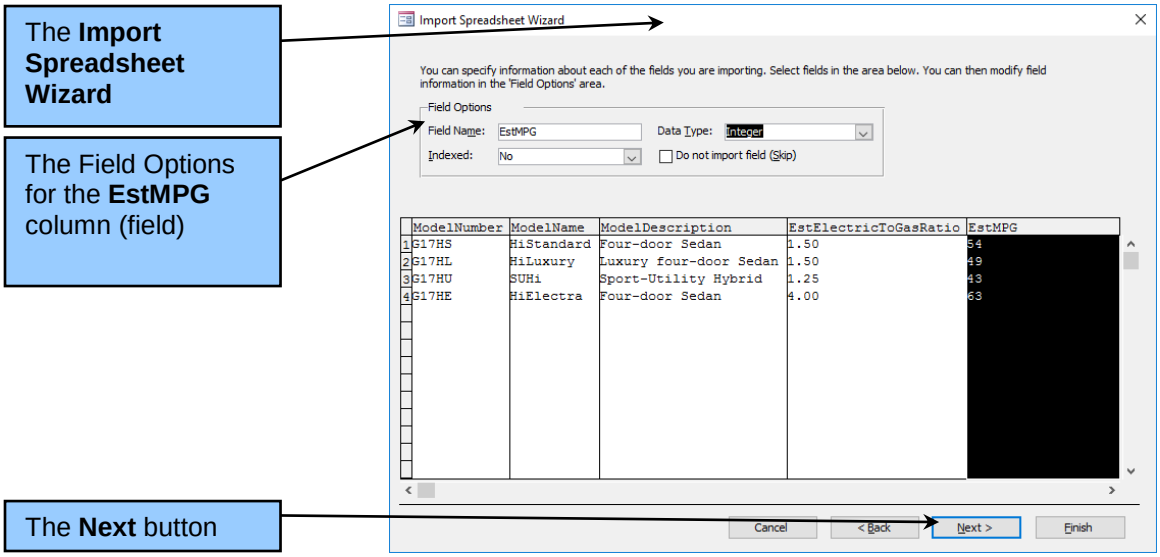


Figure AW-E-17 — The Field Options for the EstMPG Column (Field)

9. On the next page of the Import Spreadsheet Wizard, we are given a chance to review column (called field here) characteristics. For each field, we can set Field Name, DataType, and whether to index each column. We do not need to index. ModelNumber, ModelName, and ModelDescription will be **Short Text**, EstElectricToGasRatio will be **Single**, and ExtMPG will be **Integer**. Figure AW-E-17 shows the settings for the EstMPG field.

10. Click the **Next** button.

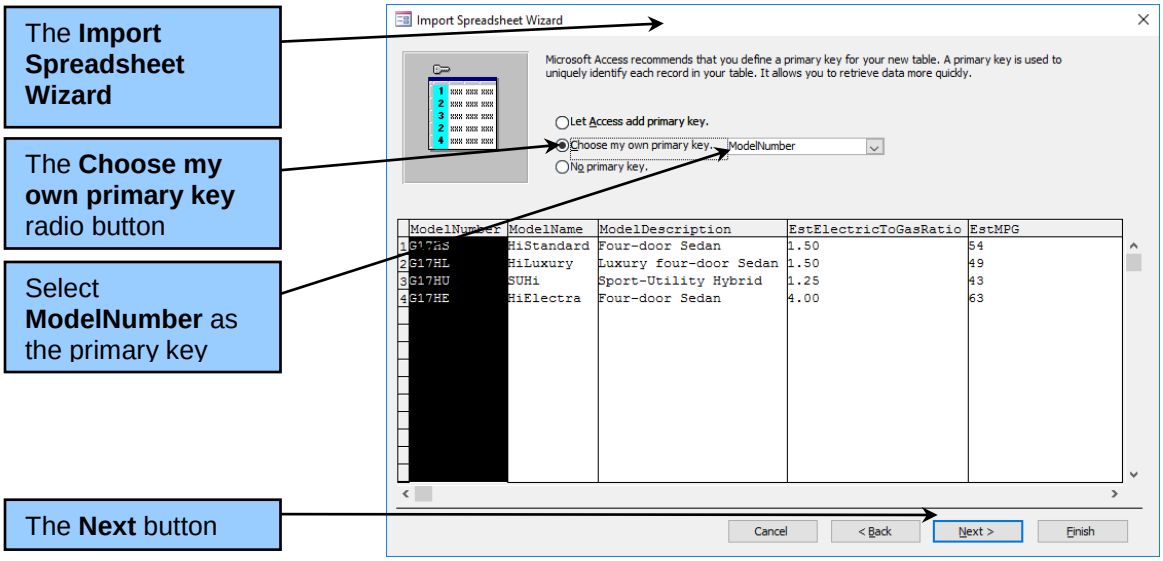


Figure AW-E-18 — Setting the Primary Key

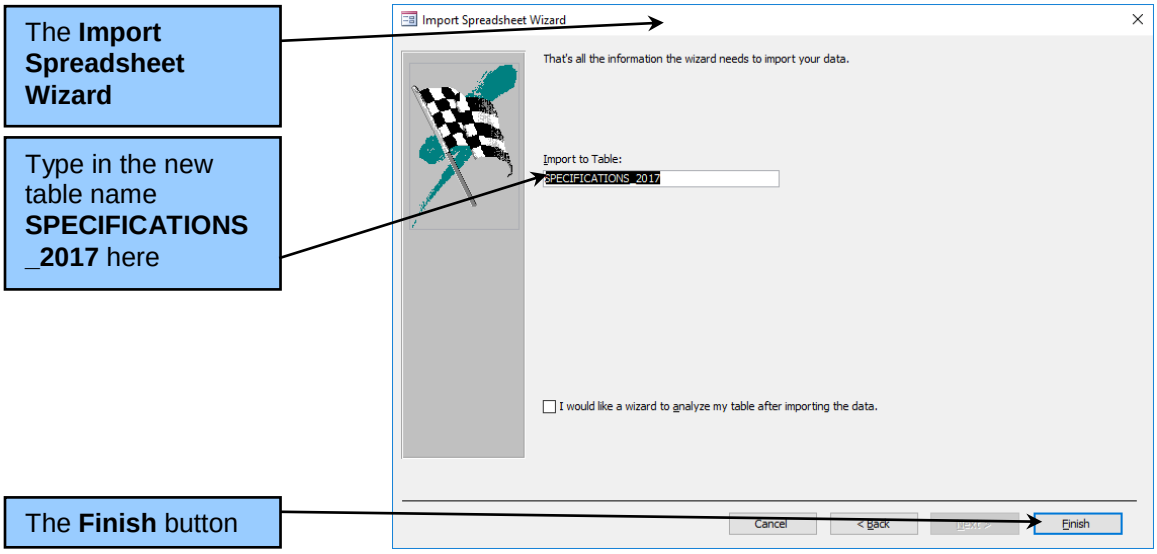


Figure AW-E-19 — Entering the New Table Name

11. On the next page we are given a chance to set a primary key. We will choose our own, and it will be ModelNumber, as shown in Figure AW-E-18.

12. Click the **Next** button.

13. On the next page we are given a chance to set a table name. The default is the Worksheet name, and, as shown in Figure AW-E-19, here it is **SPECIFICATIONS_2017**, which is what we want the table to be named. No changes are required here.

14. Click the **Finish** button.

15. The table and data are imported, and we are given a chance to save the import steps. There is no need to do this, so click the **Close** button to end the Wizard.

16. The imported SPECIFICATIONS_2017 table is shown in Datasheet View in Figure AW-E-20 (column widths have been adjusted to shown the complete column names).

17. Figure AW-E-21 shows the table in Design View. Note that the ModelNumber Short Text Field size is 255. We may want to edit that and other field characteristics. For example, we need to set the NULL/NOT NULL settings on all fields—this was not done during the import.

Although there is still some editing to do, we have successfully imported a Microsoft Excel 2016 worksheet into Microsoft Access 2016.

Closing the WMCRM Database and Exiting Access

1. To close the WMCRM database and exit Microsoft Access 2016, click the **Close** button in the upper-right corner of the Microsoft Access window.

