5.53 What entity type should come to mind when you see the words "For use by" in a form?

5.54 Give examples of 1:1, 1:N, and N:M recursive relationships (other than those presented in this chapter).

5.55 Explain why the data modeling process must be iterative. Use the Highline University example.

Project Questions

5.56 This question is for Visio users. Convert the data models in Figures 5-16, 5-20, 5-22, 5-23, 5-30, 5-33, 5-37, and 5-52 into Visio format. Use the Visio arrow notation.

5.57 This question is for Visio users. Convert the data models in Figures 5-16, 5-20, 5-22, 5-23, 5-30, 5-33, 5-37, and 5-52 into Visio format. Use the Visio version of the IE Crow's Foot model.

Answer the following questions using IE Crow's Foot notation.

5.58 Examine the subscription form shown in Figure 5-53.

Using the structure of this form, do the following:

A. Create a model with one entity. Specify the identifier and attributes.

B. Create a model with two entities, one for customer and a second for subscription. Specify identifiers, attributes, relationship name, type, and cardinalities.

C. Under what conditions do you prefer the model in A to that in B?

D. Under what conditions do you prefer the model in B to that in A?

5.59 Consider the traffic citation shown in Figure 5-54.

The rounded corners on this form provide graphical hints about the boundaries of the entities represented.

A. Create a data model with five entities. Use the data items on the form to specify identifiers and attributes for those entities.

B. Specify relationships among the entities. Name the relationship and give its type and cardinalities. Indicate which cardinalities can be inferred from data on the form and which need to be checked out with systems users.

Figure 5-53
Subscription Form

Fine Woodworking — **To subscribe**

☐ 1 year (6 issues) for just $18—20% off the newsstand price.
(Outside the U.S. $21/year—U.S. funds, please)
☐ 2 years (12 issues) for just $34—save 24%
(Outside the U.S. $40/2 years—U.S. funds, please)

Name _____

Address _____

City _____ State _____ Zip _____

☐ My payment is enclosed. ☐ Please bill me.

Please start my subscription with ☐ *current issue* ☐ *next issue*.

Amazon Ex. 1008.0211

WASHINGTON STATE PATROL CORRECTION NOTICE

NAME LAST Kroenke FIRST David M
ADDRESS 5053 88 Ave SE
CITY Mecer Island STATE Wa ZIP CODE 98040
DRIVERS LICENSE 00000 STATE Wa SEX M BIRTH DATE 2/2?46 HGT 6 WGT 165 EYES Bl
VEHICLES LICENSE AAA000 STATE Wa COLOR YEAR 90 MAKE Saab TYPE 900
VIN
REGISTERED OWNER
ADDRESS
VIOLATION DATE MO 11 DAY 7 YEAR 2009 TIME HOUR: 935 DIST 2 DETACH 17
LOCATION 17 MILES E OF Enumckum ON SR410
VIOLATIONS Writing text while driving
OFFICERS SIGNATURE S Scott PERSONNEL NUMBER 850
[X] This is a warning, no further action is required.
[ ] You are released to take this vehicle to a place of repair. Continued operation on the roadway is not authorized.
[ ] CORRECT VIOLATION(S) IMMEDIATELY. Return this signed card for proof of compliance within 15/30 days. (if this box checked)
[X] DRIVERS SIGNATURE

Figure 5-54
Traffic Citation

5.60 Examine the list of e-mail messages in Figure 5-55.

Using the structure and example data items in this list, do the following:

A. Create a single-entity data model for this list. Specify the identifier and all entities.

B. Modify your answer to A to include entities SENDER and SUBJECT. Specify the identifiers and attributes of entities and the type and cardinalities of the relationships. Explain which cardinalities can be inferred from Figure 5-55 and which need to be checked out with users.

Figure 5-55
E-mail List

From	Subject	Date ↓	Size
WDA2259@sailmail.com	Big Wind	5/13/2009	3 KB
WDA2259@sailmail.com	Update	5/12/2009	4 KB
WDA2259@sailmail.com	Re: Saturday Am	5/11/2009	4 KB
WDA2259@sailmail.com	Re: Weather window!	5/10/2009	4 KB
WDA2259@sailmail.com	Re: Howdy!	5/10/2009	3 KB
WDA2259@sailmail.com	Still here	5/9/2009	3 KB
WDA2259@sailmail.com	Re: Turle Bay	5/8/2009	4 KB
WDA2259@sailmail.com	Turle Bay	5/8/2009	4 KB
WDA2259@sailmail.com	Re: Hi	5/6/2009	3 KB
WDA2259@sailmail.com	Sunday, Santa Maria	5/5/2009	3 KB
Ki6yu@aol.com	Cabo, Thurs. Noon	5/2/2009	2 KB
WDA2259@sailmail.com	turbo	5/1/2009	3 KB
WDA2259@sailmail.com	on our way	4/28/2009	3 KB
Tom Cooper	RE: Hola!	4/26/2009	3 KB
Tom Cooper	RE: Hola!	4/24/2009	2 KB
Tom Cooper	RE: Hola!	4/23/2009	3 KB

Amazon Ex. 1008.0212

Symbol	Name	Last	Change	% Chg
$COMPX	Nasdaq Combined Composite Index	1,400.74 ▼	-4.87	-0.35%
$INDU	Dow Jones Industrial Average Index	9,255.10 ▼	-19.80	-0.21%
$INX	S&P 500 INDEX	971.14 ▼	-5.84	-0.60%
ALTR	Altera Corporation	13.45 ▼	-0.450	-3.24%
AMZN	Amazon.com, Inc.	15.62 ▲	+0.680	+4.55%
CSCO	Cisco Systems, Inc.	13.39 ▼	-0.280	-2.05%
DELL	Dell Computer Corporation	24.58 ▼	-0.170	-0.69%
ENGCX	Enterprise Growth C	14.60 ▼	-0.210	-1.42%
INTC	Intel Corporation	18.12 ▼	-0.380	-2.05%
JNJ	Johnson & Johnson	53.29 ▼	-0.290	-0.54%
KO	Coca-Cola Company	56.70 ▼	-0.580	-1.01%
MSFT	Microsoft Corporation	53.96 ▲	+1.040	+1.97%
NKE	NIKE, Inc.	57.34 ▲	+0.580	+1.02%

Figure 5-56
Stock Quotations

C. The e-mail address in the From column in Figure 5-55 is in two different styles. One style has the true e-mail address; the second style (e.g., Tom Cooper) is the name of an entry in the user's e-mail directory. Create two categories of SENDER based on these two styles. Specify identifiers and attributes.

5.61 Examine the list of stock quotes in Figure 5-56.

Using the structure and example data items in this list, do the following:

A. Create a single-entity data model for this list. Specify the identifier and attributes.

B. Modify your answer to A to include the entities COMPANY and INDEX. Specify the identifier and attributes of the entities and the type and cardinalities of the relationships. Explain which cardinalities can be inferred from Figure 5-56 and which need to be checked out with users.

C. The list in Figure 5-56 is for a quote on a particular day at a particular time of day. Suppose that the list were changed to show closing daily prices for each of these stocks and that it includes a new column: QuoteDate. Modify your model in B to reflect this change.

D. Change your model in C to include the tracking of a portfolio. Assume the portfolio has an owner name, a phone number, an e-mail address, and a list of stocks held. The list includes the identity of the stock and the number of shares held. Specify all additional entities, their identifiers and attributes, and the type and cardinality of all relationships.

E. Change your answer to part D to keep track of portfolio stock purchases and sales in a portfolio. Specify entities, their identifiers and attributes, and the type and cardinality of all relationships.

5.62 Figure 5-57 shows the specifications for single-stage air compressor products.

Note that there are two product categories that are based on Air Performance: The A models are at 125 pounds per square inch of pressure, and the E models are at 150 pounds per square inch of pressure. Using the structure and example data items in this list, do the following:

A. Create a set of exclusive subtypes to represent these compressors. The supertype will have attributes for all single-stage compressors, and the subtypes will have attributes for products having the two different types of Air Performance. Assume that there might be additional products with different types of Air Performance.

Amazon Ex. 1008.0213

Single Stage

Set 95 to 150 PSI also available, substitute "E" for "A" in model number, i.e., K15A-30 make K15E-30

HP	Model	Tank Gal	Air Performance						Approx Ship Weight	Dimensions		
			A @ 125			E@ 150						
			Pump RPM	CFM Disp	DEL'D Air	Pump RPM	CFM Disp	DEL'D Air		L	W	H
1/2	F12A-17	17	680	3.4	2.2	590	2.9	1.6	135	37	14	25
3/4	F34A-17	17	1080	5.3	3.1	950	4.7	2.3	140	37	14	25
3/4	F34A-30	30	1080	5.3	3.1	950	4.7	2.3	160	38	16	31
1	K1A-30	30	560	6.2	4.0	500	5.7	3.1	190	38	16	34
1 1/2	K15A-30	30	870	9.8	6.2	860	9.7	5.8	205	49	20	34
1 1/2	K15A-60	60	870	9.8	6.2	860	9.7	5.8	315	38	16	34
2	K2A-30	30	1140	13.1	8.0	1060	12.0	7.0	205	49	20	39
2	K2A-60	60	1140	13.1	8.0	1060	12.0	7.0	315	48	20	34
2	GC2A-30	30	480	13.1	9.1	460	12.4	7.9	270	38	16	36
2	GC2A-60	60	480	13.1	9.1	460	12.4	7.9	370	49	20	41
3	GC3A-60	60	770	21.0	14.0	740	19.9	12.3	288	38	16	36
5	GC5A-80	60	770	21.0	14.0	740	19.9	12.3	388	49	20	41
5	GC5A-60	60	1020	27.8	17.8	910	24.6	15.0	410	49	20	41
5	GC5A-80	80	1020	27.8	17.8	910	24.6	15.0	460	62	20	41
5	J5A-80	60	780	28.7	19.0	770	28.6	18.0	570	49	23	43
5	J5A-80	80	780	28.7	19.0	770	28.6	18.0	610	63	23	43

Figure 5-57
Air Compressor Specifications

Specify the entities, identifiers, attributes, relationships, type of category cluster, and possible determinant.

B. Figure 5-58 shows a different model for the compressor data. Explain the entities, their type, the relationship, its type, and its cardinality. How well do you think this model fits the data shown in Figure 5-57?

C. Compare your answer in part A with the model in Figure 5-58. What are the essential differences between the two models? Which do you think is better?

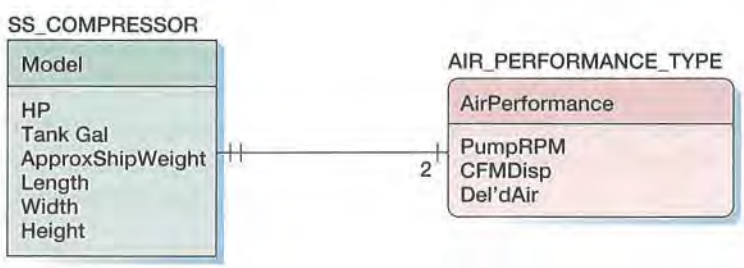
D. Suppose you had the job of explaining the differences in these two models to a highly motivated, intelligent end user. How would you accomplish this?

5.63 Figure 5-59 shows a listing of movie times at theaters in Seattle.

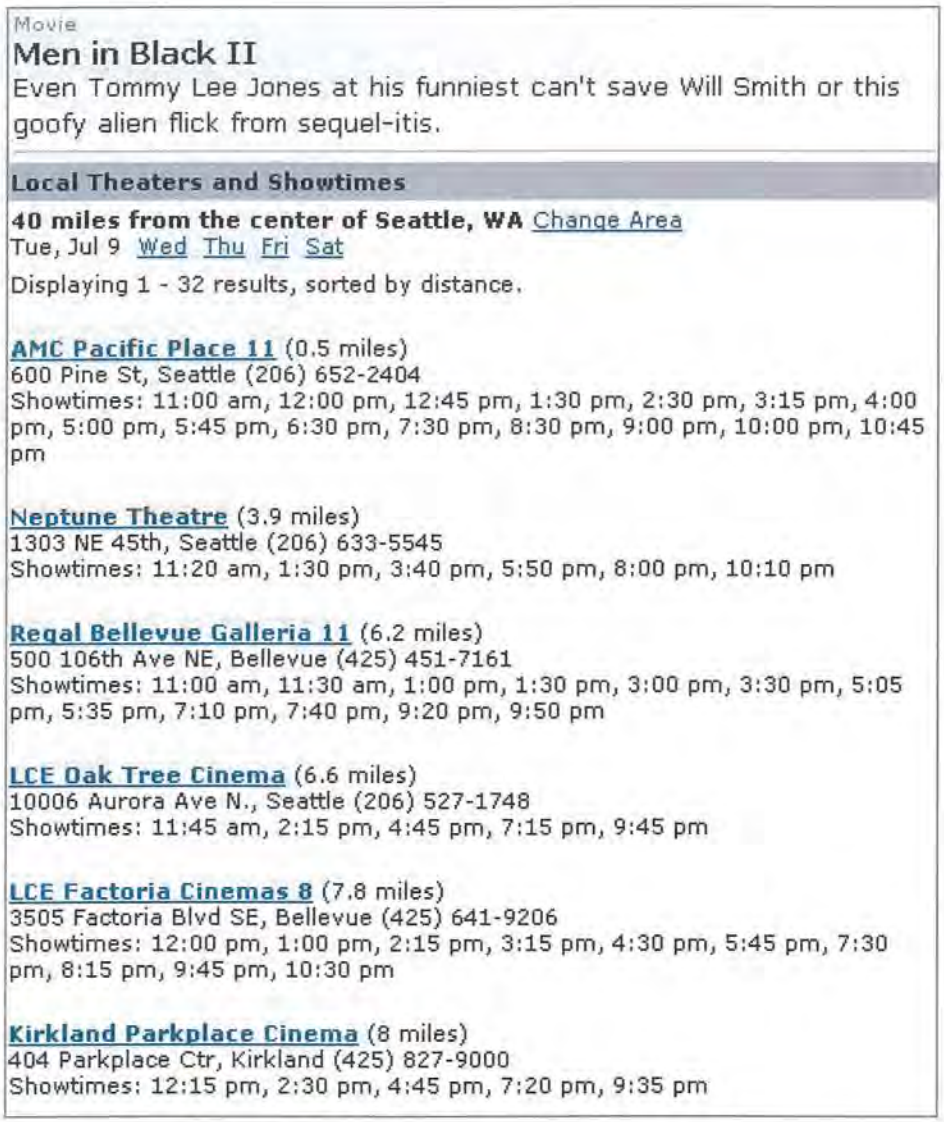
Using the data in this figure as an example, do the following:

A. Create a model to represent this report using the entities MOVIE, THEATER, and SHOW_TIME. Assume that theaters may show multiple movies. Although this report is for a particular day, your data model should allow for movie times on different days as well. Specify the identifier of the entities and their attributes. Name the relationships and the type and cardinality of all relationships. Explain which cardinalities you can logically deduce from Figure 5-59 and which need to be checked out with users. Assume that distance is an attribute of THEATER.

Figure 5-58
Alternative Model for Compressor Data



Amazon Ex. 1008.0214


Movie

Men in Black II

Even Tommy Lee Jones at his funniest can't save Will Smith or this goofy alien flick from sequel-itis.

Local Theaters and Showtimes

40 miles from the center of Seattle, WA Change Area
Tue, Jul 9 Wed Thu Fri Sat

Displaying 1 - 32 results, sorted by distance.

AMC Pacific Place 11 (0.5 miles)
600 Pine St, Seattle (206) 652-2404
Showtimes: 11:00 am, 12:00 pm, 12:45 pm, 1:30 pm, 2:30 pm, 3:15 pm, 4:00 pm, 5:00 pm, 5:45 pm, 6:30 pm, 7:30 pm, 8:30 pm, 9:00 pm, 10:00 pm, 10:45 pm

Neptune Theatre (3.9 miles)
1303 NE 45th, Seattle (206) 633-5545
Showtimes: 11:20 am, 1:30 pm, 3:40 pm, 5:50 pm, 8:00 pm, 10:10 pm

Regal Bellevue Galleria 11 (6.2 miles)
500 106th Ave NE, Bellevue (425) 451-7161
Showtimes: 11:00 am, 11:30 am, 1:00 pm, 1:30 pm, 3:00 pm, 3:30 pm, 5:05 pm, 5:35 pm, 7:10 pm, 7:40 pm, 9:20 pm, 9:50 pm

LCE Oak Tree Cinema (6.6 miles)
10006 Aurora Ave N., Seattle (206) 527-1748
Showtimes: 11:45 am, 2:15 pm, 4:45 pm, 7:15 pm, 9:45 pm

LCE Factoria Cinemas 8 (7.8 miles)
3505 Factoria Blvd SE, Bellevue (425) 641-9206
Showtimes: 12:00 pm, 1:00 pm, 2:15 pm, 3:15 pm, 4:30 pm, 5:45 pm, 7:30 pm, 8:15 pm, 9:45 pm, 10:30 pm

Kirkland Parkplace Cinema (8 miles)
404 Parkplace Ctr, Kirkland (425) 827-9000
Showtimes: 12:15 pm, 2:30 pm, 4:45 pm, 7:20 pm, 9:35 pm

Figure 5-59
Movie Time Listing

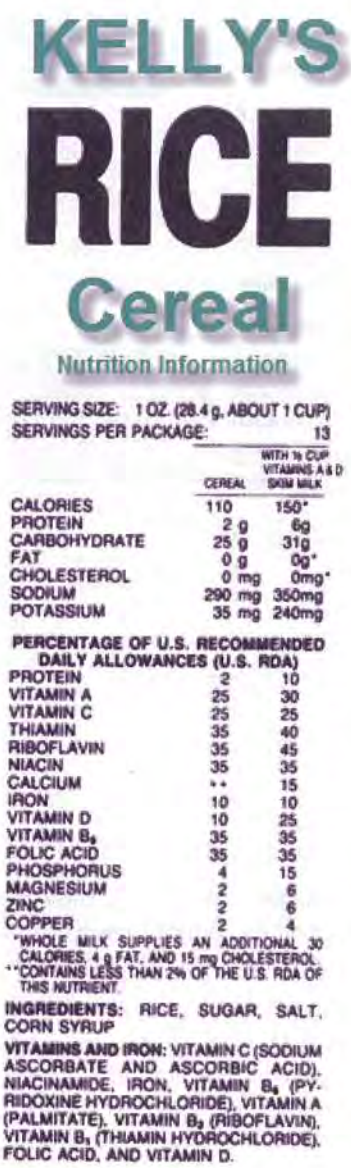
B. This report was prepared for a user who is located near downtown Seattle. Suppose that it is necessary to produce this same report for these theaters, but for a user located in a Seattle suburb, such as Bellevue, Renton, Redmond, or Tacoma. In this case, distance cannot be an attribute of THEATER. Change your answer in A for this situation. Specify the entity identifiers and attributes. Name the relationships and identify the type and cardinality of all relationships.