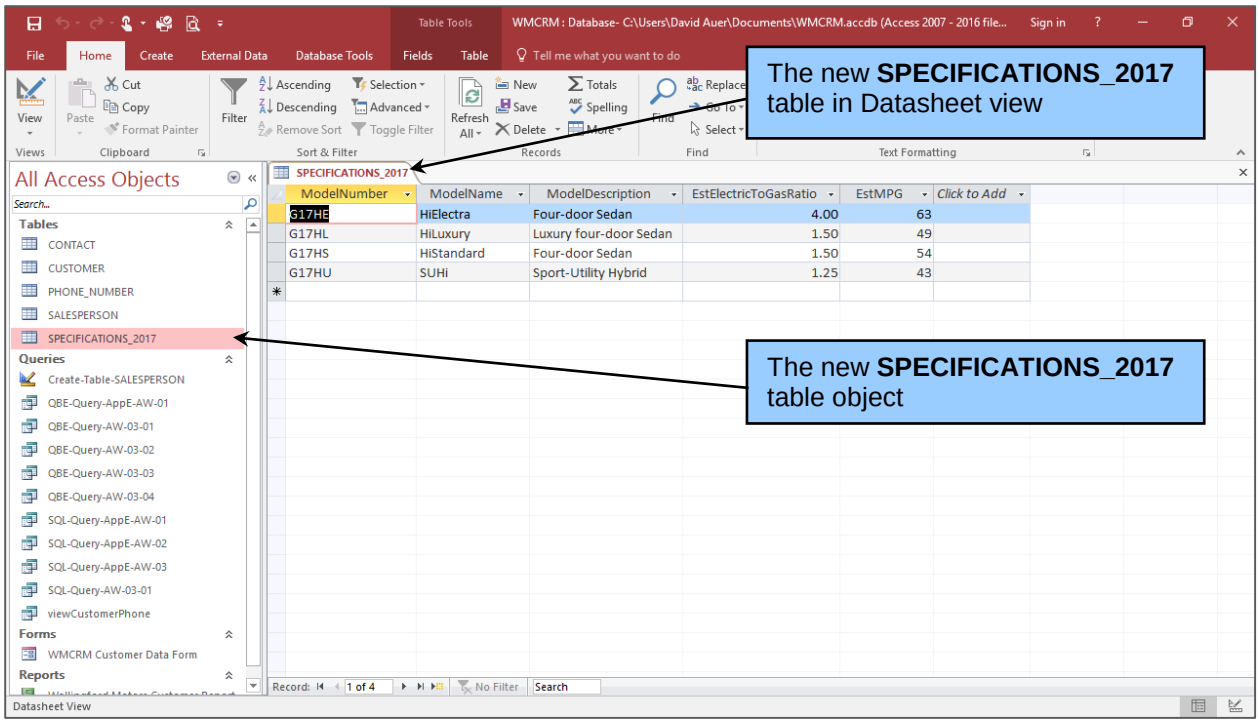


Figure AW-E-20 — The SPECIFICATIONS_2017 Table – Datasheet View

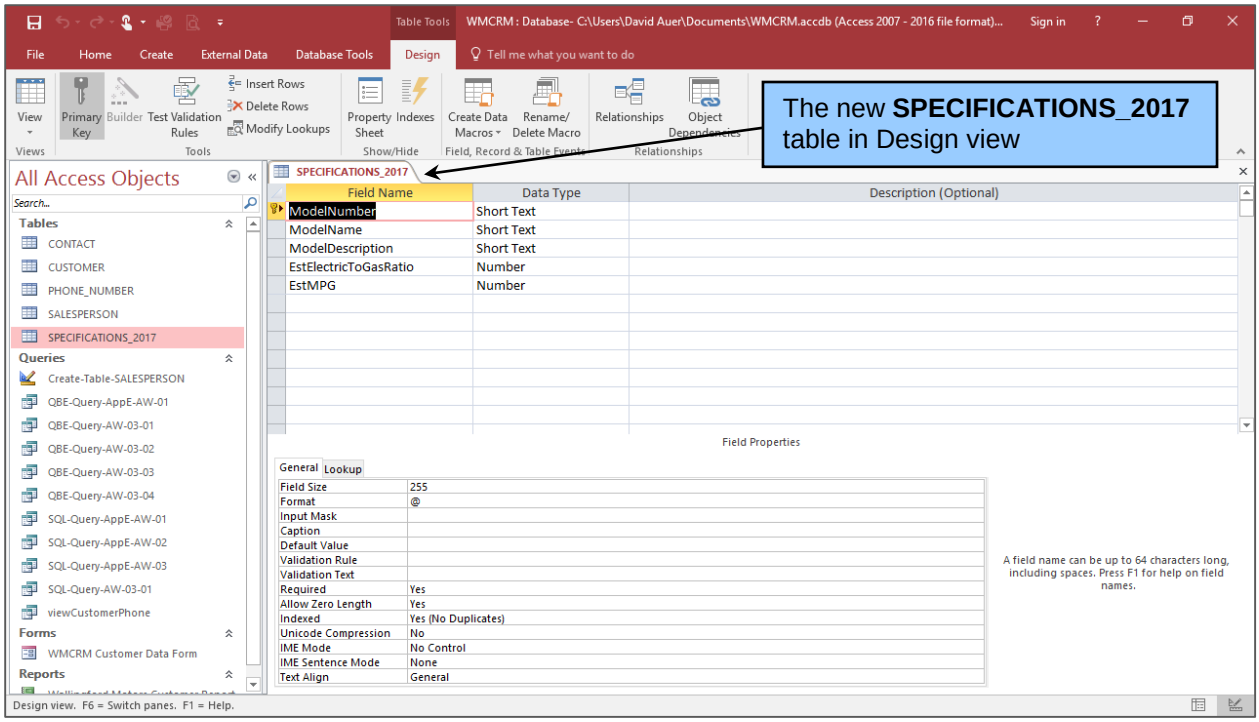


Figure AW-E-21 — The SPECIFICATIONS_2017 Table – Design View

SUMMARY

The SQL ALTER statement is used to add and remove columns and constraints. Column data types and constraints can be changed using the ALTER TABLE ALTER COLUMN statement. Constraints can be added or dropped using the ADD CONSTRAINT and DROP CONSTRAINT with the SQL ALTER TABLE statement. Use of this statement is easier if the developers have provided their own names for all constraints.

The SQL Merge statement combines the SQL INSERT statement and SQL UPDATE statement into one statement that will both insert and update data depending upon whether specific conditions are met.

Correlated subqueries and SQL EXISTS and NOT EXISTS comparison operators are important tools. They can be used to answer advanced queries. They also are useful for determining whether specified data conditions exist. A correlated subquery appears deceptively similar to a regular subquery. The difference is that a regular subquery can be processed from the bottom up. In a regular subquery, results from the lowest query can be determined and then used to evaluate the upper-level queries. In contrast, in a correlated subquery, the processing is nested; that is, a row from an upper-level query statement is compared with rows in a lower-level query. The key feature of a correlated subquery is that the lower-level SELECT statements use columns from upper-level statements.

The SQL EXISTS and NOT EXISTS keywords create specialized forms of correlated subqueries. When these are used, the upper-level query produces results, depending on the existence or nonexistence of rows in lower-level queries. An EXISTS condition is true if *any* row in the subquery meets the specified

conditions; a NOT EXISTS condition is true only if *all* rows in the subquery do not meet the specified condition.

The SQL set operator UNION can be used to combine data from two or more tables provided the tables have an identical table structure. Queries on recursive relationships combine data from one table by treating the table as two virtual tables.

An SQL view is a virtual table that is constructed from other tables and views. An SQL SELECT statement is used as part of an SQL CREATE VIEW statement to define a view. However, in some DBMS systems, view definitions cannot include ORDER BY clauses. Once defined, view names are used in SELECT statements the same way table names are used.

There are several uses for views. Views are used (1) to hide columns or rows, (2) to show the results of computed columns, (3) to hide complicated SQL syntax, and (3) to layer computations and built-in functions.

SQL/PSM is the portion of the SQL standard that provides for storing reusable modules of program code within a database. SQL/PSM specifies that SQL statements will be embedded in user-defined functions, triggers, and stored procedures in a database. It also specifies SQL variables, cursors, control-of-flow statements, and output procedures.

A user-defined function accepts input parameter values from an SQL statement, processes the parameter values, and returns a result value back to the calling statement. User-defined functions may be written to return a single value based on row values (a scalar-valued function), a table of values based on row values (a table-valued function), or a single value based on grouped column values (an aggregate function).

A trigger is a stored program that is executed by the DBMS whenever a specified event occurs on a specified table or view. Possible triggers are BEFORE, INSTEAD OF, and AFTER. Each type of trigger can be declared for insert, update, and delete actions, so nine types of triggers are possible. Oracle supports all nine trigger types, SQL Server supports only INSTEAD OF and AFTER triggers, and MySQL supports the BEFORE and AFTER triggers.

A stored procedure is a program that is stored within the database and compiled when used. Stored procedures can receive input parameters and return results. Unlike triggers, their scope is database-wide. They can be used by any process that has permission to run the stored procedure.