C. Suppose that you want to make this data model national. Change your answer to B so that it can be used for other metropolitan areas. Change your answer in A for this situation. Specify the entity identifiers and attributes. Name the relationships and identify the type and cardinality of all relationships.

D. Modify your answer to C to include the leading cast members. Assume that the role of a cast member is not to be modeled. Specify the identifier of new entities and their attributes. Name the relationships and identify the type and cardinality of all relationships.

E. Modify your answer to C to include the leading cast members. Assume that the role of a cast member is specified. Specify the identifier of new entities and their attributes. Name the relationships and identify the type and cardinality of all relationships.

Amazon Ex. 1008.0215



KELLY'S
RICE
Cereal
Nutrition Information

SERVING SIZE: 1 OZ. (28.4 g, ABOUT 1 CUP)
SERVINGS PER PACKAGE: 13

	CEREAL	WITH ½ CUP VITAMINS A & D SKIM MILK
CALORIES	110	150*
PROTEIN	2 g	6g
CARBOHYDRATE	25 g	31g
FAT	0 g	0g*
CHOLESTEROL	0 mg	0mg*
SODIUM	290 mg	350mg
POTASSIUM	35 mg	240mg

PERCENTAGE OF U.S. RECOMMENDED DAILY ALLOWANCES (U.S. RDA)

PROTEIN	2	10
VITAMIN A	25	30
VITAMIN C	25	25
THIAMIN	35	40
RIBOFLAVIN	35	45
NIACIN	35	35
CALCIUM	**	15
IRON	10	10
VITAMIN D	10	25
VITAMIN B_6	35	35
FOLIC ACID	35	35
PHOSPHORUS	4	15
MAGNESIUM	2	6
ZINC	2	6
COPPER	2	4

*WHOLE MILK SUPPLIES AN ADDITIONAL 30 CALORIES, 4 g FAT, AND 15 mg CHOLESTEROL.
**CONTAINS LESS THAN 2% OF THE U.S. RDA OF THIS NUTRIENT.

INGREDIENTS: RICE, SUGAR, SALT, CORN SYRUP

VITAMINS AND IRON: VITAMIN C (SODIUM ASCORBATE AND ASCORBIC ACID), NIACINAMIDE, IRON, VITAMIN B_6 (PYRIDOXINE HYDROCHLORIDE), VITAMIN A (PALMITATE), VITAMIN B_2 (RIBOFLAVIN), VITAMIN B_1 (THIAMIN HYDROCHLORIDE), FOLIC ACID, AND VITAMIN D.

FDA REPORT #6272
Date: June 30, 2009
Issuer: Kelly's Corporation
Report Title: Product Summary by Ingredient

Corn	Kelly's Corn Cereal
	Kelly's Multigrain Cereal
	Kelly's Crunchy Cereal
Corn syrup	Kelly's Corn Cereal
	Kelly's Rice Cereal
	Kelly's Crunchy Cereal
Malt	Kelly's Corn Cereal
	Kelly's Crunchy Cereal
Wheat	Kelly's Multigrain Cereal
	Kelly's Crunchy Cereal

(a)

SUPPLIERS LIST
Date: June 30, 2009

Ingredient	Supplier	Price
Corn	Wilson	2.80
	J. Perkins	2.72
	Pollack	2.83
	McKay	2.80
Wheat	Adams	1.19
	Kroner	1.19
	Schmidt	1.22
Barley	Wilson	0.85
	Pollack	0.84

(b)

Figure 5-60
Cereal Product Reports

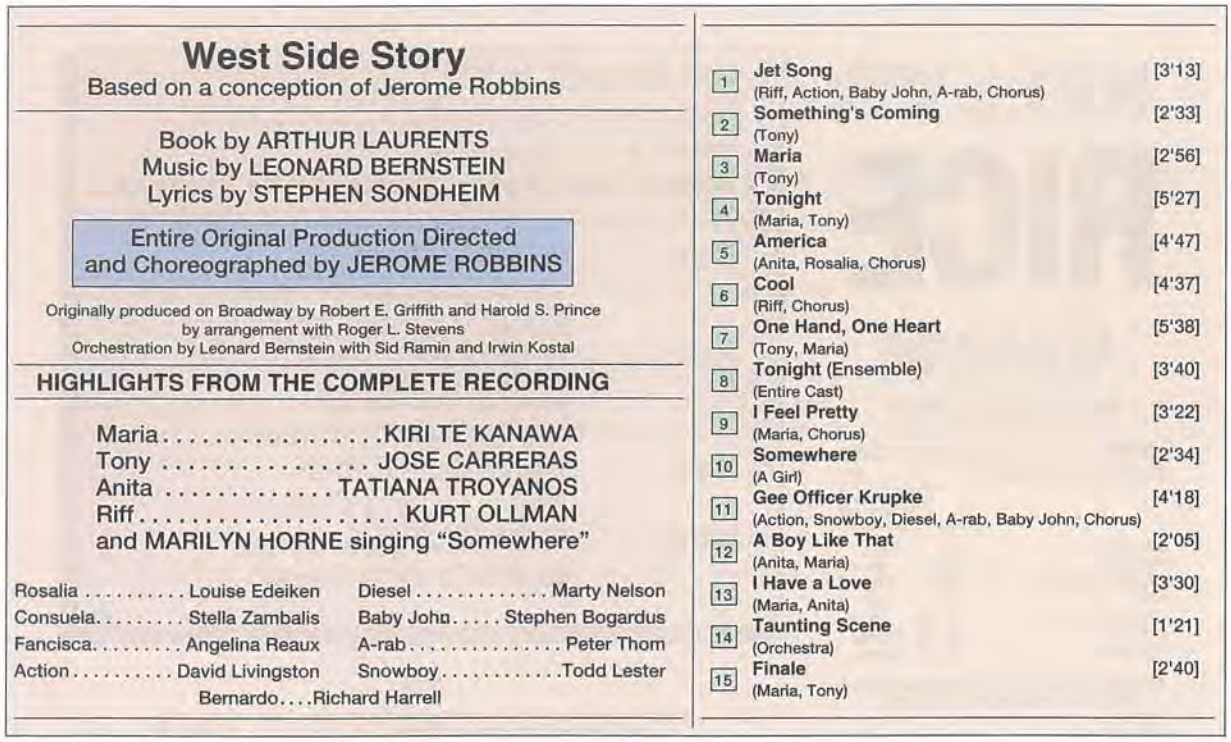
5.64 Consider the three reports in Figure 5-60. The data are samples of data that would appear in the reports like these.

A. Make a list of as many potential entities as these reports suggest.

B. Examine your list to determine whether any entities are synonyms. If so, consolidate your list.

C. Construct an IE Crow's Foot model showing relationships among your entities. Name each relationship and specify cardinalities. Indicate which cardinalities you can justify on the basis of these reports and which you will need to check out with the users.

Amazon Ex. 1008.0216



West Side Story
Based on a conception of Jerome Robbins

Book by ARTHUR LAURENTS
Music by LEONARD BERNSTEIN
Lyrics by STEPHEN SONDHEIM

Entire Original Production Directed and Choreographed by JEROME ROBBINS

Originally produced on Broadway by Robert E. Griffith and Harold S. Prince
by arrangement with Roger L. Stevens
Orchestration by Leonard Bernstein with Sid Ramin and Irwin Kostal

HIGHLIGHTS FROM THE COMPLETE RECORDING

Maria . . . . . KIRI TE KANAWA
Tony . . . . . JOSE CARRERAS
Anita . . . . . TATIANA TROYANOS
Riff . . . . . KURT OLLMAN
and MARILYN HORNE singing "Somewhere"

Rosalia . . . . . Louise Edeiken
Consuela . . . . . Stella Zambalis
Fancisca . . . . . Angelina Reaux
Action . . . . . David Livingston
Diesel . . . . . Marty Nelson
Baby John . . . . . Stephen Bogardus
A-rab . . . . . Peter Thom
Snowboy . . . . . Todd Lester
Bernardo . . . . Richard Harrell

1 Jet Song [3'13]
(Riff, Action, Baby John, A-rab, Chorus)
2 Something's Coming [2'33]
(Tony)
3 Maria [2'56]
(Tony)
4 Tonight [5'27]
(Maria, Tony)
5 America [4'47]
(Anita, Rosalia, Chorus)
6 Cool [4'37]
(Riff, Chorus)
7 One Hand, One Heart [5'38]
(Tony, Maria)
8 Tonight (Ensemble) [3'40]
(Entire Cast)
9 I Feel Pretty [3'22]
(Maria, Chorus)
10 Somewhere [2'34]
(A Girl)
11 Gee Officer Krupke [4'18]
(Action, Snowboy, Diesel, A-rab, Baby John, Chorus)
12 A Boy Like That [2'05]
(Anita, Maria)
13 I Have a Love [3'30]
(Maria, Anita)
14 Taunting Scene [1'21]
(Orchestra)
15 Finale [2'40]
(Maria, Tony)

Figure 5-61
CD Cover

5.65 Consider the CD cover in Figure 5-61.

A. Specify identifiers and attributes for the entities CD, ARTIST, ROLE, and SONG.

B. Construct a crow's foot model showing relationships among these four entities. Name each relationship and specify cardinalities. Indicate which cardinalities you can justify on the basis of the CD cover and which you will need to check out with the users.

C. Consider a CD that does not involve a musical, so there is no need for ROLE. However, the entity SONG_WRITER is needed. Create a crow's foot model for CD, ARTIST, SONG, and SONG_WRITER. Assume that an ARTIST can either be a group or an individual. Assume that some artists record individually and as part of a group.

D. Combine the models you developed in your answers to B and C. Create new entities if necessary, but strive to keep your model as simple as possible. Specify identifiers and attributes of new entities, name new relationships, and indicate their cardinalities.

5.66 Consider the data model in Figure 5-45. How should this model be altered if the users want to keep track of how many of each part are used? Suppose, for example, that the wheel assembly requires four washers and the handle assembly requires just one, and the database must store these quantities. Hint: Adding Quantity to this N:M relationship is analogous to adding Price to the N:M relationship in Figure 5-24.

5.67 The data model in Figure 5-52 uses the attribute Room in COLLEGE and DEPARTMENT, but uses OfficeNumber in PROFESSOR. These attributes have the same kind of data, even though they have different names. Examine Figure 5-46 and explain how this situation came to be. Do you think having different names for the same attribute types is rare? Do you think it is a problem? Why or why not?

Amazon Ex. 1008.0217



Suppose that you have been hired by Marcia's Dry Cleaning to create a database application to track customers, orders, and items. Marcia also wants to start a Frequent Cleaner's Club, whereby she will offer a 50 percent discount on every 10th customer order.

A. Using your knowledge, create a data model for Marcia's business. Name each entity, describe its type, and indicate all attributes and identifiers. Name each relationship, describe its type, and specify minimum and maximum cardinalities.

B. List any item in your answer to A that you believe should be checked out with Marcia and/or her employees.

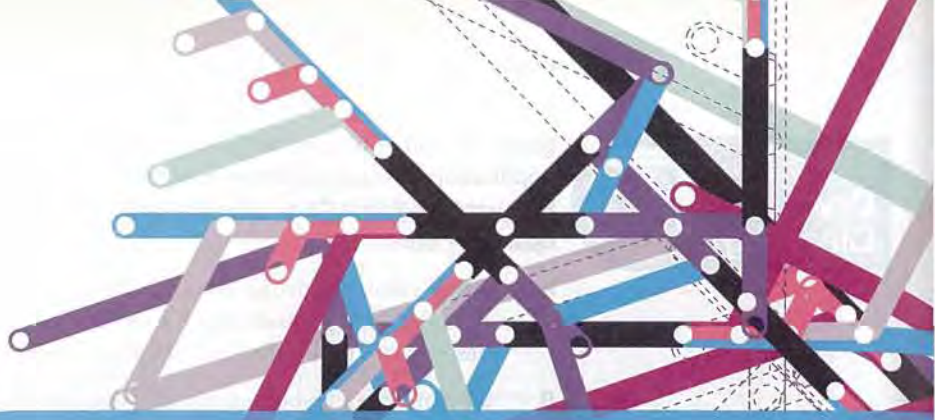


Suppose that you have been hired by Morgan Importing to create a database application to track stores, purchases, shipments, and shippers. Sometimes several items are purchased from a store on a single visit, but do not assume that all of the items are placed on the same shipment. You want to track each item in a shipment and assign an insurance value to each item.

A. Using your knowledge, create a data model for Morgan Importing. Name each entity, describe its type, and indicate all attributes and identifiers. Name each relationship, describe its type, and specify minimum and maximum cardinalities.

B. List any item in your answer to A that you believe should be checked out with Phillip Morgan and/or his employees.

Amazon Ex. 1008.0218



6 Transforming Data Models into Database Designs

Chapter Objectives

- To understand how to transform data models into database designs.
- To be able to identify primary keys and understand when to use a surrogate key.
- To understand the use of referential integrity constraints.
- To understand the use of referential integrity actions.
- To be able to represent ID-dependent, 1:1, 1:N, and N:M relationships as tables.
- To be able to represent weak entities as tables.
- To be able to represent supertype/subtypes as tables.
- To be able to represent recursive relationships as tables.
- To be to represent ternary relationships as tables.
- To be able to implement referential integrity actions required by minimum cardinalities.

This chapter explains the transformation of entity-relationship data models into relational database designs. This transformation consists of three primary tasks: (1) replacing entities and attributes with tables and columns; (2) representing relationships and maximum cardinalities by placing foreign keys; and (3) representing minimum cardinality by defining actions to constrain activities on values of primary and foreign keys. Steps 1 and 2 are relatively easy to understand and accomplish; step 3 may be easy or difficult, depending on the minimum cardinality type.

Amazon Ex. 1008.0219

Create a Table for Each Entity

We begin the database design by creating a table for each entity using the steps shown in Figure 6-1. In most cases, the table is assigned the same name as the entity. Each attribute of the entity becomes a column of the table. The identifier of the entity becomes the primary key of the table. The example in Figure 6-2 shows the creation of the EMPLOYEE table from the EMPLOYEE entity. In this text, to differentiate entities from tables, we will show entities with shadowed boxes and tables with nonshadowed boxes. This notation will help clarify our discussion, but be aware that it is not standard notation across the industry.

Be certain that you understand the difference between these similar-looking graphics. The shadowed rectangle in Figure 6-2(a) represents a logical structure that has no physical existence. It is a blueprint. The nonshadowed rectangle in Figure 6-2(b) represents a database table. It is the same as the following notation that we used in Chapters 3 and 4:

EMPLOYEE (EmployeeNumber, EmployeeName, Phone, Email, HireDate, ReviewDate, EmpCode)

Note, too, the key symbol next to EmployeeNumber. It documents the fact that Employee Number is the table key, just as the underline does in the notation used in Chapters 3 and 4.

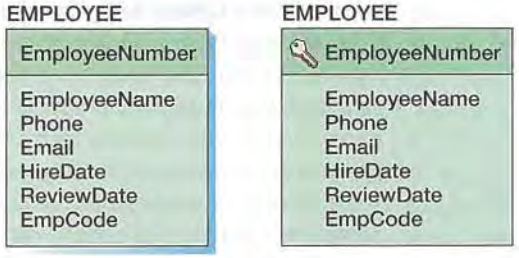
Figure 6-1

Steps for Transforming a Data Model into a Database Design

1. Create a table for each entity:
   - Specify primary key (consider surrogate keys, as appropriate)
   - Specify candidate keys
   - Specify properties for each column:
     - Null status
     - Data type
     - Default value (if any)
     - Specify data constraints (if any)
   - Verify normalization
2. Create relationships by placing foreign keys
   - Relationships between strong entities (1:1, 1:N, N:M)
   - Identifying relationships with ID-dependent entities (intersection tables, association patterns, multivalued attributes, archetype/instance patterns)
   - Relationships between a strong entity and a weak but non-ID-dependent entity (1:1, 1:N, N:M)
   - Mixed relationships
   - Relationships between supertype/subtype entities
   - Recursive relationships (1:1, 1:N, N:M)
3. Specify logic for enforcing minimum cardinality:
   - M-O relationships
   - O-M relationships
   - M-M relationships

Figure 6-2

Transforming an Entity to a Table



(a) EMPLOYEE Entity (b) EMPLOYEE Table

Amazon Ex. 1008.0220

Selecting the Primary Key

The selection of the primary key is important. The DBMS will use the primary key to facilitate searching and sorting of table rows, and some DBMS products use it to organize table storage. DBMS products almost always create indexes and other data structures using the values of the primary key.

The ideal primary key is short, numeric, and fixed. EmployeeNumber in Figure 6-2 meets all of these conditions and is acceptable. Beware of primary keys such as EmployeeName, Email, (AreaCode, PhoneNumber), (Street, City, State, Zip), and other long character columns. In cases like these, when the identifier is not short, numeric, or fixed, consider using another candidate key as the primary key. If there are no candidate keys, or if none of them is any better, consider using a surrogate key.

A **surrogate key** is a DBMS-supplied identifier of each row of a table. Surrogate key values are unique within the table, and they never change. They are assigned when the row is created, and they are destroyed when the row is deleted. Surrogate key values are the best possible primary keys because they are designed to be short, numeric, and fixed. Because of these advantages, some organizations have gone so far as to require that surrogates be used for the primary key of every table.

Before endorsing such a policy, however, consider two disadvantages of surrogate keys. First, their values have no meaning to a user. Suppose you want to determine the department to which an employee is assigned. If DepartmentName is a foreign key in EMPLOYEE, then when you retrieve an employee row, you obtain a value such as 'Accounting' or 'Finance'. That value may be all that you need to know about department.

Alternatively, if you define the surrogate key DepartmentID as the primary key of DEPARTMENT, then DepartmentID will also be the foreign key in EMPLOYEE. When you retrieve a row of EMPLOYEE, you will get back a number such as 123499788 for the DepartmentID, a value that has no meaning to you at all. You have to perform a second query on DEPARTMENT to obtain DepartmentName.

The second disadvantage of surrogate keys arises when data are shared among different databases. Suppose, for example, that a company maintains three different SALES databases, one for each of three different product lines. Assume that each of these databases has a table called SALES_ORDER that has a surrogate key called ID. The DBMS assigns values to IDs so that they are unique within a particular database. It does not, however, assign ID values so that they are unique across the three different databases. Thus, it is possible for two different SALES_ORDER rows, in two different databases, to have the same ID value.

This duplication is not a problem until data from the different databases are merged. When that happens, to prevent duplicates, ID values will need to be changed. However, if ID values are changed, then foreign key values may need to be changed as well, and the result is a mess, or at least much work to prevent a mess.

It is, of course, possible to construct a scheme using different starting values for surrogate keys in different databases. Such a policy ensures that each database has its own range of surrogate key values. This requires careful management and procedures, however; and if the starting values are too close to one another, the ranges will overlap and duplicate surrogate key values will result.



Some database designers take the position that, for consistency, if one table has a surrogate key, all of the tables in the database should have a surrogate key. Others think that such a policy is too rigid; after all, there are good data keys, such as ProductID. If such a key exists, it should be used instead of a surrogate key. Your organization may have standards on this issue that you should follow.

Be aware that DBMS products vary in their support for surrogate keys. Access, SQL Server and MySQL provide them. SQL Server allows the designer to pick the starting value and increment of the key, and MySQL allows the designer to pick the starting value. Oracle, however, does not provide direct support for surrogate keys, but you can obtain the essence of them in a rather backhanded way, as discussed in Chapter 10A.

We use surrogate keys unless there is some strong reason not to. In addition to the advantages described here, the fact that they are fixed simplifies the enforcement of minimum cardinality, as you will learn in the last section of this chapter.

Amazon Ex. 1008.0221

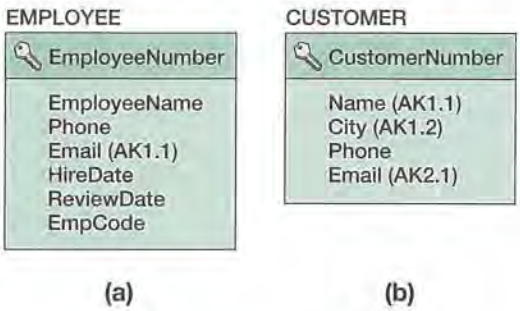


Figure 6-3
Representing Candidate (Alternative) Keys

Specifying Candidate (Alternate) Keys

The next step in creating a table is to specify candidate keys. As discussed in Chapter 3, **candidate keys** are alternative identifiers of unique rows in a table. Some products use the term **alternate key (AK)** rather than candidate key, but the two terms are synonymous. Figure 6-3 illustrates the use of alternate keys.

Figure 6-3(a) shows EMPLOYEE with a primary key of EmployeeNumber and a candidate, or alternate, key of Email. In Figure 6-3(b), CustomerNumber is the primary key of CUSTOMER, and both the composite (Name, City) and Email are candidate keys. In these diagrams, the symbol AK*n.m* means the *n*th alternate key and the *m*th column of that alternate key. In the EMPLOYEE table, Email is labeled AK1.1 because it is the first alternate key and the first column of that key. CUSTOMER has two alternate keys. The first is a composite of two columns, which are labeled AK1.1 and AK1.2. The nomenclature Name (AK1.1) means that Name is the first column of the first alternate key, and City (AK1.2) means that City is the second column of the first alternate key. In CUSTOMER, Email is marked as AK2.1 because it is the first (and only) column of the second alternate key.

Specify Column Properties

The next step in the creation of a relation is to specify the column properties. Four properties are shown in Figure 6-1: null status, data type, default value, and data constraints.

Null Status

Null status refers to whether the column can have a null value. Typically, null status is specified by using the phrase NULL if nulls are allowed and NOT NULL if not. Thus, NULL does not mean that the column is always null; it means that null values are allowed. Because of this possible confusion, some people prefer the term NULL ALLOWED rather than NULL. Figure 6-4 shows the null status of each of the columns in the EMPLOYEE table.

BY THE WAY The EMPLOYEE table in Figure 6-4 contains a subtlety. EmployeeNumber, the primary key, is marked NOT NULL, but Email, the alternate key, is marked NULL. It makes sense that EmployeeNumber should not be allowed to be null. If it were, and if more than one row had a null value, then EmployeeNumber would not identify a unique row. Why, however, should Email be allowed to have null values?

The answer is that alternate keys often are used just to ensure uniqueness. Marking Email as a (possibly null) alternate key means that Email need not have a value, but, if it has one, that value will be different from all other values of Email in the EMPLOYEE table. This answer is dissatisfying because it means that alternate keys are not truly alternate *primary* keys. Alas, that's the way it is. Just know that primary keys can never be null but that alternate keys can be.

Amazon Ex. 1008.0222

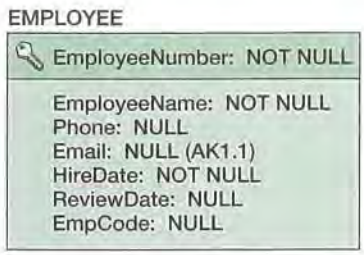


Figure 6-4
Table Display Showing Null Status

Data Type

The next step is to define the data type for each column. Unfortunately, each DBMS provides a different set of data types. For example, Microsoft Access has a data type called Currency; SQL Server has a data type called Money; and Oracle has no data type for currency. Instead, with Oracle, you use the numeric data type for currency values. A summary of data types for Oracle, SQL Server, and MySQL appears in Chapter 7 as Figure 7-4.

If you know which DBMS you will be using to create the database, you can use that product's data types in your design. Figure 6-5 illustrates the display of data types in a table using the data types for SQL Server (e.g., datetime is an SQL Server data type).

In fact, with many data modeling products, such as Computer Associate's ERwin, you can specify the DBMS you will use and the data modeling product will supply the appropriate set of data types. Other products are DBMS specific. On the other hand, Sun Microsystems' MySQL Workbench is intended to design databases for MySQL, and therefore uses MySQL-specific data types.

If you do not know which DBMS product you will be using, or if you want to preserve independence from a particular DBMS, you can specify the data types in a generic way. Typical generic data types are CHAR (n) for a fixed-length character string of length *n*; VARCHAR (n) for a variable-length character string having a maximum length of *n*; DATE; TIME; MONEY; INTEGER; and DECIMAL. If you work for a larger organization, that company probably has its own generic data standards. If so, you should use those data standards.

Figure 6-6 shows the EMPLOYEE table showing both data type and null status. The display becomes crowded, however; and from now on we will show tables with just column names. With most products, you can turn such displays on or off depending on the work you are doing.

BY THE WAY The fact that a design tool is dedicated to one DBMS product does *not* mean that it cannot be used to design databases for other DBMSs. For example, an SQL Server database can be designed in MySQL Workbench and most of the design will be correct. You will however, have to understand the relevant differences in the DBMS products and make adjustments when creating the actual database.

Figure 6-5
Table Display Showing Data Types

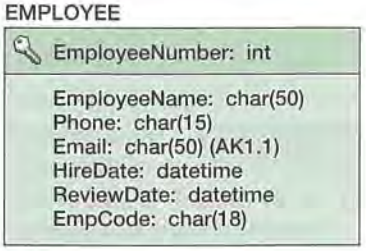
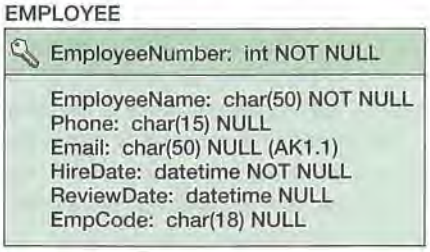


Figure 6-6
Table Display Showing Null Status and Data Types



Amazon Ex. 1008.0223

Default Value

A **default value** is a value supplied by the DBMS when a new row is created. The value can be a constant, such as the string 'New Hire' for the EmpCode column in EMPLOYEE, or it can be the result of a function, such as the date value of the computer's clock for the HireDate column.

In some cases, default values are computed using more complicated logic. The default value for a price, for example, might be computed by applying a markup to a default cost and then reducing that marked up price by a customer's discount. In such a case, an application component or a trigger (discussed in Chapter 7) will be written to supply such a value.

It is possible to use the data modeling tool to record default values, but such values often are shown in separate design documentation. Figure 6-7, for example, shows one way that default values are documented.

Data Constraints

Data constraints are limitations on data values. There are several different types. **Domain constraints** limit column values to a particular set of values. For example, EMPLOYEE.EmpCode could be limited to one of the values ['New Hire', 'Hourly', 'Salary', 'Part Time']. **Range constraints** limit values to a particular interval of values. EMPLOYEE.HireDate, for example, could be limited to dates between January 1, 1990, and December 31, 2010.

An **intrarelation constraint** limits a column's values in comparison with other columns in the same table. The constraint that EMPLOYEE.ReviewDate be at least three months after EMPLOYEE.HireDate is an intrarelation constraint. An **interrelation constraint** limits a column's values in comparison with other columns in other tables. An example for the CUSTOMER table is that CUSTOMER.Name must not be equal to BAD_CUSTOMER.Name, where BAD_ CUSTOMER is a table that contains a list of customers with credit and balance problems.

Referential integrity constraints, which we discussed in Chapter 3, are one type of interrelation constraint. Because they are so common, sometimes they are documented only when they are not enforced. For example, to save work, a design team might say that every foreign key is assumed to have a referential integrity constraint to the table that it references and that only exceptions to this rule are documented.

Verify Normalization

The last task in step 1 of Figure 6-1 is to verify table normalization. When data models are developed using forms and reports as guides, they generally result in normalized entities. This occurs because the structures of forms and reports usually reflect how users think about their data. Boundaries of a form, for example, often show the range of a functional dependency. If this is hard to understand, think of a functional dependency as a theme. A well-designed form or report will bracket themes using lines, colors, boxes, or other graphical elements. Those graphical hints will have been used by the data modeling team to develop entities, and the result will be normalized tables.

Figure 6-7

Sample Documentation for Default Values

Table	Column	Default Value
ITEM	ItemNumber	Surrogate key
ITEM	Category	None
ITEM	ItemPrefix	If Category = 'Perishable' then 'P' If Category = 'Imported' then 'I' If Category = 'One-off' then 'O' Otherwise = 'N'
ITEM	ApprovingDept	If ItemPrefix = 'I' then 'SHIPPING/PURCHASING' Otherwise = 'PURCHASING'
ITEM	ShippingMethod	If ItemPrefix = 'P' then 'Next Day' Otherwise = 'Ground'

Amazon Ex. 1008.0224

All of this, however, should be verified. You need to ask whether the resulting tables are in BCNF and whether all multivalued dependencies have been removed. If not, the tables should probably be normalized. However, as we discussed in Chapter 4, sometimes normalization is undesirable. Thus, you should also examine your tables to determine if any normalized ones should be denormalized.

Create Relationships

The result of step 1 is a set of complete, but independent, tables. The next step is to create relationships. In general, we create relationships by placing foreign keys into tables. The way in which this is done and the properties of the foreign key columns depend on the type of relationship. In this section, we consider each of the relationships described in Chapter 5: nonidentifying relationships between strong entities, identifying relationships between ID-dependent entities, relationships in mixed entity patterns, relationships between a supertype and its subtypes, and recursive relationships. We conclude this section with a discussion of special cases of ternary relationships.

Relationships Between Strong Entities

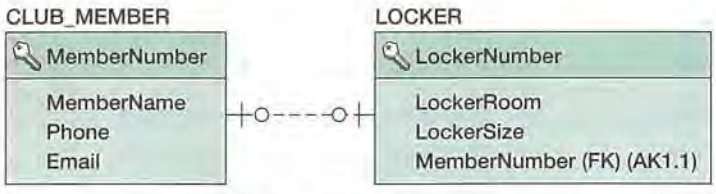
As you learned in Chapter 5, nonidentifying relationships between strong entities are characterized by their maximum cardinality. There are three types of these relationships: 1:1, 1:N, and N:M.

1:1 Relationships Between Strong Entities

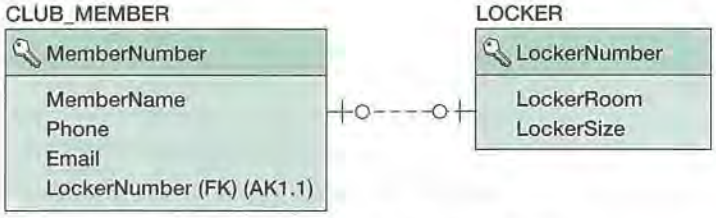
After the tables corresponding to the strong entities have been designed, a 1:1 relationship between these entities can be represented in one of two ways. You can place the primary key of the first table in the second as a foreign key, or you can place the primary key of the second table in the first as a foreign key. Figure 6-8 shows the representation of the 1:1 nonidentifying relationship between CLUB_MEMBER and LOCKER. In Figure 6-8(a), MemberNumber is placed in LOCKER as a foreign key. In Figure 6-8(b), LockerNumber is placed in CLUB_MEMBER as a foreign key.

Either of these designs will work. If you have a club member's number and want his or her locker, then using the design in Figure 6-8(a) you can query the LOCKER table for the given value of MemberNumber. But, if you have the LockerNumber and want the club member's data, then using the design in Figure 6-8(a) you can query the LOCKER table for the LockerNumber, obtain the MemberNumber, and use that value to query the CLUB_MEMBER table for the rest of the club member's data.

Figure 6-8
The Two Alternatives for Representing a 1:1 Relationship Between Strong Entities



(a) With Foreign Key in LOCKER



(b) With Foreign Key in CLUB_MEMBER

Amazon Ex. 1008.0225

Follow a similar procedure to verify that the design in Figure 6-8(b) works as well. However, one data constraint applies to both designs. Because the relationship is 1:1, a given value of a foreign key can appear only once in the table. For example, in the design in Figure 6-8(a), a given value of MemberNumber can appear just once; each value must be unique in the LOCKER table. If a value of MemberNumber were to appear in two rows, then a member would be assigned to two lockers, and the relationship would not be 1:1.

To cause the DBMS to enforce the required uniqueness of the foreign key value, we define the foreign key column as unique. This can be done either directly in the column definition of the foreign key (in which case there is no designation in the table diagram) or by defining the foreign key as an alternate key. This latter technique, though common, is a bit confusing because, logically, MemberNumber is not an alternate key for LOCKER. We are just using the fact that alternate keys are unique to document the uniqueness of the foreign key in a 1:1 relationship. Depending upon the database design software being used, the alternate key designation may appear in the database design of the tables and the relationship, and this is illustrated in Figure 6-8(a). A similar technique is used on the foreign key LockerNumber in Figure 6-8(b).

Although either of the designs in Figure 6-8 will work, a design team may prefer one over the other. If the minimum cardinalities of the relationship are either M-O or O-M, then one design will be *greatly* preferred, as you will learn in the section on minimum cardinality design. Also, application requirements may mean that one design is faster than the other.

To summarize, to represent a 1:1 strong entity relationship, place the key of one table in the other table. Enforce the maximum cardinality by defining the foreign key as unique (or as an alternate key).

1:N Relationships Between Strong Entities

After the tables corresponding to the strong entities have been designed, a 1:N relationship between the entities is represented by placing the primary key of the table on the one side into the table on the many side as a foreign key. Recall from Chapter 5 that the term *parent* is used to refer to the table on the one side and the term *child* is used to refer to the table on the many side. Using this terminology, you can summarize the design of 1:N relationships by saying, "Place the primary key of the parent in the child as a foreign key." This is illustrated in Figure 6-9.

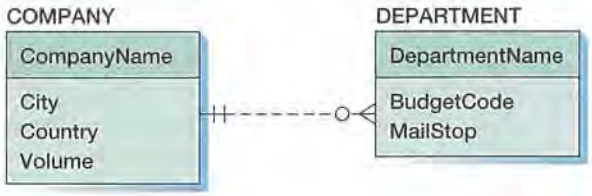
Figure 6-9 shows an E-R diagram for the 1:N relationship between the COMPANY and DEPARTMENT entities. The relationship is represented in the database design in Figure 6-9(b) by placing the primary key of the parent (CompanyName) in the child (DEPARTMENT) as a foreign key. Because parents have many children (the relationship is 1:N), there is no need to make the foreign key unique.

For 1:N relationships between strong entities, that's all there is to it. Just remember: "Place the primary key of the parent in the child as a foreign key."

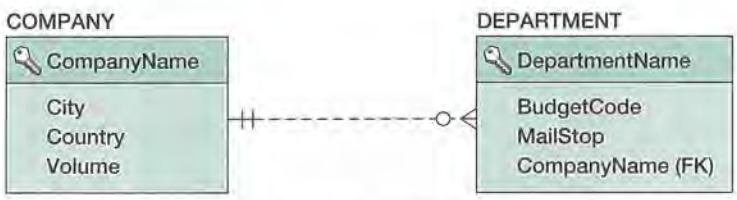
N:M Relationships Between Strong Entities

Again, we must first create the database design tables from the data model entities, and then create the relationship. However, the situation for N:M relationships is more complicated. The

Figure 6-9
Representing a 1:N Relationship Between Strong Entities

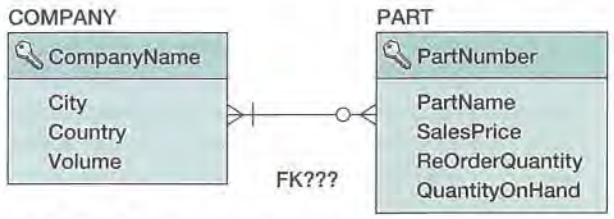


(a) 1:N Relationship Between Strong Entities

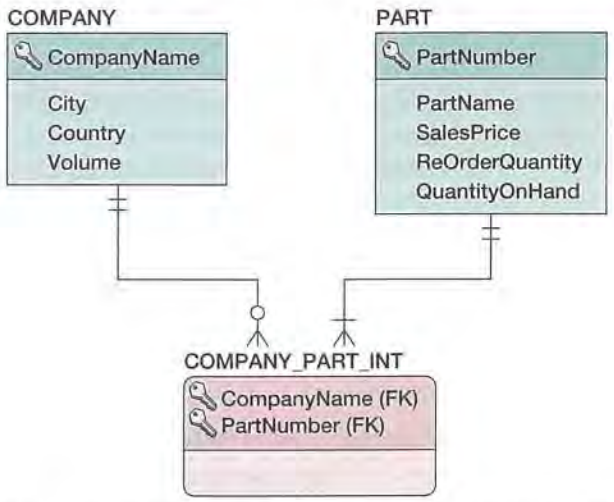


(b) Placing the Primary Key of the Parent in the Child as a Foreign Key

Amazon Ex. 1008.0226



(a) The Foreign Key Has No Place in Either Table



(b) Foreign Keys Placed in ID-Dependent Intersection Table

Figure 6-10
Representing an N:M Relationship Between Strong Entities

problem is that there is no place in either table in an N:M relationship in which to place the foreign key. Consider the example in Figure 6-10(a), which shows a relationship between COMPANY and PART that specifies which companies can supply which parts. A COMPANY may supply many PARTs, and a PART may be supplied by many different COMPANY(ies).