KEY TERMS

ADD {COLUMN} clause	**ADD CONSTRAINT clause**
ALTER {COLUMN} clause	**ALTER CONSTRAINT clause**
bulk SQL INSERT statement	**complement**
data	**DROP COLUMN clause**
DROP CONSTRAINT clause	**information**
intersection	**Open Database Connectivity (ODBC)**
set	**spreadsheet**
SQL ALTER TABLE statement	**SQL ALTER VIEW {ViewName} statement**
SQL CHECK constraint	**SQL COLUMN keyword**
SQL CONSTRAINT keyword	**SQL correlated subquery**
SQL CREATE VIEW statement	**SQL DROP VIEW {ViewName} statement**
SQL EXISTS comparison operator	**SQL LEFT keyword**
SQL MERGE statement	**SQL NOT EXISTS comparison operator**
SQL outer join	**SQL/Persistent Stored Modules (SQL/PSM)**
SQL RIGHT keyword	**SQL set operators**
SQL UNION operator	**SQL view**
stored procedure	**subset**
trigger	**union**
user-defined function	**Venn diagrams**
worksheet	

REVIEW QUESTIONS

E.1 Show an SQL statement to add an integer column C1 to the table T2. Assume that C1 is NULL.

E.2 Extend your answer to review question E.1 to add C1 when C1 is to be NOT NULL.

E.3 Show an SQL statement to drop the column C1 from table T2.

E.4 Assume that tables T1 and T2 have a 1:1 relationship. Assume that T2 has the foreign key. Show the SQL statements necessary to move the foreign key to T1. Make up your own names for the primary and foreign keys.

E.5 Explain how the SQL MERGE statement works, and discuss an example different than the one use in this book of how it could be applied to a database.

E.6 Explain how the SQL outer joins work. What is the difference between a RIGHT JOIN and a LEFT JOIN? Discuss an example different than the one use in this book of how outer joins could be applied to a database.

E.7 Write a subquery, other than one in this chapter, that is not a correlated subquery.

E.8 Explain the following statement: "The processing of correlated subqueries is nested, whereas that of regular subqueries is not."

E.9 Write a correlated subquery other than one in this chapter.

E.10 Explain how the query in your answer to review question E.7 differs from the query in your answer to review question E.9.

E.11 Explain what is wrong with the correlated subquery SQL-Query-AppE-06 on page E-26.

E.12 Explain the meaning of the SQL EXISTS comparison operator.

E.13 Explain how the words *any* and *all* pertain to the SQL EXISTS and NOT EXISTS comparison operators.

E.14 What is a query on a recursive relationship? Discuss an example different than the one use in this book of how such a query could be applied to a database.

E.15 What are *SQL set operators*? What are *Venn diagrams*?

E.16 What does an SQL UNION statement do? Given two tables T1 and T2, what must be true about these two tables if the SQL UNION operator is applied to them? What will be the result of using the UNION operator with T1 and T2?

E.17 What is an SQL view? What purposes do views serve?

E.18 What SQL statements are used to create SQL views?

E.19 What is the limitation on SELECT statements used in SQL views?

E.20 How are views handled in Microsoft Access?

Use the following tables for your answers to Review Questions E.21 through E.37:

PET_OWNER (OwnerID, OwnerLastName, OwnerFirstName, OwnerPhone, OwnerEmail)

PET_3 (PetID, PetName, PetType, PetBreed, PetDOB, PetWeight, *OwnerID*)

These are the same tables that are used in the review questions for Chapter 3, and data for these tables are shown in below in Figures 3-27 and 3-29. For each SQL statement you write, show the results based

on these data. If possible, run the statements you write in the questions that follow in an actual DBMS, as appropriate, to obtain your results.

E.21 Code an SQL statement to create a view named OwnerPhoneView that shows OwnerLastName, OwnerFirstName, and OwnerPhone.

E.22 Code an SQL statement that displays the data in OwnerPhoneView, sorted alphabetically by OwnerLastName.

E.23 Code an SQL statement to create a view named DogBreedView that shows PetID, PetName, PetBreed, and PetDOB for dogs.

E.24 Code an SQL statement that displays the data in DogBreedView, sorted alphabetically by PetName.

E.25 Code an SQL statement to create a view named CatBreedView that shows PetID, PetName, PetBreed, and PetDOB for cats.

E.26 Code an SQL statement that displays the data in CatBreedView, sorted alphabetically by PetName.

E.27 Code an SQL statement to create a view named PetOwnerView that shows PetID, PetName, PetType, OwnerID, OwnerLastName, OwnerFirstName, OwnerPhone, and OwnerEmail.

FIGURE 3-27

PET_OWNER Data

OwnerID	OwnerLastName	OwnerFirstName	OwnerPhone	OwnerEmail
1	Downs	Marsha	555-537-8765	Marsha.Downs@somewhere.com
2	James	Richard	555-537-7654	Richard.James@somewhere.com
3	Frier	Liz	555-537-6543	Liz.Frier@somewhere.com
4	Trent	Miles		Miles.Trent@somewhere.com

FIGURE 3-29

PET_3 Data

PetID	PetName	PetType	PetBreed	PetDOB	PetWeight	OwnerID
1	King	Dog	Std. Poodle	27-Feb-14	25.5	1
2	Teddy	Cat	Cashmere	01-Feb-15	10.5	2
3	Fido	Dog	Std. Poodle	17-Jul-13	28.5	1
4	AJ	Dog	Collie Mix	05-May-14	20.0	3
5	Cedro	Cat	Unknown	06-Jun-12	9.5	2
6	Wooley	Cat	Unknown		9.5	2
7	Buster	Dog	Border Collie	11-Dec-11	25.0	4

E.28 Code an SQL statement that displays the data in PetOwnerView, sorted alphabetically by OwnerLastName and PetName.

E.29 Code an SQL statement to create a view named OwnerPetView that shows OwnerID, OwnerLastName, OwnerFirstName, PetID, PetName, PetType, PetBreed, and PetDOB.

E.30 Code an SQL statement that displays the data in OwnerPetView, sorted alphabetically by OwnerLastName and PetName.

E.31 Code an SQL statement to create a view named PetCountView that shows each type (that is, dog or cat) and the number of each type (that is, how many dogs and how many cats) in the database.

E.32 Code an SQL statement that displays the data in PetCountView, sorted alphabetically by PetType.

E.33 Code an SQL statement to create a view named DogBreedCountView that shows each breed of dog and the number of each breed in the database.

E.34 Code an SQL statement that displays the data in DogBreedCountView, sorted alphabetically by PetBreed.

E.35 Write a user-defined function named FirstNameFirst that concatenates the OwnerLastName and OwnerFirstName into a single value named OwnerName and displays, in order, the OwnerFirstName and OwnerLastName with a single space between them (*Hint*: *Downs* and *Marsha* would be combined to read *Marsha Downs*).

E.36 Code an SQL statement to create a view named PetOwnerFirstNameFirstView that shows PetID, PetName, PetType, OwnerID, OwnerLastName, and OwnerFirstName concatenated using the FirstNameFirst user-defined function and displayed as PetOwnerName, OwnerPhone, and OwnerEmail.

E.37 Code an SQL statement that displays the data in PetOwnerFirstNameFirstView, sorted alphabetically by PetOwnerName and PetName.

E.38 Describe the SQL/PSM component of the SQL standard. What are PL/SQL and T-SQL? What is the MySQL equivalent?

E-39 What is a user-defined function?

E.40 What is a trigger?

E.41 Name nine possible trigger types.

E.42 What are stored procedures? How do they differ from triggers?

E.43 Summarize the key advantages of stored procedures.

EXERCISES

If you haven't created the Art Course database described in Chapter 3, create it now (by completing exercises 3.52 and 3.53). Use the Art Course database to answer exercises E.44 through E.54.

E.44 Code an SQL statement to create a view named CourseView that shows unique course numbers listed together with the corresponding course names and fees.

E.45 Code an SQL statement that displays the data in CourseView, sorted alphabetically by CourseNumber.

E.46 Code an SQL statement to create a view named CourseEnrollmentView that shows CourseNumber, Course, CourseDate, CustomerNumber, CustomerLastName, CustomerFirstName, and Phone.

E.47 Code an SQL statement that displays the data in CourseEnrollmentView for the Advanced Pastels course starting 10/01/17. Sort the data alphabetically by CustomerLastName.

E.48 Code an SQL statement that displays the data in CourseEnrollmentView for the Beginning Oils course starting 10/15/17. Sort the data alphabetically by CustomerLastName.

E.49 Code an SQL statement to create a view named CourseFeeOwedView that shows CourseNumber, Course, CourseDate, CustomerNumber, CustomerLastName, CustomerFirstName, Phone, Fee, AmountPaid, and the calculated column (Fee – AmountPaid), renamed as AmountOwed.

E.50 Code an SQL statement that displays the data in CourseFeeOwedView, sorted alphabetically by CustomerLastName.

E.51 Code an SQL statement that displays the data in CourseFeeOwedView, sorted alphabetically by CustomerLastName for any customer who still owes money for a course fee.

E.52 Write a user-defined function named FirstNameFirst that concatenates the CustomerLastName and CustomerFirstName into a single value named CustomerName and displays, in order, the CustomerFirstName, space, and the CustomerOwnerLastName (*Hint*: *Johnson* and *Ariel* would be combined to read *Ariel Johnson*).