Suppose we try to represent this relationship by placing the primary key of one table as a foreign key in the second table, as we did for 1:N relationships. Say we place the primary key of PART in COMPANY as follows:

COMPANY (CompanyName, City, Country, Volume, *PartNumber*)

PART (PartNumber, PartName, SalesPrice, ReOrderQuantity, QuantityOnHand)

With this design, a given PartNumber may appear in many rows of COMPANY so that many companies can supply the part. But, how do we show that a company can supply many parts? There is only space to show one part. We do not want to duplicate the entire row for a company just to show a second part; such a strategy would result in unacceptable data duplication and data integrity problems. Therefore, this is not an acceptable solution, and a similar problem will occur if we try to place the primary key of COMPANY, CompanyName, into PART as a foreign key.

The solution is to create a third table, called an **intersection table**. Such a table shows the correspondences of a given company and a given part. It holds only the primary keys of the two tables as foreign keys, and this combination of keys serves as the composite primary key of the intersection table itself. The intersection holds only the key data; it contains no other user data. For the example in Figure 6-10(a) we create the following intersection table:

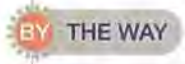
COMPANY_PART_INT (*CompanyName*, *PartNumber*)

The COMPANY_PART_INT table has one row for each company–part combination. Notice that both columns are part of the primary key, and that each column is a foreign key to a

Amazon Ex. 1008.0227

different table. Because both columns are keys of other tables, intersection tables are always ID dependent on both of their parent tables and the relationships with the parent tables are identifying relationships.

Thus, in Figure 6-10(b) COMPANY_PART_INT is shown as ID dependent with identifying relationship lines. Like all ID-dependent tables, the parent tables are required; COMPANY_PART_INT requires both a COMPANY and PART. The parents may or may not require an intersection table row, depending on application requirements. In Figure 6-10(b), a COMPANY need not supply a PART, but a PART must be supplied by at least one COMPANY.



The problem for the data models of N:M relationships between strong entity is that they have no direct representation. An N:M relationship must always be decomposed into two 1:N relationships using an intersection table in the database design. This is why products like MySQL Workbench and Visio are unable to represent N:M relationships in a data model. These products force you to make the transformation to two 1:N relationships ahead of time, during modeling. As stated in Chapter 5, however, most data modelers consider this requirement to be a nuisance because it adds complexity to data modeling when the whole purpose of data modeling is to reduce complexity to the logical essentials.

Relationships Using ID-Dependent Entities

Figure 6-11 summarizes the four uses for ID-dependent entities. We have already described the first use shown in Figure 6-11: the representation of N:M relationships. As shown in Figure 6-10, an ID-dependent intersection table is created to hold the foreign keys of the two tables participating in the relationship, and indentifying 1:N relationships are created between each table and the intersection table.

The other three uses shown in Figure 6-11 were discussed in Chapter 5, and here we will describe the creation of tables and relationships for each of these three uses.

Association Relationships

As we discussed in Chapter 5, an association relationship is subtly close to an N:M relationship between two strong entities. The only difference between the two types of relationships is that an association relationship has one or more attributes that pertain to the relationship and not to either of the entities. These attributes must be added to what would otherwise be the intersection table in the N:M relationship. Figure 6-12(a) shows the association relationship data model created in Figure 5-22. In this example, the association of a company and a part carries an attribute named Price.

The representation of such a relationship using a table is straightforward. Just create a table that is ID dependent on both of its parents and place the Price attribute in that table. The result for the example in Figure 6-12(a) is the table:

QUOTATION (*CompanyName*, *PartNumber*, Price)

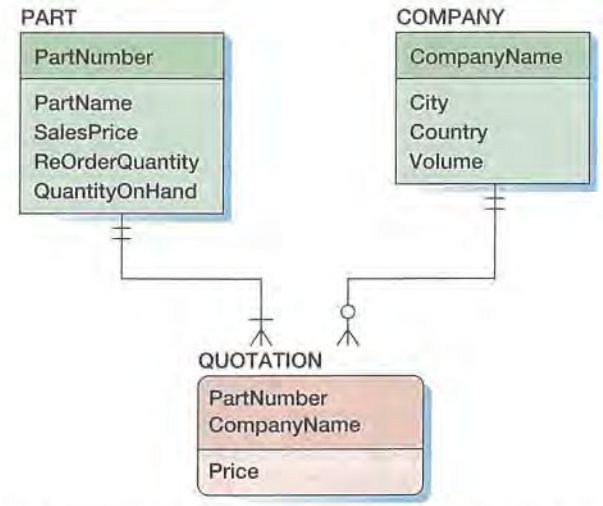
This table appears in the database design in Figure 6-12(b). Like all ID-dependent relationships, the parents of an association table are required. The parents may or may not require the

Figure 6-11

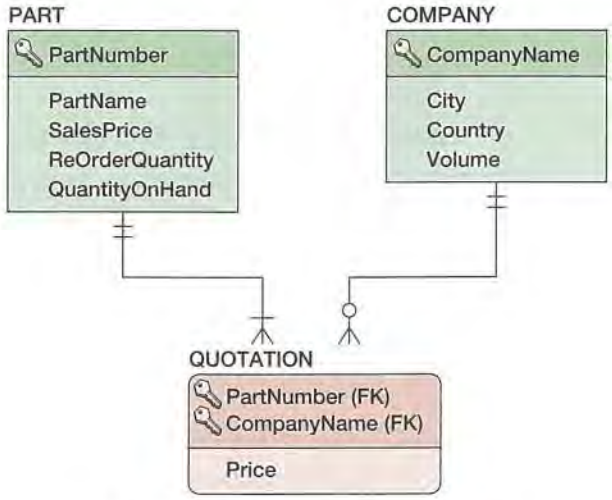
Four Uses for ID-Dependent Entities

- Representing N:M Relationships
- Association Relationships
- Multivalued Attributes
- Archetype/Instance Relationships

Amazon Ex. 1008.0228



(a) Association Pattern Data Model from Figure 5-22



(b) Association Pattern Database Design

Figure 6-12
Using ID-Dependent Entities in an Association Relationship

rows of the association table, depending on application requirements. In Figure 6-12(b), a COMPANY need not have any QUOTATION rows, but a PART must have at least one QUOTATION row.

BY THE WAY The table that represents the association entity looks very much like an intersection table; the only difference is the presence of the Price attribute. Because of the attribute, the need for association tables, such as QUOTATION, will appear in user requirements. Somewhere there will be a form or a report that has the attribute Price. However, the need for intersection tables never appears in the users' world. Such tables are an artifact of the relational model, and no form, report, or other user requirement will indicate the need for one.

Intersection tables complicate the construction of applications. They must be processed to obtain related rows, but they never directly appear on a form or report. In Access, they are frustratingly difficult to mangle into the form and report design tools. You will see more about this in later chapters. In any case, for now understand the key difference between association and intersection tables: Association tables have user data, but intersection tables do not.

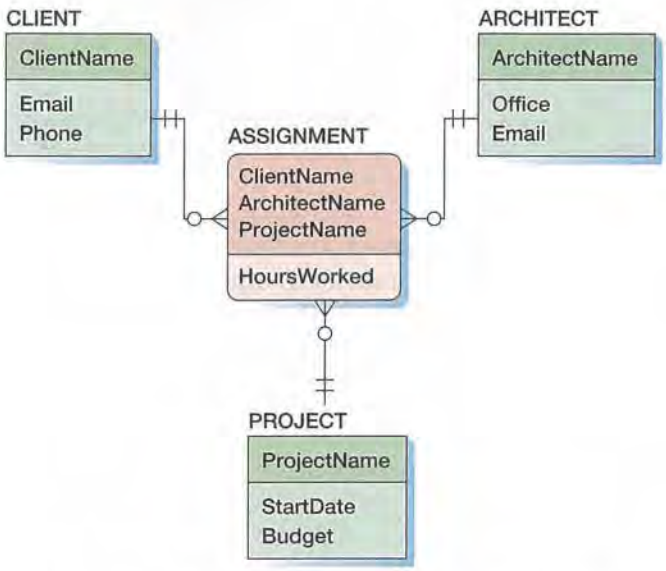
Amazon Ex. 1008.0229

As shown in Figure 6-13, association entities sometimes connect more than two entity types. Figure 6-13(a) shows the association relationship among the CLIENT, ARCHITECT, and PROJECT entities from the data model we created in Figure 5-23. When there are several participants in the association, the strategy just shown is simply extended. The association table will have the key of each of its parents, as shown in Figure 6-13(b). In this case, the ASSIGNMENT table has three foreign keys and one nonkey attribute, HoursWorked.

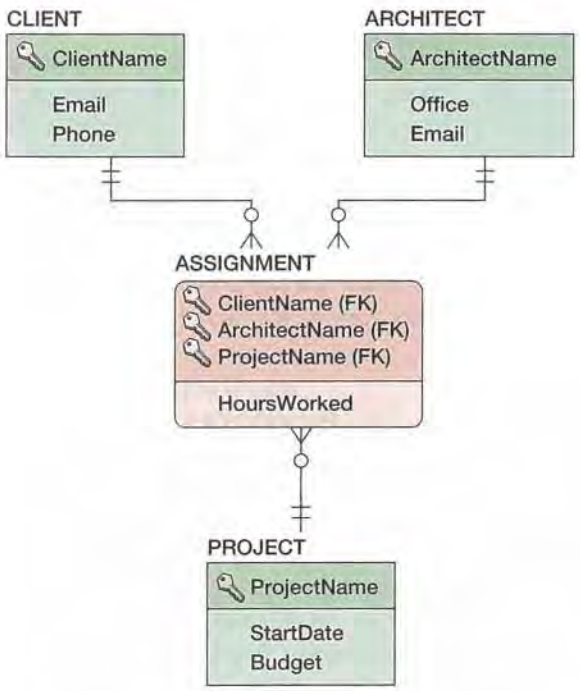
In both of these examples, it is only coincidence that the association tables have only one nonkey attribute. In general, an association table can have as many nonkey attributes as necessary to meet user requirements.

Figure 6-13

Using ID-Dependent Entities in an Association Relationship Among Three Entities

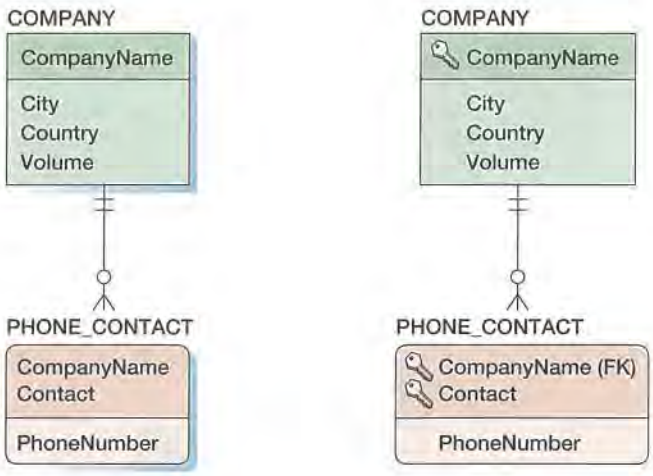


(a) Association Pattern Data Model from Figure 5-23



(b) Association Pattern Database Design

Amazon Ex. 1008.0230



(a) Data Model with Multivalued Attributes from Figure 5-29

(b) Database Design to Store Multivalued Attributes

Figure 6-14
Using ID-Dependent Entities to Store Multivalued Attributes

Multivalued Attributes

The third use for ID-dependent entities is to represent multivalued entity attributes, and this is illustrated in Figure 6-14. Figure 6-14(a) is a copy of Figure 5-29. Here, COMPANY has a multivalued composite, (Contact, PhoneNumber), that is represented by the ID-dependent entity PHONE_CONTACT.

As shown in Figure 6-14(b), representing the PHONE_CONTACT entity is straightforward. Just replace it with a table and replace each of its attributes with a column. In this example, the CompanyName attribute is both a part of the primary key and a foreign key.

Like all ID-dependent tables, PHONE_CONTACT must have a parent row in COMPANY. However, a COMPANY row may or may not have a required PHONE_CONTACT, depending on application requirements.

BY THE WAY As you can see from these examples, it is not much work to transform an ID-dependent entity into a table. All that is necessary is to transform the entity into a table, and copy the attributes into columns.

Why is it so simple? There are two reasons. First, all identifying relationships are 1:N. If they were 1:1, there would be no need for the ID-dependent relationship. The attributes could just be placed in the parent entity. Given that the relationship is 1:N, the design principle is to place the key of the parent into the child. However, the definition of an ID-dependent relationship is that part of its identifier is an identifier of its parent. Thus, by definition, the key of the parent is already in the child. Hence, it is not necessary to create a foreign key; that work has already been done during data modeling.

Archetype/Instance Pattern

As illustrated in Figure 6-15, the fourth use for ID-dependent entities and identifying relationships is the archetype/instance pattern. Figure 6-15(a), which is a copy of Figure 5-30, shows the CLASS/SECTION archetype/instance example from Chapter 5, and Figure 6-15(b) shows the relational design.

As noted in the last chapter, however, sometimes the instances of an archetype/instance pattern are given identifiers of their own. In that case, the instance entity becomes a weak, but not ID-dependent, entity. When this occurs, the relationship must be transformed using the rules of a 1:N relationship between a strong entity and a weak but non-ID-dependent entity. However, this transformation is the same as a 1:N relationship between two strong entities. This just means that the primary key of the parent table should be placed in the child table as a foreign key.

Amazon Ex. 1008.0231

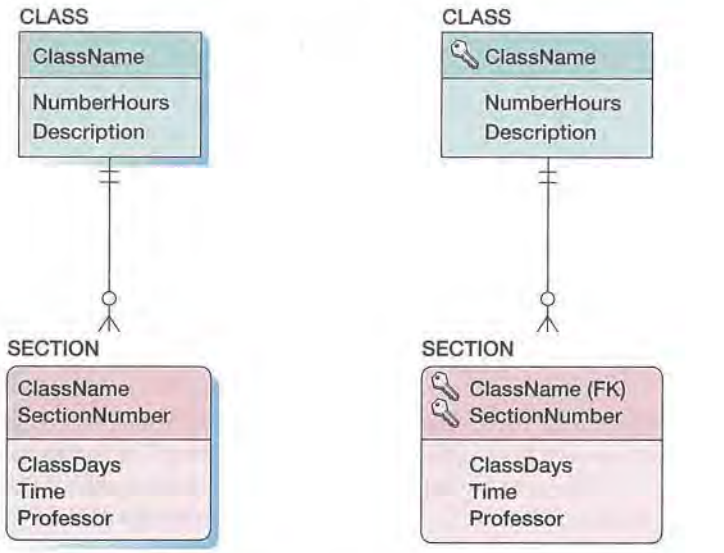


Figure 6-15
Using ID-Dependent Entities in an Archetype/Instance Pattern

(a) Data Model with Archetype/Instance Pattern from Figure 5-30

(b) Database Design for Archetype/Instance Pattern

Figure 6-16(a) shows a copy of the data model in Figure 5-31 in which SECTION has been given the identifier ReferenceNumber. In the relational database design in Figure 6-16(b), ClassName (the primary key of the parent CLASS table) has been placed in SECTION (the child table) as a foreign key.

Keep in mind, however, that even though SECTION is no longer ID-dependent, it is still weak. SECTION requires a CLASS for its existence. This means that a SECTION must always have a CLASS as its parent, and this restriction arises from logical necessity, not just from application requirements. The fact that SECTION is weak should be recorded in design documentation.

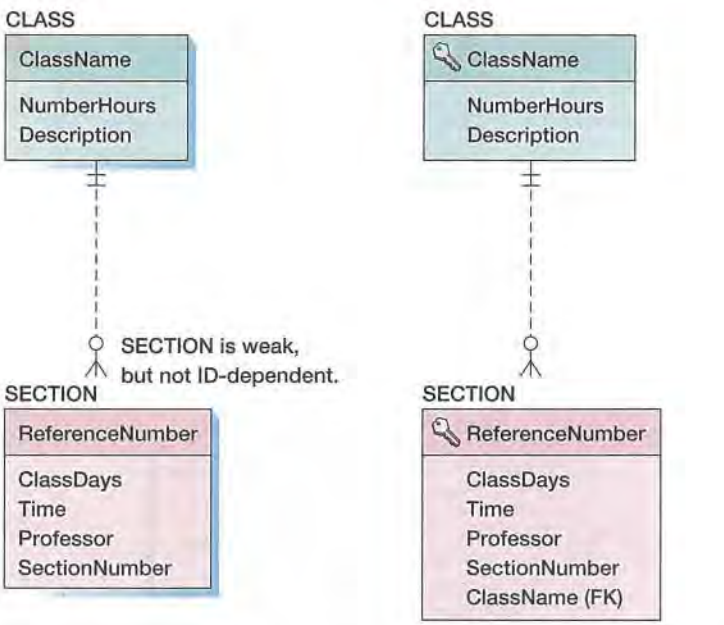


Figure 6-16
Transformation of the Archetype/Instance Pattern Using Non-ID-Dependent Weak Entities

(a) Data Model with Non-ID-Dependent Weak Entity from Figure 5-31

(b) Database Design for Non-ID-Dependent Weak Entity

Amazon Ex. 1008.0232

Relationships with a Weak Non-ID-Dependent Entity

As you learned in Chapter 5, a relationship between a strong entity and a weak but non-ID-dependent entity behaves just the same as a relationship between two strong entities. The relationship is a nonidentifying relationship, and, again, these relationships are characterized by their maximum cardinality. The previous discussion of 1:1, 1:N, and N:M relationships between strong entities also applies to these types of relationships between a strong entity and a weak but non-ID-dependent entity.

For example, what happens when the identifier of the parent of an ID-dependent entity is replaced with a surrogate key? Consider the example of BUILDING and APARTMENT, in which the identifier of APARTMENT is the composite of an apartment number and a building identifier.

Suppose that the identifier of BUILDING is (Street, City, State/Province, Country). In this case, the identifier of APARTMENT is (Street, City, State/Province, Country, ApartmentNumber). This design can be improved by replacing the long BUILDING identifier with a surrogate key. Suppose that we replace the key of BUILDING with BuildingID, a surrogate.

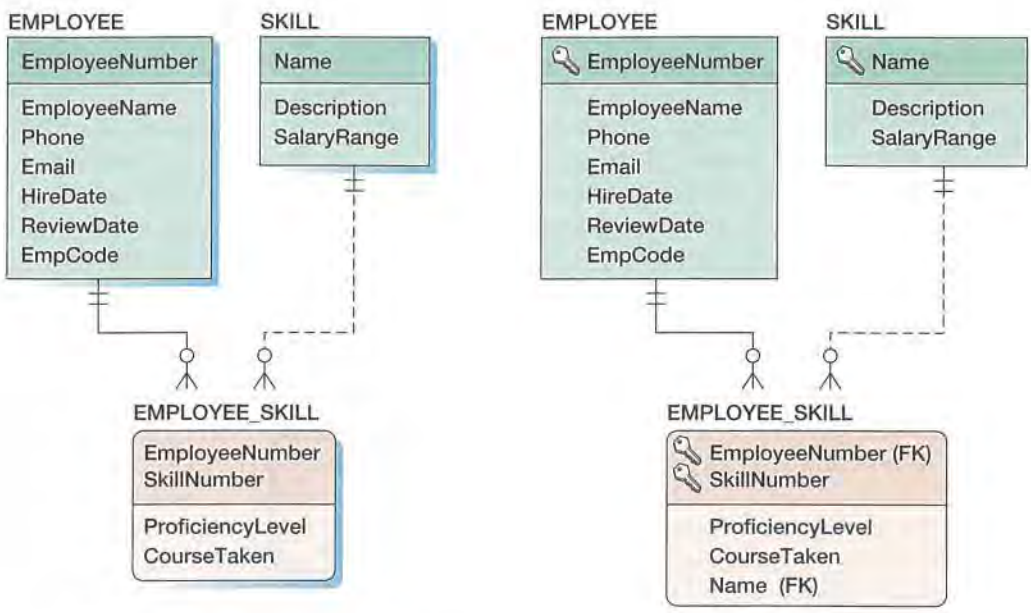
Now, with a surrogate key for BUILDING, what is the key of APARTMENT? When we place the key of the parent in the child, we obtain (BuildingID, ApartmentNumber). But this combination has no meaning to the user. What does an identifier of (10045898, '5C') mean to a user? Nothing. The key became meaningless when Street, City, State/Province, and Country were replaced by BuildingID in BUILDING.

We can improve the design by using the following principle: When replacing the identifier of the parent of an ID-dependent entity with a surrogate key, replace the identifier of the ID-dependent entity with its own surrogate key. The resulting table will be weak, but not ID dependent.

Relationships in Mixed Entity Designs

As you might guess, the design of mixed entity patterns is a combination of strong entities and ID-dependent entities designs. Consider the example of employees and skills in Figure 6-17. Figure 6-17(a) is a copy of Figure 5-35. Here, the entity EMPLOYEE_SKILL is ID dependent on EMPLOYEE, but it has a nonidentifying relationship to SKILL.

Figure 6-17 Transformation of the Mixed Entity Pattern



(a) Data Model with Mixed Entity Pattern from Figure 5-35

(b) Database Design for Mixed Entity Pattern

Amazon Ex. 1008.0233

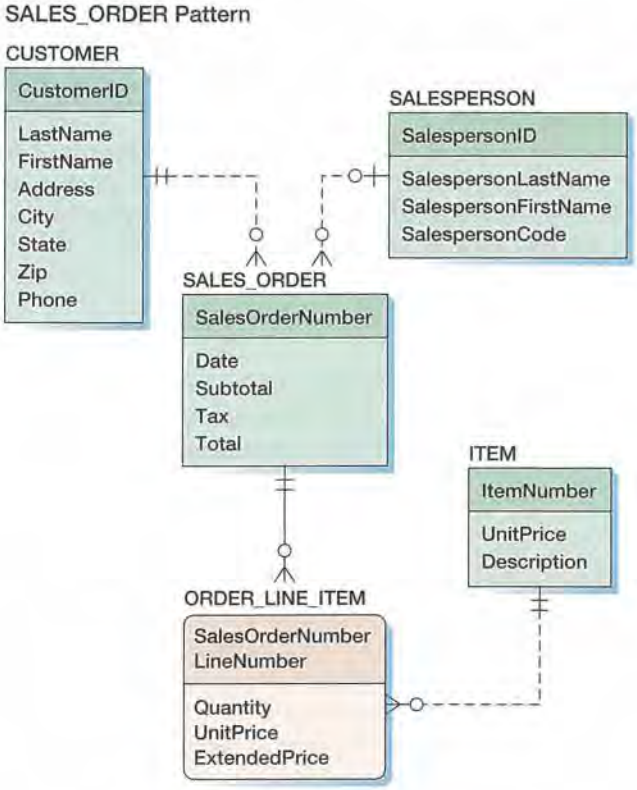
The database design of the E-R model for the data model in Figure 6-17(a) is shown in Figure 6-17(b). Notice that EmployeeNumber is both a part of the primary key of EMPLOYEE_SKILL and also a foreign key to EMPLOYEE. The 1:N nonidentifying relationship between SKILL and EMPLOYEE_SKILL is represented by placing the key of SKILL, which is Name, in EMPLOYEE_SKILL. Note that EMPLOYEE_SKILL.Name is a foreign key but not part of the primary key of EMPLOYEE_SKILL.

A similar strategy is used to transform the SALES_ORDER data model in Figure 6-18. Figure 6-19(a) is a copy of the SALES_ORDER data model originally shown in Figure 5-33. Here, the ID-dependent table, ORDER_LINE_ITEM, has SalesOrderNumber as part of its primary key and as a foreign key. It has ItemNumber as a foreign key only.

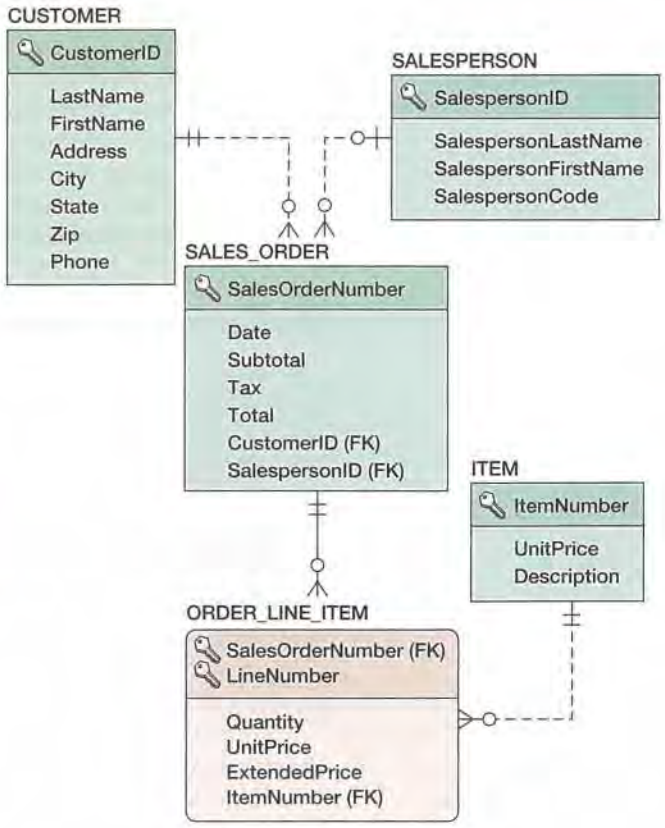
BY THE WAY The design transformation process for all HAS-A relationships can be summarized by the phrase, "Place the primary key of the parent in the child as a foreign key." For strong entities, a 1:1 relationship can have either entity as the parent, hence the foreign key can go in either table. For 1:N relationships, the primary key of the parent goes in the child as the foreign key. For N:M relationships, decompose the model into two 1:N relationships by defining an intersection table and place the parent key of the parent in the child as a foreign key for each.

For identifying relationships, the primary key of the parent is already in the child, so there is nothing more to do. For mixed relationships, on the identifying side, the primary key of the parent is already in the child. On the nonidentifying side, place the primary key of the parent in the child. In short, if you're going to memorize just a few rules for creating relationships, the first one is "HAS-A: Place the primary key of the parent in the child as the foreign key."

Figure 6-18
Transformation of the SALES_ORDER Pattern

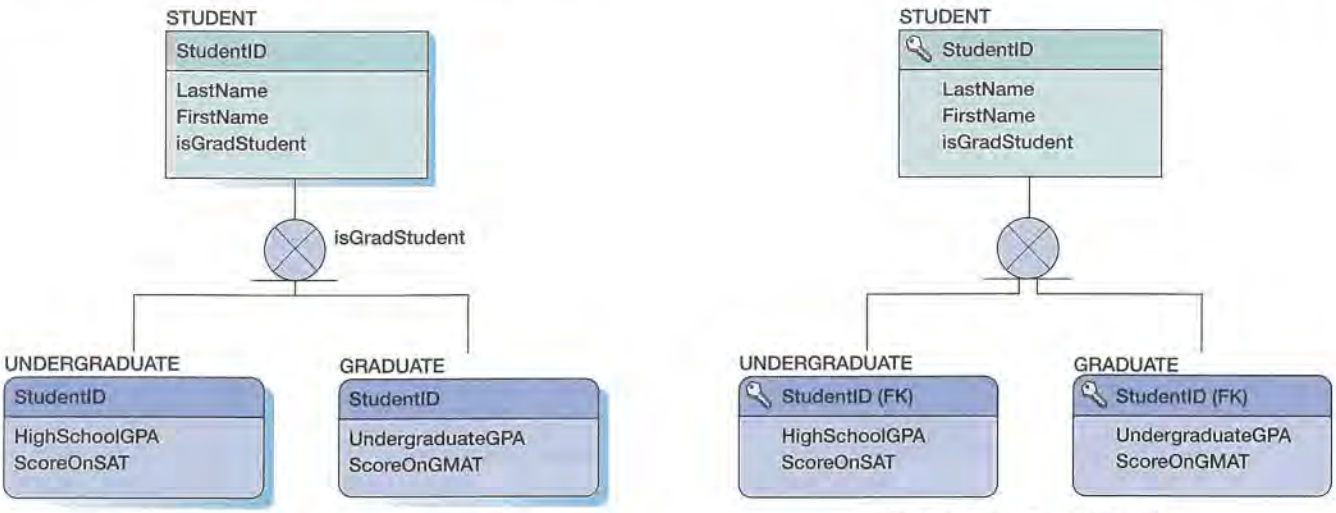


(a) Data Model of SALES_ORDER Pattern from Figure 5-33



(b) Database Design for the SALES_ORDER Pattern

Amazon Ex. 1008.0234



(a) Data Model of the Supertype/Subtype Relationship from Figure 5-20(a)

(b) Database Design for the Supertype/Subtype Relationship

Figure 6-19

Transformation of the Supertype/Subtype Entities

Relationships Between Supertype and Subtype Entities

Representing relationships between supertype entities and their subtypes is easy. Recall that these relationships are also called IS-A relationships because a subtype and its supertype are representations of the same underlying entity. A MANAGER (subtype) is an EMPLOYEE (supertype), and a SALESCLERK (subtype) is also an EMPLOYEE (supertype). Because of this equivalence, the keys of all subtype tables are identical to the key of the supertype table.

Figure 6-19(a) shows the data model in Figure 5-13(a), an example for two subtypes of STUDENT. Notice that the key of STUDENT is StudentID and that the key of each of the subtypes also is StudentID. UNDERGRADUATE.StudentID and GRANDUATE.StudentID are both primary keys and foreign keys to their supertype.

Discriminator attributes cannot be represented in relational designs. In Figure 6-19(b), we can do nothing with isGradStudent except note in the design documentation that isGradStudent determines subtype. Application programs will need to be written to use isGradStudent to determine which subtype pertains to a given STUDENT.

Recursive Relationships

The representation of recursive relationships is just an extension of the techniques used for representing relationships between strong entities. These techniques may be a bit difficult to comprehend at first because they appear strange, but they involve principles that you have already learned.

1:1 Recursive Relationships

Consider the 1:1 recursive BOXCAR relationship in Figure 6-20(a), which is the same data model we developed in Figure 5-38. To represent the relationship, we create a foreign key in BOXCAR that contains the identifier of the boxcar ahead, as shown in Figure 6-20(b). Because the relationship is 1:1, we make the foreign key unique by defining it as unique (shown here as an alternate key). This restriction enforces the fact that a boxcar can have at most one boxcar in front of it.

Notice that both sides of the relationship are optional. This occurs because the last car on the train is ahead of no other car, and because the first car on the train has no other car ahead of it. If the data structure were circular, this restriction would not be necessary. For example, if you wanted to represent the sequence of names of the calendar months, and you wanted December to lead to January, then you could have a 1:1 recursive structure with required children.

Amazon Ex. 1008.0235

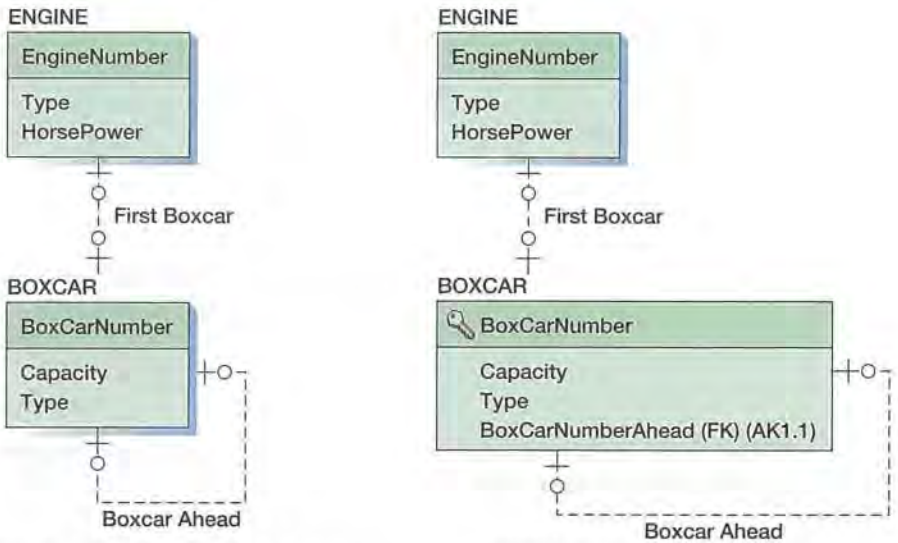


Figure 6-20 Representing 1:1 Recursive Relationships

(a) Data Model for a 1:1 Recursive Relationship in Figure 5-38

(b) Database Design for a 1:1 Recursive Relationship

BY THE WAY If you find the concept of recursive relationships confusing, try this trick. Assume that you have two entities: BOXCAR_AHEAD and BOXCAR_BEHIND, each having the same attributes. Notice that there is a 1:1 relationship between these two entities. Replace each entity with its table. Like all 1:1 strong entity relationships, you can place the key of either table as a foreign key in the other table. For now, place the key of BOXCAR_AHEAD into BOXCAR_BEHIND.

Now realize that BOXCAR_AHEAD only duplicates data that reside in BOXCAR_BEHIND. The data are unnecessary. So, discard BOXCAR_AHEAD and you will have the same design as shown in Figure 6-20(b).

1:N Recursive Relationships

As with all 1:N relationships, 1:N recursive relationships are represented by placing the primary key of the parent in the child as a foreign key. Consider the Manages relationship in Figure 6-21(a), which is the data model we developed in Figure 5-41. In this case, we place the name of the manager in each employee's row. Thus, in Figure 6-21(b), the EmployeeNameMgr has been added to the EMPLOYEE table.

Notice that both the parent and the child are optional. This is true because the lowest-level employees manage no one and because the highest-level person, the CEO or other most senior person, has no manager. If the data structure were circular, this would not be the case.