E.53 Code an SQL statement to create a view named CourseEnrollmentFirstNameFirstView that shows CourseNumber, Course, CourseDate, CustomerNumber, CustomerLastName and CustomerFirstName concatenated using the FirstNameFirst user-defined function and displayed as CustomerName, and Phone.

E.54 Code an SQL statement that displays the data in CourseEnrollmentLastNameFirstView, sorted alphabetically by CustomerName and CourseNumber.

For exercises E.55 through E.59 we will use the WP database as described in Chapter 3 and this appendix.

E.55 Insert the data shown in Figure E-36 into the PRODUCTION_ITEM table. table. Note that if you executed the merge statement in SQL-Merge-AppE-01 earlier in this appendix, then the first two rows will already be there, but you will need to change their QuantityOnHand values to 50.

E.56 Create a new table named CATALOG_SKU_2017 based the column characteristics shown in Figure E-2. Include a referential integrity constraint to the PRODUCTION_ITEM table.

E.57 Populate the CATALOG_SKU_2017 table with the data in Figure E-37.

E.58 Create and run an SQL SELECT statement that answers the question: "What products were available for sale (by either catalog or Web site) in 2016 and 2017?" Sort your result by SKU.

E.59 Create and run an SQL SELECT statement that answers the question: "What products were available for sale (by either catalog or Web site) in 2015, 2016, and 2017?" Sort your result by SKU.

SKU	SKU_Description	ProductionStartDate	ProductionEndDate	QuantityOnHand	QuantityInProduction	ApprovalDate
170104001	Alpha IV, Black	11/15/16	NULL	50	200	10/15/16
170104005	Alpha IV, White	11/15/16	NULL	50	200	10/15/16
170203001	Bravo III, Black	12/15/16	NULL	100	250	11/15/16
170203005	Bravo III, White	12/15/16	NULL	100	250	11/15/16

Figure E-36 — Data for the WP PRODUCTION_ITEM Table

CatalogID	SKU	CatalogDescription	CatalogPage	DateOnWebSite
20170001	170104001	Our low price Alpha IV model in black.	9	01/01/17
20170002	170104005	Our low price Alpha IV model in white.	10	01/01/17
20170003	170203001	Our popular Bravo III model in black.	15	01/01/17
20170004	170203005	Our popular Bravo III model in white.	16	01/01/17
20170005	160304001	Our high performance Delta IV model in black.	20	01/01/16
20170006	160304005	Our high performance Delta IV model in white.	22	01/01/16

Figure E-37 — Data for the WP CATALOG_SKU_2017 Table

ACCESS WORKBENCH

Key Terms

Microsoft Access equivalent of an SQL view	**multivalue, multicolumn problem**

Exercises

In the "Access Workbench Exercises" sections for Chapters 1, 2, and 3, you created a database for Wedgewood Pacific (WP) of Seattle, Washington. In this set of exercises, you will use that database, as completed in Chapter 3's section of "The Access Workbench Exercises," to create and use Microsoft Access queries as SQL view equivalents.

AW.E.1 Using Access QBE or SQL, create and run view-equivalent queries to complete the questions that follow. Save each query using the query name format viewViewQueryName, where ViewQueryName is the name specified in the question.

A. Create an Access view–equivalent query named Computer that shows Make, Model, SerialNumber, ProcessorType, ProcessorSpeed, MainMemory, and DiskSize.

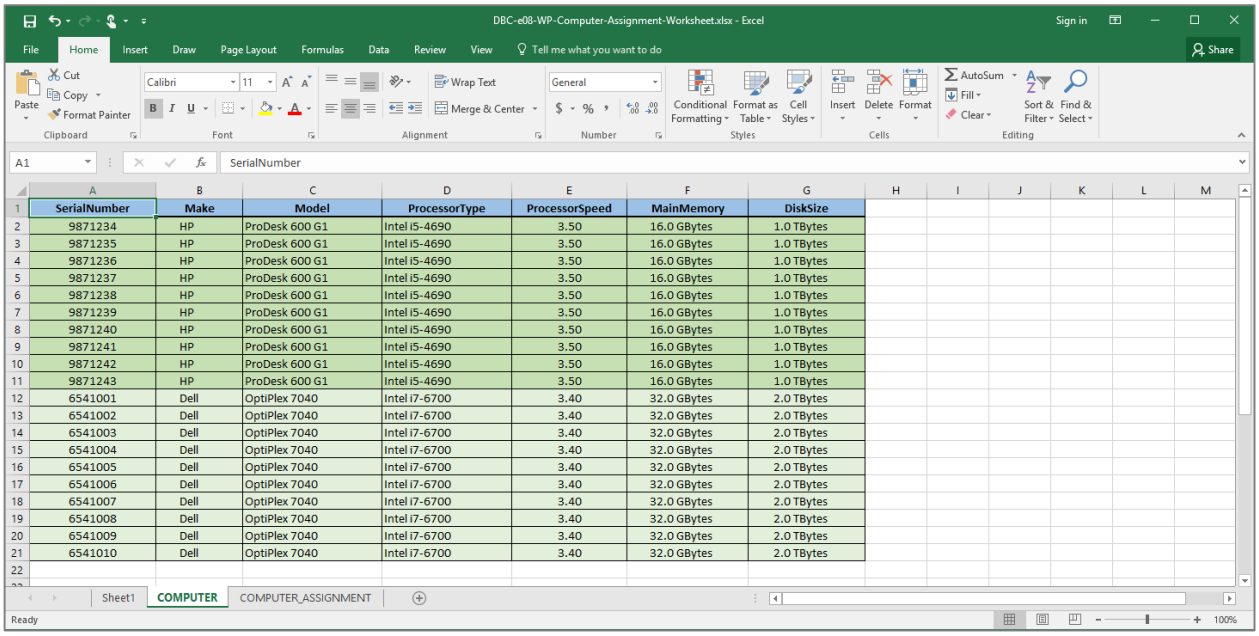
B. Create an Access view–equivalent query named EmployeeComputer that uses viewComputer for part A to show EMPLOYEE.EmployeeNumber, LastName, FirstName, and the data about the computer assigned to that employee, including Make, Model, SerialNumber, ProcessorType, ProcessorSpeed, MainMemory, and DiskSize.

AW.E.2 Use Access QBE to create and run the queries that follow. Save each query using the query name format QBE-Query-AppE-AW-2-##, where ## is replaced by the letter designator of the question. For example, the first query will be saved as QBE-Query-AppE-AW-2-A.

A. Create an Access QBE query to display the data in viewComputer, sorted alphabetically by Make and Model and then numerically by SerialNumber.

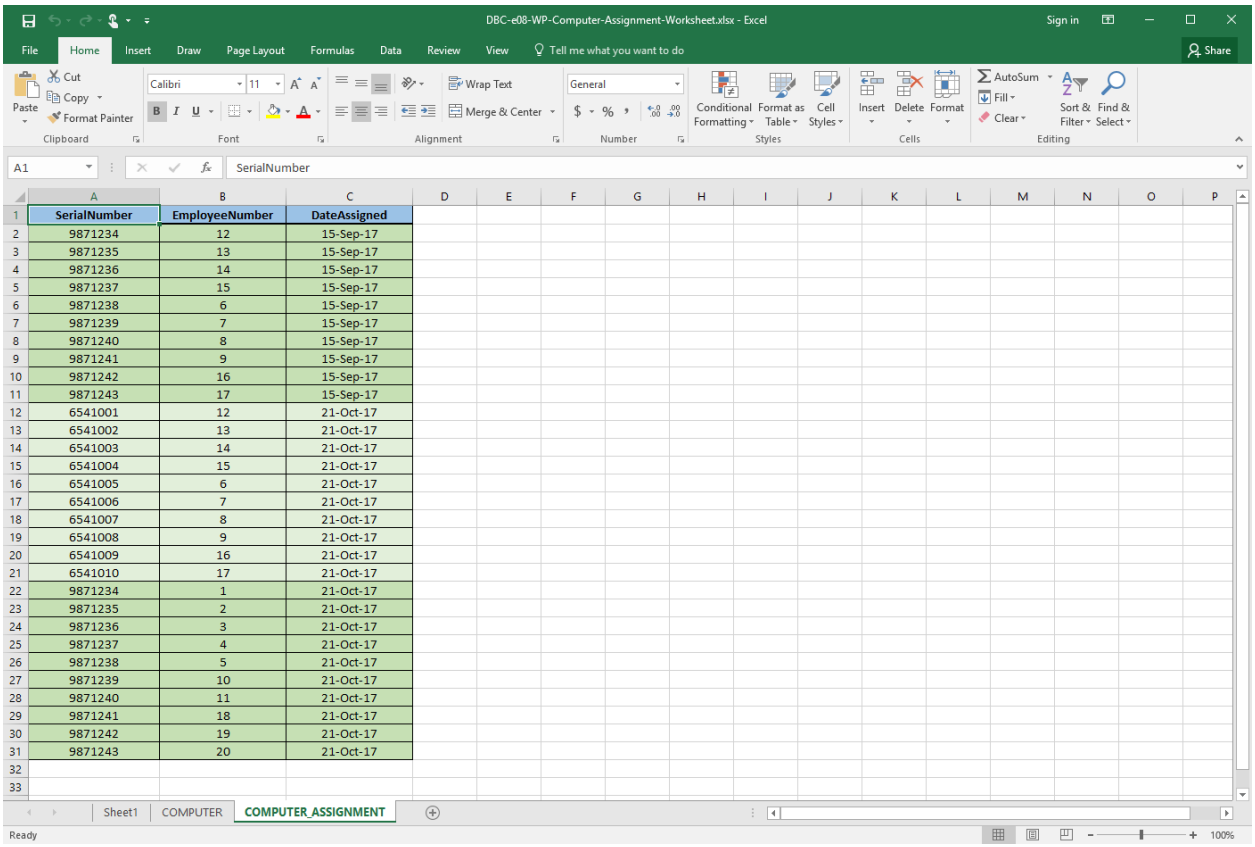
B. Create an Access QBE query to display the data in viewEmployeeComputer. Sort the results alphabetically by LastName, FirstName, Make, and Model and then numerically by SerialNumber.