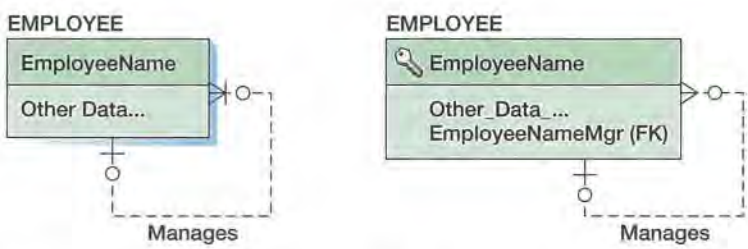
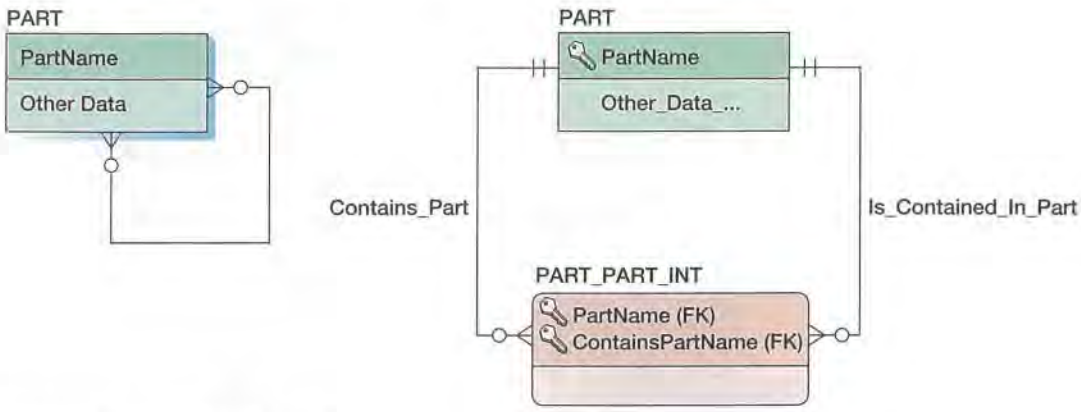


Figure 6-21 Representing 1:N Recursive Relationships

(a) Data Model for a 1:N Recursive Relationship in Figure 5-41

(b) Database Design for a 1:N Recursive Relationship

Amazon Ex. 1008.0236



(a) Database Design for an N:M Recursive Relationship

(b) Table Representation

Figure 6-22
Representing N:M Recursive Relationships

N:M Recursive Relationships

The trick for representing N:M recursive relationships is to decompose the N:M relationship into two 1:N relationships. We do this by creating an intersection table, just as we did for N:M relationships between strong entities.

Figure 6-22(a) is the data model we developed in Figure 5-43. It shows the solution to an example for a bill-of-materials problem. Each part has potentially many subordinate parts, and each part can be used as a component in potentially many other parts. To represent this relationship, create an intersection table that shows the correspondence of a part/part use. You can model upwards or downwards. If the former, the intersection table will carry the correspondence of a part and where that part is used. If the latter, the intersection table will carry the correspondence of a part and the parts that it contains. Figure 6-22(b) shows the intersection table for modeling downwards in the bill of materials.

BY THE WAY Again, if you find this to be confusing, assume that you have two different tables, one called PART and a second called CONTAINED_PART. Create the intersection table between the two tables. Note that CONTAINED_PART duplicates the attributes in PART, and is thus unnecessary. Eliminate the table and you will have the design in Figure 6-22(b).

Representing Ternary and Higher-Order Relationships

As we discussed in Chapter 5, ternary and higher-order relationships can be represented by multiple binary relationships. Most of the time, such a representation works without problem. However, in some cases, there are constraints that add complexity to the situation. For example, consider the ternary relationship among the entities ORDER, CUSTOMER, and SALESPERSON. Assume that the relationship from CUSTOMER to ORDER is 1:N and that the relationship from SALESPERSON to ORDER also is 1:N. We can represent the three-part relationship among ORDER:CUSTOMER:SALESPERSON as two separate binary relationships, one between ORDER and CUSTOMER and a second between SALESPERSON and CUSTOMER. The design of the tables will be:

CUSTOMER (CustomerNumber, {nonkey data attributes})

SALESPERSON (SalespersonNumber, {nonkey data attributes})

ORDER (OrderNumber, {nonkey data attributes}, *CustomerNumber*, *SalespersonNumber*)

Amazon Ex. 1008.0237

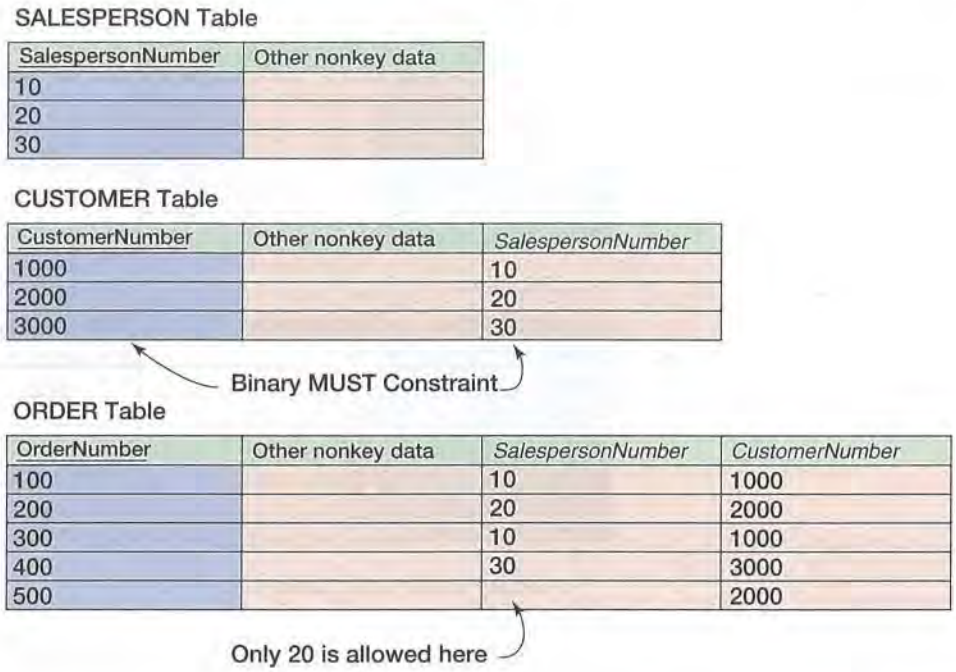


Figure 6-23
Ternary Relationship with a MUST Constraint

Suppose, however, that the business has a rule that each CUSTOMER can place orders only with a particular SALESPERSON. In this case, the ternary relationship ORDER:CUSTOMER:SALESPERSON is constrained by an additional binary 1:N relationship between SALESPERSON and CUSTOMER. To represent the constraint, we need to add the key of SALESPERSON to CUSTOMER. The three relations will now be:

CUSTOMER (CustomerNumber, {nonkey data attributes}, *SalespersonNumber*)

SALESPERSON (SalespersonNumber, {nonkey data attributes})

ORDER (OrderNumber, {nonkey data attributes}, *CustomerNumber*, *SalespersonNumber*)

The constraint that a particular CUSTOMER is sold to by a particular SALESPERSON means that only certain combinations of CustomerNumber and SalespersonNumber can exist together in ORDER. Unfortunately, this constraint cannot be expressed in a relational model. It must be documented in the design, however, and enforced by program code, as shown in Figure 6-23.

A constraint that requires one entity to be combined with another entity is called a **MUST constraint**. Other similar constraints are the MUST NOT constraint and the MUST COVER constraint. In a **MUST NOT constraint**, the binary relationship indicates combinations that are not allowed to occur in the ternary relationship. For example, the ternary relationship PRESCRIPTION:DRUG:CUSTOMER shown in Figure 6-24 can be constrained by a binary relationship in the ALLERGY table that lists the drugs that a customer is not allowed to take.

In a **MUST COVER constraint**, the binary relationship indicates all combinations that must appear in the ternary relationship. For example, consider the relationship AUTO:REPAIR:TASK in Figure 6-25. Suppose that a given REPAIR consists of a number of TASKs, all of which must be performed for the REPAIR to be successful. In this case, in the relation AUTO-REPAIR, when a given AUTO has a given REPAIR, then all of the TASKs for that REPAIR must appear as rows in that relation.

None of the three types of binary constraints discussed here can be represented in the relational design. Instead, they are documented in the design and implemented in application code.

Amazon Ex. 1008.0238

Figure 6-24

Ternary Relationship with a MUST NOT Constraint

DRUG Table

DrugNumber	Other nonkey data
10	
20	
30	
45	
70	
90	

ALLERGY Table

CustomerNumber	DrugNumber	Other nonkey data
1000	10	
1000	20	
2000	20	
2000	45	
3000	30	
3000	45	
3000	70	

Binary MUST NOT Constraint

PRESCRIPTION Table

PrescriptionNumber	Other nonkey data	*DrugNumber*	*CustomerNumber*
100		45	1000
200		10	2000
300		70	1000
400		20	3000
500			2000

Neither 20 nor 45 can appear here

Figure 6-25

Ternary Relationship with a MUST COVER Constraint

REPAIR Table

RepairNumber	Other nonkey data
10	
20	
30	
40	

TASK Table

TaskNumber	Other nonkey data	*RepairNumber*
1001		10
1002		10
1003		10
2001		20
2002		20
3001		30
4001		40

Binary MUST COVER Constraint

AUTO-REPAIR Table

InvoiceNumber	RepairNumber	TaskNumber	Other nonkey data
100	10	1001	
200	10	1002	
300	10	1003	
400	20	2001	
500	20		

2002 must appear here

Amazon Ex. 1008.0239

Relational Representation of the Highline University Data Model

Let's consider the data model we created for Highline University in Chapter 5. Our final data model for Highline University is shown in Figure 6-26.

Using the principles we've discussed in this chapter, we can turn this into a relational database design, and the resulting database design is a straightforward application of the principles described in this chapter. The database design for Highline University is shown in Figure 6-27.

Review Figure 6-27 to ensure that you understand the representation of every relationship. Note that there are actually two foreign key references to a DepartmentName primary key column in STUDENT. The first is DepartmentName (FK), which is the foreign key linking to the DepartmentName primary key in DEPARTMENT. This relationship has the referential integrity constraint:

DepartmentName in STUDENT must exist in DepartmentName in DEPARTMENT

The second is ProfessorDepartment (FK), which is part of the composite foreign key (ProfessorDepartment, ProfessorFirstName, ProfessorLastName). This foreign key links to the primary key (DepartmentName, ProfessorFirstName, ProfessorLastName) in APPOINTMENT, and has the referential integrity constraint:

(ProfessorDepartment, ProfessorFirstName, ProfessorLastName) in STUDENT must exist in (DepartmentName, ProfessorFirstName, ProfessorLastName) in APPOINTMENT

Note that we had to change DepartmentName in APPOINTMENT to ProfessorDepartment in STUDENT because we cannot have two columns named DepartmentName in STUDENT and we had already used DepartmentName as the foreign key linking to DEPARTMENT.

This illustrates that a foreign key does not have to have the same name as the primary key it links to. As long as the referential integrity constraints are correctly specified, the foreign key name can be whatever we want it to be.

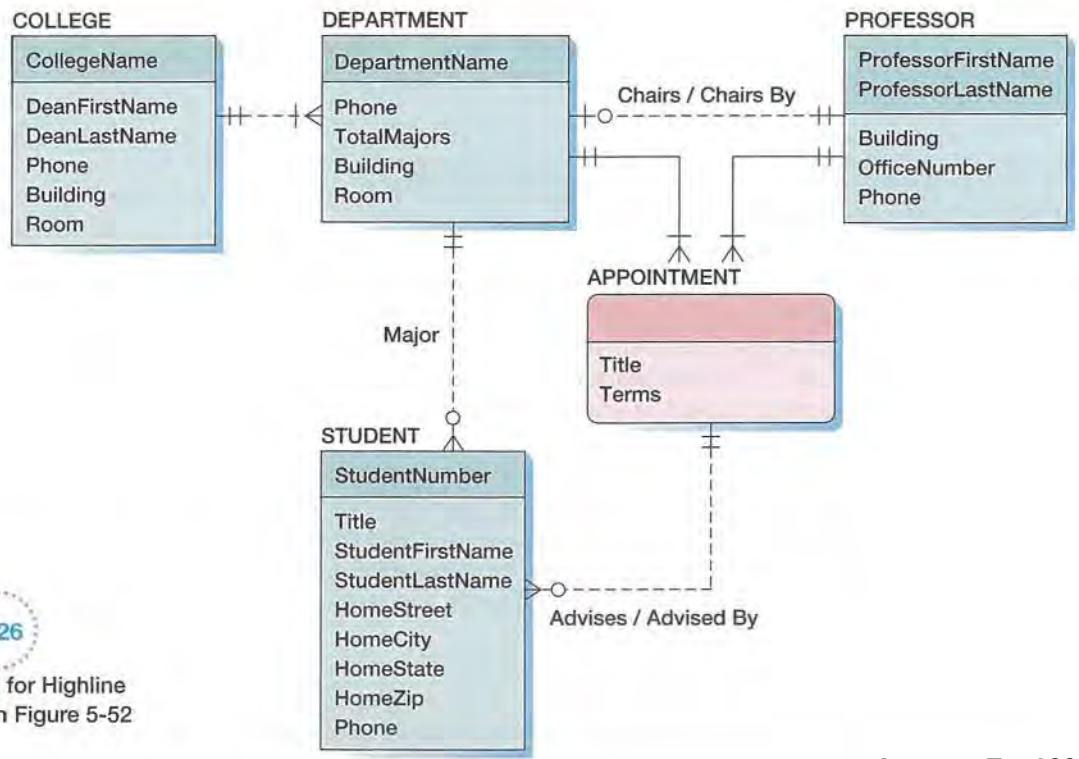


Figure 6-26
Data Model for Highline University in Figure 5-52

Amazon Ex. 1008.0240

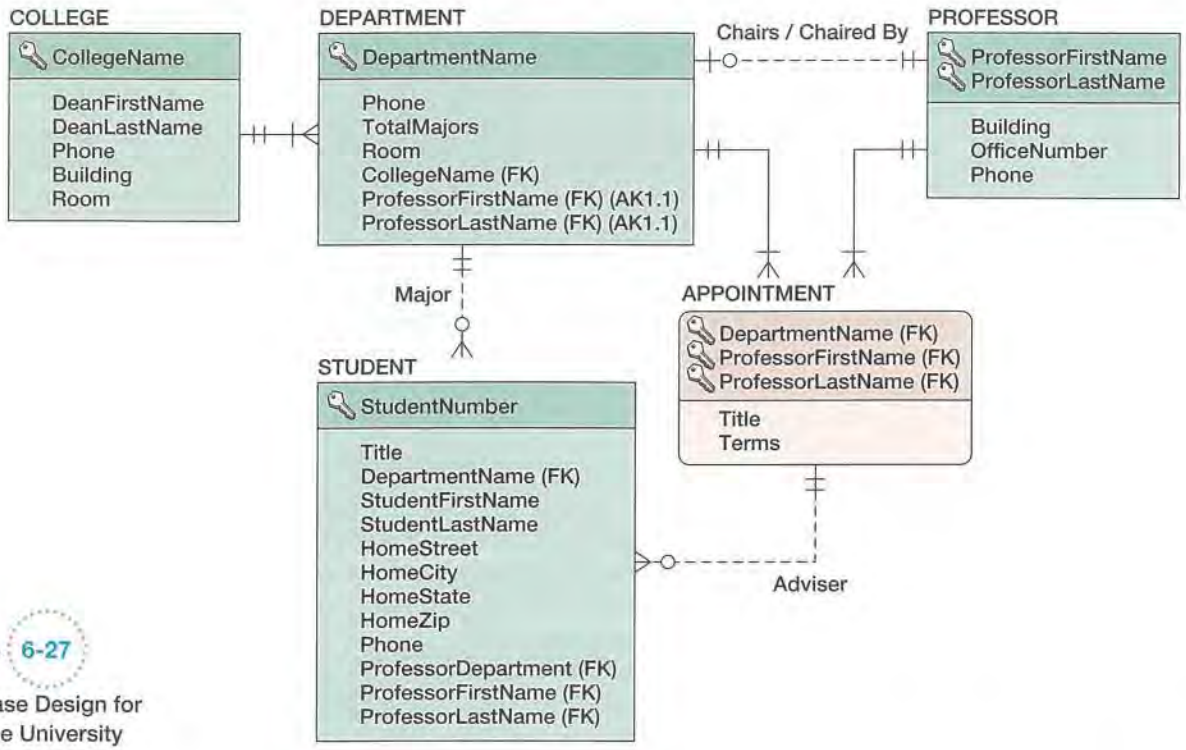


Figure 6-27
Database Design for Highline University

In addition to the two referential integrity constraints above our database design, we will also have the following:

CollegeName in DEPARTMENT must exist in CollegeName in COLLEGE

(ProfessorFirstName, ProfessorLastName) in DEPARTMENT must exist in (Professor FirstName, ProfessorLastName) in PROFESSOR

DepartmentName in APPOINTMENT must exist in DepartmentName in DEPARTMENT

(ProfessorFirstName, ProfessorLastName) in APPOINTMENT must exist in (Professor FirstName, ProfessorLastName) in PROFESSOR

Design for Minimum Cardinality

The third and last step of transforming data models into database designs is to create a plan for enforcing minimum cardinality. Unfortunately, this step can be considerably more complicated than the first two design steps. Relationships that have required children entities are particularly problematic because we cannot enforce such constraints with database structures. Instead, as you will see, we must design procedures for execution by the DBMS or by applications.

Relationships can have one of four minimum cardinalities: **parent optional and child optional (O-O)**, **parent mandatory and child optional (M-O)**, **parent optional and child mandatory (O-M)**, or **parent mandatory and child mandatory (M-M)**. As far as enforcing minimum cardinality is concerned, no action needs to be taken for O-O relationships, and we need not consider them further. The remaining three relationships pose restrictions on insert, update, and delete activities.

Figure 6-28 summarizes the actions needed to enforce minimum cardinality. Figure 6-28(a) shows needed actions when the parent row is required (M-O and M-M relationships), and Figure 6-28(b) shows needed actions when the child row is required (O-M and M-M relationships). In

Amazon Ex. 1008.0241

Parent Required	Action on Parent	Action on Child
Insert	None.	Get a parent. Prohibit.
Modify key or foreign key	Change children's foreign key values to match new value (**cascade update**). Prohibit.	OK, if new foreign key value matches existing parent. Prohibit.
Delete	Delete children (**cascade delete**). Prohibit.	None.

(a) Actions When the Parent Is Required

Child Required	Action on Parent	Action on Child
Insert	Get a child. Prohibit.	None.
Modify key or foreign key	Update the foreign key of (at least one) child. Prohibit.	If not last child, OK. If last child, prohibit or find a replacement.
Delete	None.	If not last child, OK. If last child, prohibit or find a replacement.