AW.E.3 Figures E-38 and E-39 show to worksheets in a Microsoft Excel 2016 workbook. The COMPUTER worksheet shows data about computers owned at WP, while the COMPUTER_ASSIGNMENT worksheet shows, in the correct format, the data about which computer has been or is currently assigned to which WP employee. We want to import this data into two tables in the Microsoft Access 2016 WP.accdb database.



SerialNumber	Make	Model	ProcessorType	ProcessorSpeed	MainMemory	DiskSize
9871234	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871235	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871236	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871237	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871238	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871239	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871240	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871241	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871242	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
9871243	HP	ProDesk 600 G1	Intel i5-4690	3.50	16.0 GBytes	1.0 TBytes
6541001	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541002	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541003	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541004	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541005	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541006	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541007	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541008	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541009	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes
6541010	Dell	OptiPlex 7040	Intel i7-6700	3.40	32.0 GBytes	2.0 TBytes

Figure E-38 — The WP COMPUTER Worksheet



SerialNumber	EmployeeNumber	DateAssigned
9871234	12	15-Sep-17
9871235	13	15-Sep-17
9871236	14	15-Sep-17
9871237	15	15-Sep-17
9871238	6	15-Sep-17
9871239	7	15-Sep-17
9871240	8	15-Sep-17
9871241	9	15-Sep-17
9871242	16	15-Sep-17
9871243	17	15-Sep-17
6541001	12	21-Oct-17
6541002	13	21-Oct-17
6541003	14	21-Oct-17
6541004	15	21-Oct-17
6541005	6	21-Oct-17
6541006	7	21-Oct-17
6541007	8	21-Oct-17
6541008	9	21-Oct-17
6541009	16	21-Oct-17
6541010	17	21-Oct-17
9871234	1	21-Oct-17
9871235	2	21-Oct-17
9871236	3	21-Oct-17
9871237	4	21-Oct-17
9871238	5	21-Oct-17
9871239	10	21-Oct-17
9871240	11	21-Oct-17
9871241	18	21-Oct-17
9871242	19	21-Oct-17
9871243	20	21-Oct-17

Figure E-39 — The WP COMPUTER_ASSIGNMENT Worksheet

Actually, we have already done a version of this exercise in the Chapter 3 Access Workbench Exercises. So, in case you have completed those exercises, we will import the data into two new tables use alternate table names: COMPUTER_2 and COMPUTER_ASSIGNMENT_2.

A. In Microsoft Access 2016, create a workbook named **DBC-e08-WP-Computer-Assignment-Worksheet.xlsx**. In this workbook, create the **COMPUTER** and **COMPUTER_ASSIGNMENT** worksheets shown in Figures E-38 and E-39.

B. Using the table names COMPUTER_2 and COMPUTER_ASSIGNMENT_2, import the data from the DBC-e08-WP-Computer-Assignment-Worksheet.xlsx file into the WP database.

C. Figure E-40 shows the column characteristics for the COMPUTER_2 table. Modify the COMPUTER_2 table structure as needed to match the table column characteristics shown in Figures E-40.

D. Figure E-41 shows the column characteristics for the COMPUTER_ASSIGNMENT_2 table. Modify the COMPUTER_ASSIGNMENT_2 table structure as needed to match the table column characteristics shown in Figures E-41. *Note:* These column characteristics intentionally vary from those shown in Figure 3-34—can you figure out why?

E. Create the needed relationships between COMPUTER_2, COMPUTER_ASSIGNMENT_2 and EMPLOYEE.

F. Complete exercises AW.E.1 and AW.E.2 using COMPUTER_2 and COMPUTER_ASSIGNMENT_2and with appropriate view names. For the questions in AW.E.2, save each query using the query name format QBE-Query-AppE-AW-3-##, where ## is replaced by the letter designator of the question. For example, the first query will be saved as QBE-Query-AppE-AW-3-A.

Column Name	Type	Key	Required	Remarks
SerialNumber	Number	Primary Key	Yes	Long Integer
Make	Short Text (12)	No	Yes	Must be "Dell" or "Gateway" or "HP" or "Other"
Model	Short Text (24)	No	Yes	
ProcessorType	Short Text (24)	No	No	
ProcessorSpeed	Number	No	Yes	Double [3,2], Between 1.0 and 4.0
MainMemory	Short Text (15)	No	Yes	
DiskSize	Short Text (15)	No	Yes	

Figure E-40 — Database Column Characteristics for the WP COMPUTER_2 Table

Column Name	Type	Key	Required	Remarks
SerialNumber	Number	Primary Key, Foreign Key	Yes	Long Integer
EmployeeNumber	Number	Primary Key, Foreign Key	Yes	Long Integer
DateAssigned	Date/Time	Primary Key	Yes	Medium Date

Figure E-41 — Database Column Characteristics for the WP COMPUTER_ASSIGNMENT_2 Table

HEATHER SWEENEY DESIGNS CASE QUESTIONS

These questions are based on Chapter 3's Heather Sweeney Designs case questions. Base your answers to the questions that follow on the HSD database, as described there. If possible, run your SQL statements in an actual DBMS to validate your work.

A. Add a column to the CUSTOMER table named ReferredBy, which will contain data on which customer referred the new customer to the store. Customers may refer only one new customer. The column characteristics for the ReferredBy column are shown in Figure E-42. Populate the column with the data shown in Figure E-43.

B. Add a column to the SEMINAR_CUSTOMER table named Attended, which will contain data showing whether a customer who registered for a seminar actually attended the seminar. Column characteristics for the Attended column are shown in Figure E-44. Populate the column with the data shown in Figure E-45.

C. How did your steps to add the Attended column differ from your steps to add the ReferredBy column? Why was (were) the additional step(s) necessary?

D. Add an SQL CHECK constraint to the SEMINAR_CUSTOMER table to ensure that only the values of *Attended* or *Did not attend* are allowed as data in the Attended column.

E. The HSD DBA has realized that the SEMINAR_CUSTOMER table Attended column should be NULL instead of NOT NULL as it was created, because we do not know whether or not a customer will attend a seminar at the time the customer registers for the seminar. Alter the Attended column to NULL instead of NOT NULL.

F. Write an SQL SELECT statement to create a query on the recursive relationship in the CUSTOMER table that shows each customer's FirstName (as CustomerFirstName) and LastName (as CustomerEmployeeLastName) followed by the name of the customer who referred him or her to Heather Sweeney Designs using the referring customer's FirstName (as ReferrerFirstName) and LastName (as ReferrerLastName). Do *not* include customers who were not referred by another customer.

G. Write an SQL SELECT statement to create a query on the recursive relationship in the CUSTOMER table that shows each customer's FirstName (as CustomerFirstName) and LastName (as CustomerEmployeeLastName) followed by the name of the customer who referred him or her to Heather Sweeney Designs using the referring customer's FirstName (as ReferrerFirstName) and LastName (as ReferrerLastName). *Do* include customers who were not referred by another customer.

Column Name	Type	Key	Required	Remarks
ReferredBy	Integer	Foreign Key	No	UNIQUE REF: CustomerID in CUSTOMER

Figure E-42 — Column Characteristics for the New Column in the HSD CUSTOMER Table

CustomerID	LastName	FirstName	...	ReferredBy
1	Jacobs	Nancy	...	NULL
2	Jacobs	Chantel	...	1
3	Able	Ralph	...	NULL
4	Baker	Susan	...	3
5	Eagleton	Sam	...	4
6	Foxtrot	Kathy	...	NULL
7	George	Sally	...	7
8	Hullett	Shawn	...	6
9	Pearson	Bobbi	...	NULL
10	Ranger	Terry	...	9
11	Tyler	Jenny	...	NULL
12	Wayne	Joan	...	2

Figure E-43 — Data for the New Column in the HSD CUSTOMER Table

Column Name	Type	Key	Required	Remarks
Attended	Varchar (20)	No	Yes	CHECK Values: "Attended", "Did not attend"

Figure E-44 — Column Characteristics for the New Column in the HSD SEMINAR_CUSTOMER Table

---

H Write a user-defined function named FirstNameFirst that concatenates the customer's LastName and FirstName into a single value named CustomerName and displays, in order, the FirstName, a space, and the LastName (*Hint*: *Jacobs* and *Nancy* would be combined to read *Nancy Jacobs*).

SeminarID	CustomerID	...	Attended
1	1	...	Attended
1	2	...	Did not attend
1	3	...	Attended
1	4	...	Attended
1	5	...	Attended
2	6	...	Attended
2	7	...	Attended
2	8	...	Attended
3	9	...	Attended
3	10	...	Attended
4	6	...	Attended
4	7	...	Did not attend
4	11	...	Attended
4	12	...	Attended

Figure E-45 — Data for the New Column in the HSD SEMINAR_CUSTOMER Table

I. Create the following SQL views:

1. Create an SQL view named CustomerSeminarView that shows CUSTOMER.CustomerID, LastName, FirstName, EmailAddress, City, State, ZIP, SeminarDate, Location, and SeminarTitle.
2. Create an SQL view named CustomerFirstNameFirstSeminarView that shows CUSTOMER.CustomerID, then LastName and FirstName concatenated using the FirstNameFirst user-defined function and displayed as CustomerName, EmailAddress, City, State, ZIP, SeminarDate, Location, and SeminarTitle.
3. Create an SQL view named CustomerProductView that shows CustomerID, LastName, FirstName, EmailAddress, INVOICE.InvoiceNumber, InvoiceDate, PRODUCT.ProductNumber, and ProductDescription.
4. Create an SQL view named CustomerFirstNameFirstProductView that shows CustomerID, then LastName and FirstName concatenated using the FirstNameFirst user-defined function and displayed as CustomerName, EmailAddress, INVOICE.InvoiceNumber, InvoiceDate, PRODUCT.ProductNumber, and ProductDescription.

J. Create and run the following SQL queries:

1. Create an SQL statement to run CustomerSeminarView, with the results sorted alphabetically by State, City, and ZIP (in that order) in descending order.
2. Create an SQL statement to run CustomerFirstNameFirstSeminarView, with the results sorted alphabetically by State, City, and ZIP (in that order) in descending order.
3. Create an SQL statement to run CustomerSeminarView, with the results sorted alphabetically by Location, SeminarDate, and SeminarTitle (in that order) in ascending order.
4. Create an SQL statement to run CustomerFirstNameFirstSeminarView, with the results sorted alphabetically by Location, SeminarDate, and SeminarTitle (in that order) in ascending order.
5. Create an SQL statement to run CustomerProductView, with the results sorted alphabetically by LastName, FirstName, InvoiceNumber, and ProductNumber in ascending order.
6. Create an SQL statement to run CustomerFirstNameFirstProductView, with the results sorted alphabetically by CustomerName, InvoiceNumber, and ProductNumber in ascending order.

K. Heather Sweeney Designs staff keep employees in a Microsoft Excel worksheet, as shown in Figure E-46. Heather now wants to import this data into one or more database tables.

- The steps for importing data into Microsoft SQL Server 2016 from Microsoft Excel 2016 are discussed in Appendix A, “Getting Started with Microsoft SQL Server 2016.”
- The steps for importing data into Oracle Database XE from Microsoft Excel 2016 are discussed in Appendix B, “Getting Started with Oracle Database XE.”
- The steps for importing data into MySQL 5.7 from Microsoft Excel 2016 are discussed in Appendix C, “Getting Started with MySQL 5.7 Community Server.”

1. Duplicate Figure E-46 in a worksheet (or spreadsheet) in an appropriate tool (such as Microsoft Excel or Apache OpenOffice Calc).
2. Import the data into a *temporary* table in the HSD database.
3. Create a new table in the HSD database named EMPLOYEE. The column characteristics for the HSD EMPLOYEE table are shown in Figure E-47.
4. Populate the EMPLOYEE table as much as you can with the imported data in the temporary table. Note that you may not be able to populate all the columns in the EMPLOYEE table based on the available data. *Hint:* Consider using a bulk SQL INSERT statement.

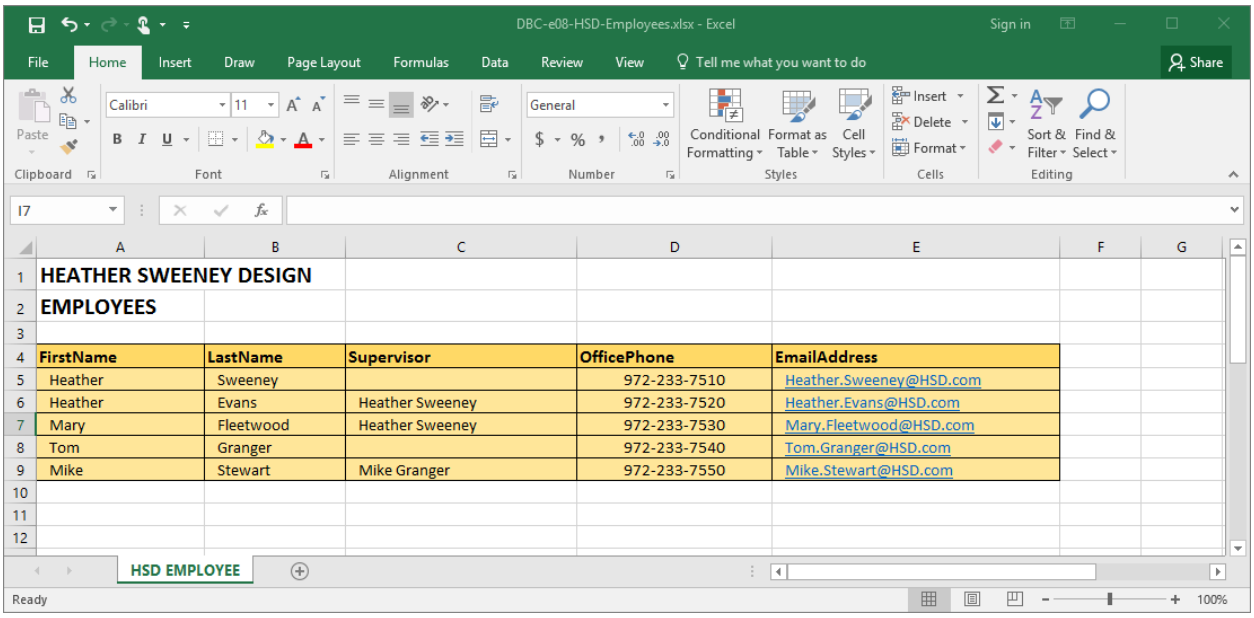


Figure E-46 — The Heather Sweeney Designs Employee Worksheet

Column Name	Type	Key	Required	Remarks
EmplyeeNumber	Integer	Primary Key	Yes	Surrogate Key: Start at 1, Increment by 1
FirstName	Char (25)	No	Yes	
LastName	Char (25)	No	Yes	
Supervisor	Integer	Foreign Key	No	REF: EmployeeID in EMPLOYEE
OfficePhone	Char (12)	No	No	
EmailAddress	Char (100)	No	Yes	

Figure E-47 — Column Characteristics for the New Columns in the HSD EMPLOYEE Table

5. If there are any columns that still need data, write and run the SQL statements necessary to finish populating the EMPLOYEE table.

6. Write an SQL SELECT statement to create a query on the recursive relationship in the EMPLOYEE table that shows each employee's FirstName (as EmployeeFirstName) and LastName (as EmployeeLastName) followed by that employee's supervisor's FirstName (as SupervisorFirstName) and LastName (as SupervisorLastName). *Do* include employees who do not have a supervisor.

GARDEN GLORY PROJECT QUESTIONS

These questions are based on Chapter 3's Garden Glory project questions. Base your answers to the questions that follow on the Garden Glory database, as described there. Run your SQL statements in an actual DBMS to validate your work.

A. Add a column to the EMPLOYEE table named Supervisor, which will contain data showing who supervises an employee. One employee may supervise more than one other employee. Column characteristics for the GG Supervisor column are shown in Figure E-48. Populate the column with the data shown in Figure E-49.

B. Garden Glory requires that each employee complete an apprenticeship program. Add a column to the EMPLOYEE table named Apprenticeship, which will contain data showing the apprenticeship status for each employee. Column characteristics for the GG Apprenticeship column are shown in Figure E-48. Populate the column with the data shown in Figure E-49.

C. How did your steps to add the Apprenticeship column differ from your steps to add the Supervisor column? Why was (were) the additional step(s) necessary?