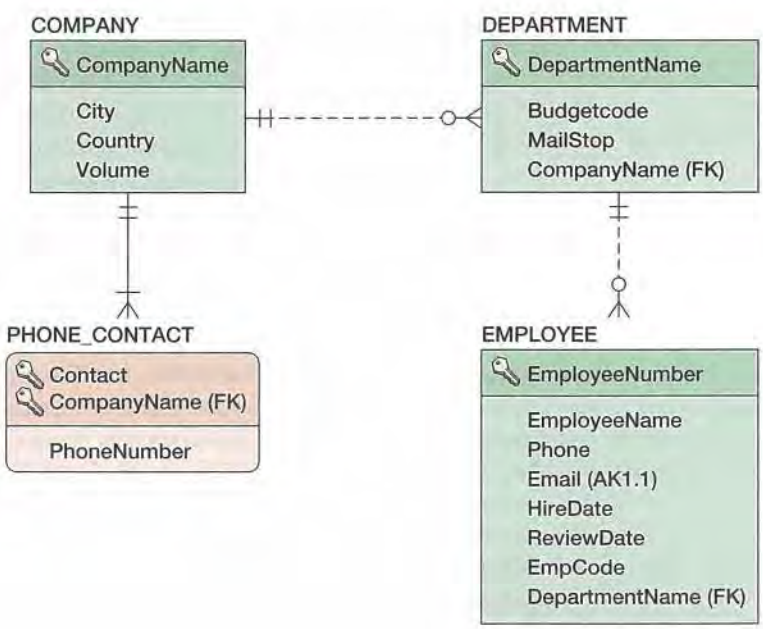
Figure 6-28
Summary of Actions to Enforce Minimum Cardinality

(b) Actions When the Child Is Required

these figures and the accompanying discussion, the term **action** means **minimum cardinality enforcement action**. We use the shorter term *action* for ease of discussion.

To discuss these rules, we will use the database design for storing data on several companies shown in Figure 6-29. In this diagram, we have a 1:N M-O relationship between COMPANY and DEPARTMENT and between DEPARTMENT and EMPLOYEE, and a 1:N M-M relationship between COMPANY and PHONE_CONTACT. In the COMPANY-to-DEPARTMENT relationship, COMPANY (on the 1 side of the relationship) is the parent entity and DEPARTMENT (on the N side of the relationship) is the child entity. In the DEPARMENT-to-EMPLOYEE relationship, DEPARTMENT (on the 1 side of the relationship) is the parent entity and EMPLOYEE (on the N

Figure 6-29
Database Design for Data on Several Companies



Amazon Ex. 1008.0242

side of the relationship) is the child entity. In the COMPANY-to-PHONE_CONTACT relationship, COMPANY (on the 1 side of the relationship) is the parent entity and PHONE_CONTACT (on the N side of the relationship) is the child entity.

Actions When the Parent Is Required

When the parent is required, we need to ensure that every row of the child table has a valid, non-null value of the foreign key. To accomplish this, we must restrict actions to update or delete the parent's primary key and actions to create or modify the child's foreign key. Consider actions on the parent first.

Actions on the Parent Row When the Parent Is Required

According to Figure 6-28(a), when a new parent is created, nothing needs to be done. No child row can yet be dependent upon the new row. In our example, we can create a new DEPARTMENT and not worry about minimum cardinality enforcement.

However, consider what happens if we attempt to change the value of an existing parent row's primary key. If that row has children, then those children have a foreign key value that matches the current primary key value. If the primary key of the parent changes, then any existing children will become orphans; their foreign key values will no longer match a parent row. To prevent the creation of orphans, either the foreign key values must be changed to match the new value of the parent's primary key or the modification to the parent's primary key must be prohibited.

In our example, if a DEPARTMENT attempts to change its DepartmentName from 'Info Sys' to 'Information Systems', then any child rows that have a foreign key value of 'Info Sys' will no longer match a parent and will be orphans. To prevent orphans, either the values of the foreign key in EMPLOYEE must also be changed to 'Information Systems' or the update to the primary key in DEPARTMENT must be prohibited. The policy of propagating a change from the parent's primary key to the children's foreign key is called **cascading updates**.

Now consider what happens when there is an attempt to delete a parent. If that row has children, and if the deletion is allowed, then the children will become orphans. Hence, when such a delete attempt is made, either the children must be deleted as well or the deletion must be prohibited. Deleting the children along with the parent is called **cascading deletions**. In our example, when an attempt is made to delete a DEPARTMENT, either all related rows in EMPLOYEE must be deleted as well or the deletion must be disallowed.

BY THE WAY Generally, cascading deletions are not chosen for relationships between strong entities. The deletion of a DEPARTMENT row should not force the deletion of EMPLOYEE rows. Instead, the deletion should be disallowed. To remove a DEPARTMENT row, the EMPLOYEE rows would be reassigned to a new DEPARTMENT and then the DEPARTMENT row would be deleted.

However, cascading deletions are almost always chosen for weak child entities. For example, when you delete a COMPANY, you should always delete all of the weak PHONE_NUMBER rows that depend on that COMPANY.

Actions on the Child Row When the Parent Is Required

Now consider actions on the child row. If the parent is required, then when a new child row is created, the new row must have a valid foreign key value. When we create a new EMPLOYEE, for example, if DEPARTMENT is required, then the new EMPLOYEE row must have a valid value for DepartmentName. If not, the insert must be disallowed. Usually there is a default policy for assigning parents to a new row. In our example, when a new row is added to EMPLOYEE, the default policy could be to add the new employee to the department named 'Human Resources'.

Amazon Ex. 1008.0243

With regards to modifications to the foreign key, the new value must match a value of the primary key in the parent. In EMPLOYEE, if we change DepartmentName from 'Accounting' to 'Finance', then there must already be a DEPARTMENT row with the primary key value of 'Finance'. If not, the modification must be prohibited.

If the parent row is required, there are no restrictions on the deletion of the child row. The child can go away without consequence on the parent.

BY THE WAY When the parent has a surrogate key, the enforcement actions for update are different between the parent and the child. On the parent side, the surrogate key will never change, and hence update actions can be ignored. On the child side, however, the foreign key can change if the child switches to a new parent. Hence, on the parent side, you can ignore actions when the key is a surrogate. On the child side, however, you must consider update actions even when the parent's key is a surrogate.

Actions When the Child Is Required

When the child is required, we need to ensure that there is at least one child row for the parent at all times. The last child cannot leave the parent. For example, in the DEPARTMENT-to-EMPLOYEE relationship, if a DEPARTMENT requires an EMPLOYEE, then the last EMPLOYEE cannot leave the DEPARTMENT. This has ramifications on actions on the child, as shown in Figure 6-28(b).

Enforcing required children is much more difficult than enforcing required parents. To enforce a required parent, we just need to check for a match between primary key and foreign key values. To enforce a required child, we must count the number of children that a parent has. This difference forces us to write code to enforce required children. To begin, consider the required child actions from the perspective of the parent.

Actions on the Parent Row When the Child Is Required

If the child is required, then we cannot create a new parent without also creating a relationship to a child. This means that we must either find an existing child row and change its foreign key to match that of the new parent or we must create a new child row at the same time the parent is created. If neither action can be taken, then the insertion of the new parent must be prohibited. These rules are summarized in the first row of Figure 6-28(b).

If the child is required, then to modify the parent's primary key, either the key of at least one child must also be changed or the update must be disallowed. This restriction never applies to parents with surrogate keys because their values never change.

Finally, if the child is required and the parent is deleted, no action need be taken. Because it is the child that is required, and not the parent, the parent can disappear without any consequence.

Actions on the Child Row When the Child Is Required

As shown in Figure 6-28(b), if the child is required, then no special action needs to be taken when inserting a new child. The child comes into existence without influencing any parent.

However, there are restrictions on updating the foreign key of a required child. In particular, if the child is the last child of its current parent, then the update cannot occur. If it were to occur, the current parent would be childless, and that is not allowed. Thus, a procedure must be written to determine the number of children of the current parent. If that number is two or greater, then the child foreign key value can be changed. Otherwise the update is prohibited.

A similar restriction pertains to the deletion of required children. If the child is the last child to the parent, then the deletion is not allowed. Otherwise, the child can be deleted without restriction.

Amazon Ex. 1008.0244

Relationship Minimum Cardinality	Action to Apply	Remarks
O-O	Nothing	
M-O	Parent-required actions [Figure 6-28(a)]	Easily enforced by DBMS; define referential integrity constraint and make foreign key NOT NULL.
O-M	Child-required actions [Figure 6-28(b)]	Difficult to enforce. Requires use of triggers or other application code.
M-M	Parent-required actions and child-required actions [Figures 6-28(a) and 6-28(b)]	Very difficult to enforce. Requires a combination of complex triggers. Triggers can lock each other out. Many problems!

Figure 6-30
Actions to Apply to Enforce Minimum Cardinality

Implementing Actions for M-O Relationships

Figure 6-30 summarizes the application of the actions in Figure 6-28 for each type of minimum cardinality. As stated earlier, O-O relationships pose no restrictions and need not be considered.

M-O relationships require that the actions in Figure 6-28(a) be enforced. We need to make sure that every child has a parent and that operations on either parent or child rows never create orphans.

Fortunately, these actions are easy to enforce using facilities available in most DBMS products. It turns out that we can enforce these actions with just two limitations. First, we need to define a referential integrity constraint that ensures that every foreign key value has a match in the parent table. Second, we make the foreign key column NOT NULL. With these two restrictions, all of the actions in Figure 6-28(a) will be enforced.

Consider the DEPARTMENT-to-EMPLOYEE example. If we define the referential integrity constraint

DepartmentName in EMPLOYEE must exist in DepartmentName in DEPARTMENT

then we know that every value of DepartmentName in EMPLOYEE will match a value in DEPARTMENT. If we then make DepartmentName required, we know that every row in EMPLOYEE will have a valid DEPARTMENT.

Almost every DBMS product has facilities for defining referential integrity constraints. You will learn how to write SQL statements for that purpose in the next chapter. In those statements, you will have the option of declaring whether updates and deletions are to cascade or are to be prohibited. Once you have defined the constraint and made the foreign key NOT NULL, the DBMS will take care of all of the actions in Figure 6-28(a) for you.

BY THE WAY Recall that in a 1:1 relationship between strong entities the key of either table can be placed in the other table. If the minimum cardinality of such a relationship is either M-O or O-M, it is generally best to place the key in the optional table. This placement will make the parent required, which is easier to enforce. With a required parent, all you have to do is define the referential integrity constraint and set the foreign key to NOT NULL. However, if you place the foreign key so that the child is required, let the work begin! You will have your hands full, as you're about to see.

Amazon Ex. 1008.0245

Implementing Actions for O-M Relationships

Unfortunately, if the child is required, the DBMS does not provide much help. There is no easy mechanism to ensure that appropriate child foreign keys exist nor is there any easy way to ensure that valid relationships stay valid when rows are inserted, updated, or deleted. You're on your own.

In most cases, required children constraints are enforced using **triggers**, which are modules of code that are invoked by the DBMS when specific events occur. Almost all DBMS products have triggers for insert, update, and delete actions. Triggers are defined for these actions on a particular table. Thus, you can create a trigger on *CUSTOMER INSERT* or a trigger on *EMPLOYEE UPDATE*, and so forth. You will learn more about triggers in the next chapter.

To see how you would use triggers to enforce required children, consider Figure 6-28(b) again. On the parent side, we need to write a trigger on insert and update on the parent row. These triggers either create the required child or they steal an existing child from another parent. If they are unable to perform one of these actions, they must cancel the insert or update.

On the child side, a child can be inserted without problem. Once a child gets a parent, however, it cannot leave that parent if it is the last or only child. Hence, we need to write update and delete triggers on the child that have the following logic: If the foreign key is null, the row has no parent, and the update or delete can proceed. If the foreign key does have a value, however, check whether the row is the last child. If so, it must either delete the parent, find a substitute child, or disallow the update or delete.

None of these actions will be automatically enforced by the DBMS. Instead, you must write code to enforce these rules. You will see generic examples of such code in the next chapter and real examples for SQL Server 2008 in Chapter 10, Oracle Database 11g in Chapter 10A, and MySQL 5.1 in Chapter 10B.

Implementing Actions for M-M Relationships

It is very difficult to enforce M-M relationships. All of the actions in both Figure 6-28(a) and Figure 6-28(b) must be enforced simultaneously. We have a needy parent and a needy child, and neither will let go of the other.

Consider, for example, what would happen if we change the relationship between DEPARTMENT and EMPLOYEE in Figure 6-29 to M-M, and the effect that would have on the creation of new rows in DEPARTMENT and EMPLOYEE. On the DEPARTMENT side, we must write an insert department trigger that tries to insert a new EMPLOYEE for the new DEPARTMENT. However, the EMPLOYEE table will have its own insert trigger. When we try to insert the new EMPLOYEE, the DBMS calls the insert employee trigger, which will prevent the insertion of an EMPLOYEE unless it has a DEPARTMENT row. But the new DEPARTMENT row does not yet exist because it is trying to create the new EMPLOYEE row, which does not exist because the new DEPARTMENT row does not yet exist, and round and round we go!

Now consider a deletion in this same M-M relationship. Suppose we want to delete a DEPARTMENT. We cannot delete a DEPARTMENT that has any EMPLOYEE children. So, before deleting the DEPARTMENT, we must first reassign (or delete) all of the employees in that department. However, when we try to reassign the last EMPLOYEE, an EMPLOYEE update trigger will be fired that will not allow the last employee to be reassigned. (The trigger is programmed to ensure that every DEPARTMENT has at least one EMPLOYEE.) We have a stalemate; the last employee cannot get out of the department, and the department cannot be deleted until all employees are gone!

This problem has several solutions, but none are particularly satisfying. In the next chapter, we will show one solution using SQL Views. That solution is complicated and requires careful programming that is difficult to test and fix. The best advice is to avoid M-M relationships if you can. If you cannot avoid them, budget your time with foreknowledge that a difficult task lies ahead.

Designing Special Case M-M Relationships

Not all M-M relationships are as bad as the last section indicates. Although M-M relationships between strong entities generally are as complicated as described, M-M relationships between strong and weak entities are often easier. For example, consider the relationship between COMPANY and PHONE_CONTACT in Figure 6-29. Because PHONE_CONTACT is an ID-dependent weak, it must have a COMPANY parent. In addition, assume that application

Amazon Ex. 1008.0246

requirements indicate that each COMPANY row must have at least one row in PHONE_CONTACT. Hence, the relationship is M-M.

However, transactions are almost always initiated from the side of the strong entity. A data entry form will begin with a COMPANY and then, somewhere in the body of the form, the data from the PHONE_CONTACT table will appear. Hence, all insert, update, and deletion activity on PHONE_CONTACT will come as a result of some action on COMPANY. Given this situation, we can ignore the Action on Child columns in Figure 6-28(a) and Figure 6-28(b), because no one will ever try to insert, modify, or delete a new PHONE_CONTACT except in the context of inserting, modifying, or deleting a COMPANY.

Because the relationship is M-M, however, we must take all of the actions in the Action on Parent columns of both Figure 6-28(a) and Figure 6-28(b). With regards to inserts on parents, we must always create a child. We can meet this need by writing a COMPANY INSERT trigger that automatically creates a new row of PHONE_CONTACT with null values for Contact and PhoneNumber.

With regard to updates and deletions, all we need to do is to cascade all of the remaining actions in Figure 6-28(a) and Figure 6-28(b). Changes to COMPANY.CompanyName will be propagated to PHONE_CONTACT.CompanyName. The deletion of a COMPANY will automatically delete that company's PHONE_CONTACT rows. This makes sense; if we no longer want data about a company, we certainly no longer want its contact and phone data.

BY THE WAY Because of the difficulty of enforcing M-M relationships, developers look for special circumstances to ease the task. Such circumstances usually exist for relationships between strong and weak entities, as described. For relationships between strong entities, such special circumstances may not exist. In this case, the M-M cardinality is sometimes just ignored. Of course, this cannot be done for applications such as financial management or operations that require careful records management, but, for an application such as airline reservations, where seats are overbooked anyway, it might be better to redefine the relationship as M-O.

Documenting the Minimum Cardinality Design

Because enforcing minimum cardinality can be complicated, and because it often involves the creation of triggers or other procedures, clear documentation is essential. Because the design for the enforcement of required parents is easier than that for required children, we will use different techniques for each.

Documenting Required Parents

Database modeling and design tools such as ERwin, Visio, and MySQL Workbench allow you to define **referential integrity (RI) actions** on each table. These definitions are useful for documenting the actions necessary for required parent. According to Figure 6-28(a), three design decisions are necessary for required parents: (1) determining whether updates to the parent's primary key should cascade or be prohibited; (2) determining whether deletions of the parent should cascade or be prohibited; and (3) identifying how a parent row is to be selected on the insert of a child.

BY THE WAY In theory, referential integrity actions can be used to document the actions to be taken to enforce required children as well as required parents. When they are used for both purposes, however, they become confusing and ambiguous. In an M-M relationship, for example, a child may have one set of rules for insert because of its required parent and another set of rules for insert because it is a required child. The insert referential integrity action will be overloaded with these two purposes, and its meaning will be ambiguous, at best. Hence, in this text, we will use referential integrity actions only for documenting required parents. We will use another technique, described next, for documenting required children.

Amazon Ex. 1008.0247

Documenting Required Children

One easy and unambiguous way for defining the actions to enforce a required child is to use Figure 6-28(b) as a boilerplate document. Create a copy of this figure for each relationship that has a required child and fill in the specific actions for insert, update, and delete operations.

For example, consider Figure 6-31, which shows the O-M relationship between HOUSE and INSPECTION. A given house must have at least one inspection, but an inspection need not be related to any house. HOUSE has a surrogate key, HouseID, and other columns, as shown in Figure 6-31.

Because the HOUSE-to-INSPECTION relationship has a required child, we will fill out the table in Figure 6-28(b). Figure 6-32 shows the result. Here, triggers are described for HOUSE insert and INSPECTION deletion. HOUSE update actions are unneeded because HOUSE has a surrogate key, and INSPECTION update is prohibited because of the surrogate key and also because inspections are never reassigned to a different house.