D. Add an SQL CHECK constraint to the EMPLOYEE table to ensure that only the values of *Completed*, *In process*, or *Not started* are allowed as data in the Apprenticeship column.

Column Name	Type	Key	Required	Remarks
Supervisor	Integer	Foreign Key	No	REF: EmployeeID in EMPLOYEE
Apprenticeship	Varchar (20)	No	Yes	CHECK Values: "Completed", "In process", "Not started"

Figure E-48 — Column Characteristics for the New Columns in the GG EMPLOYEE Table

EmployeeID	LastName	FirstName	...	Supervisor	Apprenticeship
1	Smith	Sam	...	NULL	Completed
2	Evanston	John	...	5	In process
3	Murray	Dale	...	1	Not started
4	Murphy	Jerry	...	1	Completed
5	Fontaine	Joan	...	NULL	In process

Figure E-49 — Data for the New Columns in the GG EMPLOYEE Table

E. Write an SQL SELECT statement to create a query on the recursive relationship in the EMPLOYEE table that shows each employee's FirstName (as EmployeeFirstName) and LastName (as EmployeeLastName) followed by that employee's supervisor's FirstName (as SupervisorFirstName) and LastName (as SupervisorLastName). Do *not* include employees who do not have a supervisor.

F. Write an SQL SELECT statement to create a query on the recursive relationship in the EMPLOYEE table that shows each employee's FirstName (as EmployeeFirstName) and LastName (as EmployeeLastName) followed by that employee's supervisor's FirstName (as SupervisorFirstName) and LastName (as SupervisorLastName). *Do* include employees who do not have a supervisor.

G. Write an SQL SELECT statement to create a correlated subquery to determine if there are any employees who have the same combination of LastName and FirstName. *Hint*: If *no* employees meet this condition, the correct query result will be an **empty set**.

H. Write a user-defined function named FirstNameFirst that concatenates the employee's LastName and FirstName into a single value named FullName and displays, in order, the FirstName, a space, and the LastName (*Hint*: *Smith* and *Steve* would be combined to read *Steve Smith*).

I. Create the following SQL views:

1. Create an SQL view named OwnerPropertyView that shows OWNER.OwnerID, OwnerName, PropertyType, PropertyID, PropertyName, Street, City, State, and Zip.
2. Create an SQL view named PropertyServiceView that shows OWNED_PROPERTY.PropertyID, PropertyName, Street, City, State, Zip, ServiceDate, FirstName, LastName, and HoursWorked.
3. Create an SQL view named PropertyServiceFirstNameFirstView that shows OWNED_PROPERTY.PropertyID, PropertyName, Street, City, State, Zip, ServiceDate, then LastName and FirstName concatenated using the FirstNameFirst user-defined function and displayed as EmployeeName, and HoursWorked.

J. Create (and run) the following SQL queries:

1. Create an SQL statement to run OwnerPropertyView, with the results sorted alphabetically by OwnerName.
2. Create an SQL statement to run PropertyServiceView, with the results sorted alphabetically by Zip, State, and City.
3. Create an SQL statement to run PropertyServiceFirstNameFirstView, with the results sorted alphabetically by Zip, State, and City.

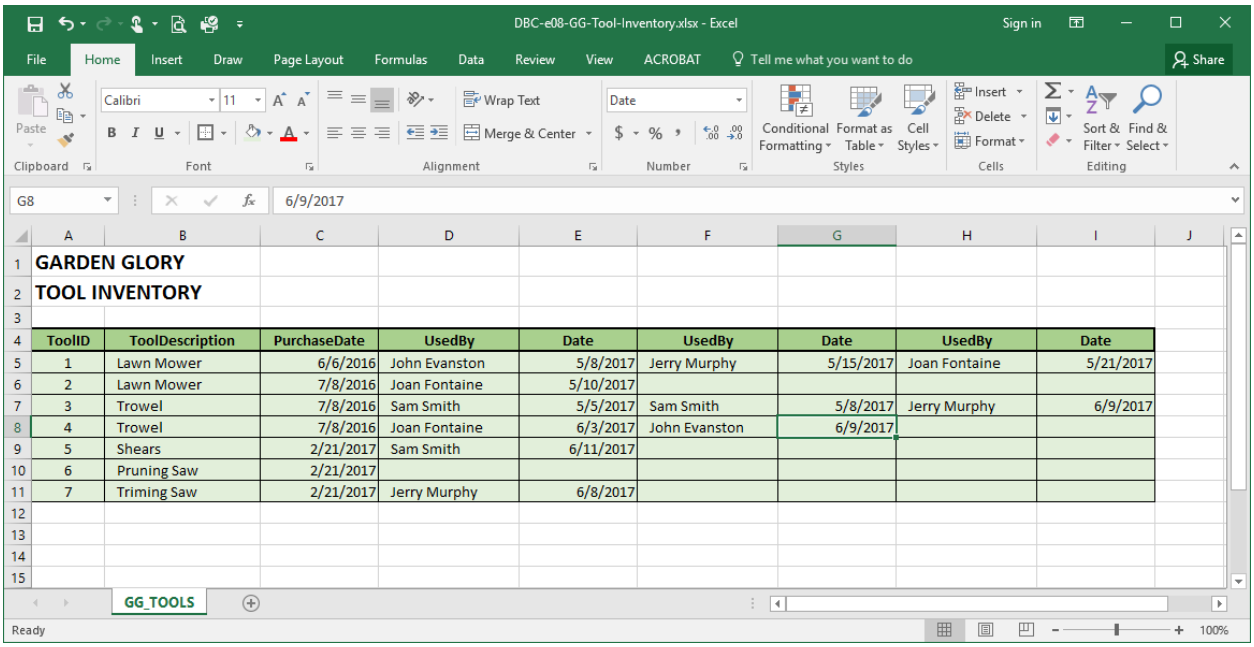


Figure E-50 — The Garden Glory Tool Inventory Worksheet

K. Garden Glory staff keep a record of tool inventory and who uses those tools in a Microsoft Excel worksheet, as shown in Figure E-50. Garden Glory now wants to import this data into one or more database tables.

- The steps for importing data into Microsoft SQL Server 2016 from Microsoft Excel 2016 are discussed in Appendix A, “Getting Started with Microsoft SQL Server 2016.”
- The steps for importing data into Oracle Database XE from Microsoft Excel 2016 are discussed in Appendix B, “Getting Started with Oracle Database XE.”
- The steps for importing data into MySQL 5.7 from Microsoft Excel 2016 are discussed in Appendix C, “Getting Started with MySQL 5.7 Community Server.”

1. Duplicate Figure E-50 in a worksheet (or spreadsheet) in an appropriate tool (such as Microsoft Excel or Apache OpenOffice Calc).
2. Import the data into one or more new tables in the GG database. You must determine all tables characteristics needed (primary key, foreign keys, data types, etc.).
3. Link this (these) new table(s) as appropriate to one or more existing tables in the GG database. Explain why you chose to make the connection(s) you made.

JAMES RIVER JEWELRY PROJECT QUESTIONS

The James River Jewelry project questions are available in online Appendix D, which can be downloaded from the textbook's Web site: www.pearsonhighered.com/kroenke.

THE QUEEN ANNE CURIOSITY SHOP PROJECT QUESTIONS

These questions are based on Chapter 3's Queen Anne Curiosity Shop project questions. Base your answers to the questions that follow on the Queen Anne Curiosity Shop project database, as described there. Run your SQL statements in an actual DBMS to validate your work.

A. Add a column to the CUSTOMER table named ReferredBy, which will contain data on which customer referred the new customer to the store. Customers may only refer one new customer. The column characteristics for the ReferredBy column are shown in Figure E-51. Populate the column with the data shown in Figure E-52.

B. Add a column to the EMPLOYEE table named Supervisor, which will contain data showing who supervises an employee. One employee may supervise more than one other employee. Column characteristics for the QACS Supervisor column are shown in Figure E-53. Populate the column with the data shown in Figure E-54.

C. The Queen Anne Curiosity Shop requires that each employee complete a training program. Add a column to the EMPLOYEE table named Training, which will contain data showing the training status for each employee. Column characteristics for the QACS Training column are shown in Figure E-53. Populate the column with the data shown in Figure E-54.

D. How did your steps to add the Training column differ from your steps to add the Supervisor column? Why was (were) the additional step(s) necessary?

E. Add an SQL CHECK constraint to the EMPLOYEE table to ensure that only the values of *Completed*, *In process*, or *Not started* are allowed as data in the Training column

F. Write an SQL SELECT statement to create a query on the recursive relationship in the CUSTOMER table that shows each customer's FirstName (as CustomerFirstName) and LastName (as CustomerLastName) followed by the name of the customer who referred him or her to QACS using the referring customer's FirstName (as ReferrerFirstName) and LastName (as ReferrerLastName). Do *not* include customers who were not referred by another customer.

G. Write an SQL SELECT statement to create a query on the recursive relationship in the CUSTOMER table that shows each customer's FirstName (as CustomerFirstName) and LastName (as CustomerLastName) followed by the name of the customer who referred him or her to QACS using the referring customer's FirstName (as RefereerFirstName) and LastName (as RefereerLastName). *Do* include customers who were not referred by another customer.

Column Name	Type	Key	Required	Remarks
ReferredBy	Integer	Foreign Key	No	UNIQUE REF: CustomerID in CUSTOMER

Figure E-51 — Column Characteristics for the New Column in the QACS CUSTOMER Table

CustomerID	LastName	FirstName	...	ReferredBy
1	Shire	Robert	...	NULL
2	Goodyear	Katherine	...	1
3	Bancroft	Chris	...	NULL
4	Griffith	John	...	2
5	Tierney	Doris	...	3
6	Anderson	Donna	...	NULL
7	Svane	Jack	...	4
8	Walsh	Denesha	...	5
9	Enquist	Craig	...	6
10	Anderson	Rose	...	7

Figure E-52 — Data for the New Column in the QACS CUSTOMER Table

Column Name	Type	Key	Required	Remarks
Supervisor	Integer	Foreign Key	No	REF: EmployeeID in EMPLOYEE
Training	Varchar (20)	No	Yes	CHECK Values: "Completed", "In process", "Not started"

Figure E-53 — Column Characteristics for the New Columns in the QACS EMPLOYEE Table

EmployeeID	LastName	FirstName	...	Supervisor	Training
1	Stuart	Anne	...	NULL	Completed
2	Stuart	George	...	1	Completed
3	Stuart	Mary	...	1	Completed
4	Orange	William	...	3	In process
5	Griffith	John	...	3	In process

Figure E-54 — Data for the New Columns in the QACS EMPLOYEE Table

H. Write an SQL SELECT statement to create a query on the recursive relationship in the EMPLOYEE table that shows each employee's FirstName (as EmployeeFirstName) and LastName (as EmployeeLastName) followed by that employee's supervisor's FirstName (as SupervisorFirstName) and LastName (as SupervisorLastName). Do *not* include employees who do not have a supervisor.

I. Write an SQL SELECT statement to create a query on the recursive relationship in theEMPLOYEE table that shows each employee's FirstName (as EmployeeFirstName) and LastName (as EmployeeLastName) followed by that employee's supervisor's FirstName (as SupervisorFirstName) and LastName (as SupervisorLastName). *Do* include employees who do not have a supervisor.

J. Suppose that the Queen Anne Curiosity Shop owners are considering changing the primary key of CUSTOMER to (FirstName, LastName). Write a correlated subquery to display any data that indicate that this change is not justifiable. *Hint*: If *no* employees meet this condition, the correct query result will be an **empty set**.

K. Write a user-defined function named FirstNameFirst that concatenates the employee's LastName and FirstName into a single value named FullName and displays, in order, the FirstName, a space, and the LastName (*Hint*: *Smith* and *Steve* would be combined to read *Steve Smith*).

L. Create the following SQL view statements:

1. Create an SQL view named BasicCustomerView that shows each customer's CustomerID, LastName, FirstName, Phone, and EmailAddress.
2. Create an SQL view named BasicCustomerFirstNameFirstView that shows each customer's CustomerID, then LastName and FirstName concatenated using the LastNameFirst user-defined function and displayed as CustomerName, Phone, and EmailAddress.
3. Create an SQL view named SaleItemItemView that shows SaleID, SaleItemID, SALE_ITEM.ItemID, SaleDate, ItemDescription, ItemCost, ITEM.ItemPrice as ListItemPrice, and SALE_ITEM.ItemPrice as ActualItemPrice.

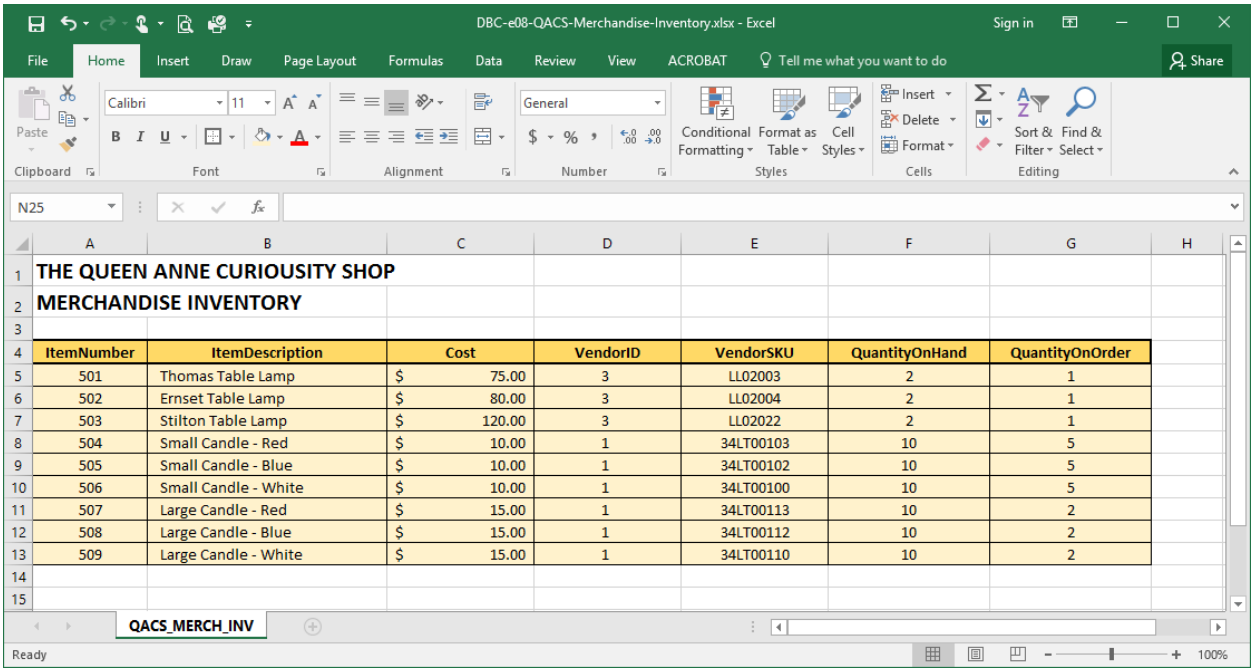
M. Create (and run) the following SQL queries:

1. Create an SQL statement to run BasicCustomerView, with the results sorted alphabetically by LastName and FirstName.
2. Create an SQL statement to run BasicCustomerFirstNameFirstView, with the results sorted alphabetically by CustomerName.
3. Create an SQL statement to run SaleItemItemView, with the results sorted by SaleID and SaleItemID.
4. Create an SQL query that uses SaleItemItemView to calculate and display the sum of SALE_ITEM.ItemPrice (which is relabeled as ActualItemPrice) as TotalPretaxRetailSales.

N. The Queen Anne Curiosity Shop owners and staff have decided to sell standardized items that can be stocked and reordered as necessary. So far, they have kept their records for these items in a Microsoft Excel worksheet, as shown in Figure E-51. They have decided to integrate this data into the QACS database, and they want to import this data into one or more database tables.

- The steps for importing data into Microsoft SQL Server 2016 from Microsoft Excel 2016 are discussed in Appendix A, "Getting Started with Microsoft SQL Server 2016."
- The steps for importing data into Oracle Database XE from Microsoft Excel 2016 are discussed in Appendix B, "Getting Started with Oracle Database XE."
- The steps for importing data into MySQL 5.7 from Microsoft Excel 2016 are discussed in Appendix C, "Getting Started with MySQL 5.7 Community Server."

1. Duplicate Figure E-51 in a worksheet (or spreadsheet) in an appropriate tool (such as Microsoft Excel or Apache OpenOffice Calc).
2. Import the data into one or more new tables in the QACS database. You must determine all table characteristics needed (primary key, foreign keys, data types, etc.).
3. Link this (these) new table(s) as appropriate to one or more existing tables in the QACS database. Explain why you chose to make the connection(s) that you made.



ItemNumber	ItemDescription	Cost	VendorID	VendorSKU	QuantityOnHand	QuantityOnOrder
501	Thomas Table Lamp	$ 75.00	3	LL02003	2	1
502	Ernset Table Lamp	$ 80.00	3	LL02004	2	1
503	Stilton Table Lamp	$ 120.00	3	LL02022	2	1
504	Small Candle - Red	$ 10.00	1	34LT00103	10	5
505	Small Candle - Blue	$ 10.00	1	34LT00102	10	5
506	Small Candle - White	$ 10.00	1	34LT00100	10	5
507	Large Candle - Red	$ 15.00	1	34LT00113	10	2
508	Large Candle - Blue	$ 15.00	1	34LT00112	10	2
509	Large Candle - White	$ 15.00	1	34LT00110	10	2

Figure E-55 — The Queen Anne Curiosity Shop Standard Merchandise Inventory Worksheet