Figure 6-31

HOUSE-to-INSPECTION O-M Relationship

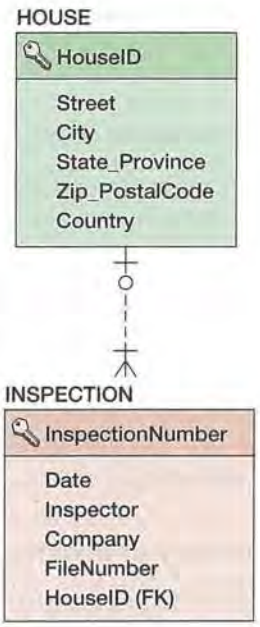


Figure 6-32

Actions to Enforce the O-M Relationship Between HOUSE and INSPECTION

INSPECTION Is Required	Action on HOUSE	Action on INSPECTION
Insert	Trigger to create row in INSPECTION when inserting HOUSE. Disallow HOUSE insert if INSPECTION data are not available.	None.
Modify key or foreign key	Not possible, surrogate key.	Prohibit. HOUSE has surrogate key and inspections never change to a different house.
Delete	None.	Trigger to prohibit if sole INSPECTION report.

Amazon Ex. 1008.0248

Relationship Minimum Cardinality	Design Decisions to Be Made	Design Documentation
M-O	• Update cascade or prohibit? • Delete cascade or prohibit? • Policy for obtaining parent on insert of child	Referential integrity (RI) actions plus documentation for policy on obtaining parent for child insert.
O-M	• Policy for obtaining child on insert of parent • Primary key update cascade or prohibit? • Policy for update of child foreign key • Policy for deletion of child	Use Figure 6-28(b) as a boilerplate
M-M	All decisions for M-O and O-M above, plus how to process trigger conflict on insertion of first instance of parent/child and deletion of last instance of parent/child.	For mandatory parent, RI actions plus documentation for policy on obtaining parent for child insert. For mandatory child, use 6-28(b) as a boilerplate. Add documentation on how to process trigger conflict.

Figure 6-33
Summary of Design Decisions for Minimum Cardinality

An Additional Complication

You should be aware of an additional complication that is beyond the scope of this text. A table can participate in many relationships. In fact, there can be multiple relationships between the same two tables. You need to specify a design for the minimum cardinality of every relationship. The minimum cardinality of each relationship will vary. Some will be O-M, some will be M-O, and some will be M-M. Some of the relationships will require triggers, which may mean that you have several sets of insert, update, and delete triggers per table. This array of triggers is not only complicated to write and test, the actions of different triggers may interfere with one another during execution. You will need more experience and knowledge to design, implement, and test such complex arrays of trigger code and DBMS constraints. For now, just be aware that these problems exist.

Summary of Minimum Cardinality Design

Figure 6-33 summarizes the design for relationship minimum cardinality. It shows each type of relationship, the design decisions that need to be made, and the documentation that should be created. Use this figure as a guide.

The View Ridge Gallery Database

We conclude this chapter with an example database design problem. This design will be used throughout the rest of the text, so take the time to understand it. This particular problem was chosen because it has typical relationships and moderate complexity. It has enough challenges to make it interesting, but not so many as to make it overwhelming.

Summary of Requirements

The **View Ridge Gallery (View Ridge** or **VRG)** is a small art gallery that sells contemporary European and North American fine art, including lithographs, high-quality reproduction prints, original paintings and other artwork, and photographs. All of the lithographs, prints, and photos are signed and numbered, and the original art is usually signed. View Ridge also provides art framing services. It creates a custom frame for each artwork (rather than selling standardized, premade frames), and is known for its excellent collection of frame stock.

View Ridge emphasizes reproduction artworks of European Impressionist, Abstractionist, and Modernist artists such as Wassily Kandinsky and Henri Matisse. For original art, View Ridge

Amazon Ex. 1008.0249

- Track customers and their artist interests
- Record gallery's purchases
- Record customers' art purchases
- List the artists and works that have appeared in the gallery
- Report how fast an artist's works have sold and at what margin
- Show current inventory in a Web page

Figure 6-34
Summary of View Ridge Gallery Database Requirements

concentrates on Northwest School artists, such as Mark Tobey, Morris Graves, Guy Anderson, and Paul Horiuchi, and produces shows of contemporary artists who work in the Northwest School tradition or in Northwest Maritime art. The price of new reproduction prints ranges up to $1,000, and prices for contemporary artists range from $500 to $10,000. The price of art from the Northwest School artists varies considerably, depending on the artwork itself. Small pencil, charcoal, or watercolor sketches may sell for as little as $2,000, whereas major works can range from $10,000 to $100,000. Very occasionally, View Ridge may carry Northwest School art priced up to $500,000, but art priced above $250,000 is more likely to be sold at auction by a major art auction house.

View Ridge has been in business for 30 years and has one full-time owner, three salespeople, and two workers who make frames, hang art in the gallery, and prepare artwork for shipment. View Ridge holds openings and other gallery events to attract customers to the gallery. View Ridge owns all of the art that it sells—even sales of contemporary artwork is treated as a purchase by View Ridge that then is resold to a customer. View Ridge does not take items on a consignment basis.

The requirements for the View Ridge application are summarized in Figure 6-34. First, both the owner and the salespeople want to keep track of customers' names, addresses, phone numbers, and e-mail addresses. They also want to know which artists have appeal to which customers. The salespeople use this information to determine whom to contact when new art arrives and to personalize verbal and e-mail communications with their customers.

When the gallery purchases new art, data about the artist, the nature of the work, the acquisition date, and the acquisition price are recorded. Also, on occasion, the gallery repurchases art from a customer and resells it, thus a work may appear in the gallery multiple times. When art is repurchased, the artist and work data are not reentered, but the most recent acquisition date and price are recorded. In addition, when art is sold, the purchase date, sales price, and identity of the purchasing customer are stored in the database.

Salespeople want to examine past purchase data so that they can devote more time to the most active buyers. They also sometimes use the purchase records to identify the location of artworks they have sold in the past.

For marketing purposes, View Ridge wants its database application to provide a list of artists and works that have appeared in the gallery. The owner also would like to be able to determine how fast an artist's work sells and at what sales margin. The database application also should display current inventory on a Web page that customers can access via the Internet.

The View Ridge Data Model

Figure 6-35 shows a data model for the View Ridge database. This model has two strong entities: CUSTOMER and ARTIST. In addition, the entity WORK is ID dependent on ARTIST, and the entity TRANS is ID dependent on WORK. There is also a nonidentifying relationship from CUSTOMER to TRANS.

Note that we are using the entity name TRANS instead of TRANSACTION. We are doing this because *transaction* is a **DBMS reserved word** in most (if not all) DBMS products. Using DBMS reserved words such as *table*, *column*, or other names can create problems. Similarly, we cannot use the reserved word *tran*. The word *trans*, however, is not a DBMS reserved word, and we can use it without problems. We will discuss this problem more when we discuss specific DBMS products in Chapters 10, 10A, and 10B.

Amazon Ex. 1008.0250

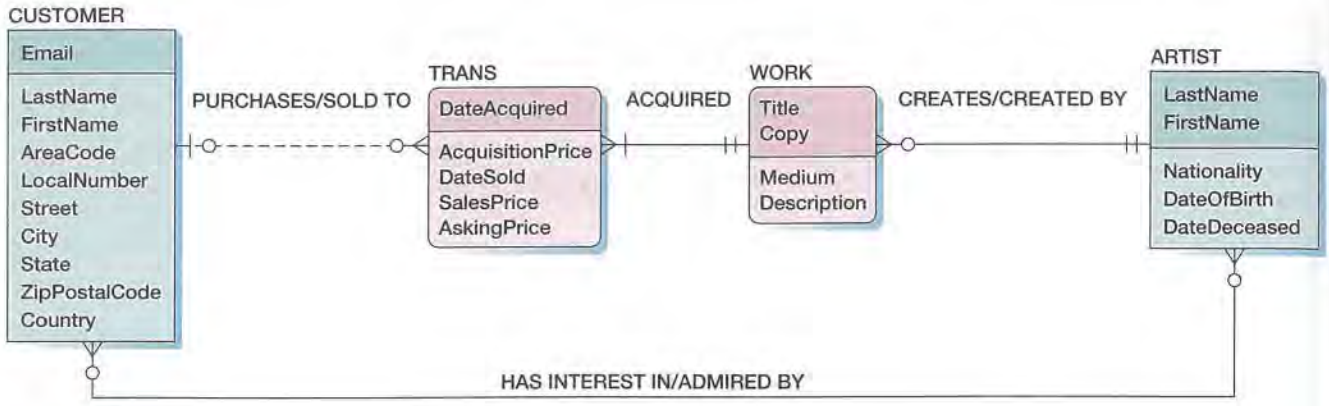


Figure 6-35
View Ridge Gallery Data Model

In the View Ridge data model, an artist may be recorded in the database even if none of his or her works has appeared in the gallery. This is done to record customer preferences for artists whose works might appear in the future. Thus, an artist may have from zero to many works.

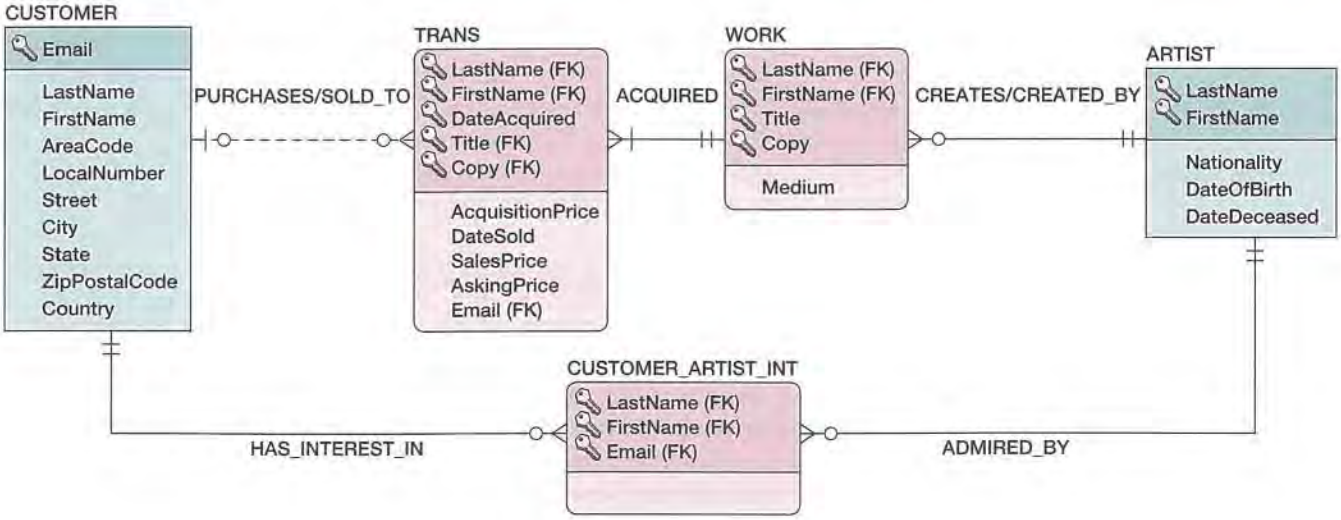
The identifier of WORK is the composite (Title, Copy) because, in the case of lithographs and photos, there may be many copies of a given title. Also, the requirements indicate that a work may appear in the gallery many times, so there is a need for potentially many TRANS entities for each WORK. Each time a work appears in the gallery, the acquisition date and price must be recorded. Thus, each WORK must have at least one TRANS row.

A customer may purchase many works; this is recorded in the 1:N relationship from CUSTOMER to TRANS. Note that this relationship is optional in both directions. Finally, there is an N:M relationship between CUSTOMERs and ARTISTs. This is an N:M relationship between strong entities—the team searched in vain for a missing attribute that would indicate an association pattern rather than an N:M relationship.

Database Design with Data Keys

A database design for the data model in Figure 6-35 is shown in Figure 6-36. This design uses data keys, and every primary key except the composite (ARTIST.LastName, ARTIST.FirstName) has problems. The keys for WORK and TRANS are huge, and the key for CUSTOMER is doubtful; many customers may not have an e-mail address. Because of these problems, this design cries out for surrogate keys.

Figure 6-36
Initial View Ridge Gallery Database Design



Amazon Ex. 1008.0251

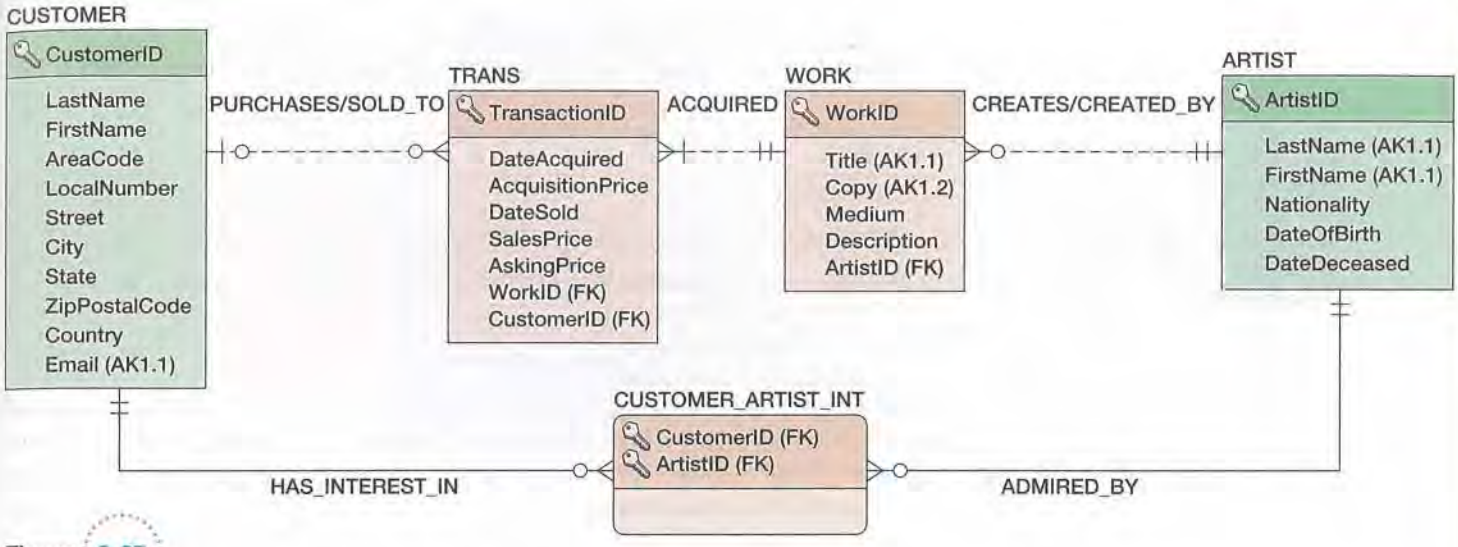


Figure 6-37
Final View Ridge Gallery Database Design

Surrogate Key Database Design

The database design for the View Ridge database using surrogate keys is shown in Figure 6-37. Notice that two identifying relationships (TRANS-to-WORK) and (WORK-to-ARTIST) have been changed to nonidentifying relationships represented by dashed lines. This was done because once ARTIST has a surrogate key, there is no need to keep ID-dependent keys in WORK and TRANS. Realize that WORK and TRANS are both weak entities even though they are no longer ID dependent.

Notice that (LastName, FirstName) in ARTIST has been defined as an alternate key. This notation indicates that (LastName, FirstName) has a UNIQUE constraint, which ensures that artists are not duplicated in the database. Similarly, (Title, Copy) in WORK is defined as an alternate key so that a given work cannot appear more than once.

The foreign key placement is a straightforward application of the techniques described in this chapter, but note that the foreign key CustomerID in TRANS can have null values. This specification allows the creation of a TRANS row when a work is acquired, before any customer has purchased the work. All other foreign keys are required.

Minimum Cardinality Enforcement for Required Parents

According to Figure 6-28(a), for each relationship that involves a required parent, we need to decide:

- Whether to cascade or prohibit updates of the parent's primary key
- Whether to cascade or prohibit deletions of the parent
- How to obtain a parent when a new child is created.

Because there is no consistent means of documenting these actions in commercial database design products, we will use the templates in Figure 6-28 to document our decisions. Figure 6-38 summarizes the relationships in the View Ridge database design.

Because all tables have surrogate keys, there is no need for any update cascade behavior for any parent. However, some update actions on child tables must be restricted. For example, once a WORK (child) is assigned to an ARTIST (parent), it is never to change to another parent. Because this database is used to record purchases and sales, View Ridge management never wants to delete any data that are related to a transaction. From time to time, they may remove prior year's data in bulk, but they will do that using bulk data transfer and not as part of any application.

Amazon Ex. 1008.0252

Relationship		Cardinality		
Parent	Child	Type	MAX	MIN
ARTIST	WORK	Nonidentifying	1:N	M-O
WORK	TRANS	Nonidentifying	1:N	M-M
CUSTOMER	TRANS	Nonidentifying	1:N	O-O
CUSTOMER	CUSTOMER_ARTIST_INT	Identifying	1:N	M-O
ARTIST	CUSTOMER_ARTIST_INT	Identifying	1:N	M-O

Figure 6-38
Summary of View Ridge Database Design Relationships

Hence, any CUSTOMER, WORK, or ARTIST row that is related to a TRANS row is never to be deleted. Note, however, that rows of CUSTOMERs who have never made a purchase and rows of ARTISTs whose works have never been carried in the gallery can be deleted. If either a CUSTOMER or ARTIST is deleted under these circumstances, the deletion will cascade to rows in the intersection table CUSTOMER_ARTIST_INT.

Finally, referential integrity actions are necessary for obtaining a parent WORK when a TRANS record is created, and a parent ARTIST when a WORK record is created. In both cases, the policy will be for the application program to provide the ID of the required parent at the time the WORK or TRANS record is created.

All these actions are documented in Figure 6-39, where each part is based on the template for required children shown in Figure 6-28(a). Note that there is no diagram for the CUSTOMER-to-TRANS relationship, because that is an O-O relationship without a required parent (or child).

Figure 6-39
Actions to Enforce Minimum Cardinality for Required Parents

ARTIST Is Required Parent	Action on ARTIST (Parent)	Action on WORK (Child)
Insert	None.	Get a parent.
Modify key or foreign key	Prohibit—ARTIST uses a surrogate key.	Prohibit—ARTIST uses a surrogate key.
Delete	Prohibit if WORK exists—data related to a transaction is never deleted (business rule). Allow if no WORK exists (business rule).	None.

(a) For the ARTIST-to-WORK Relationship

WORK Is Required Parent	Action on WORK (Parent)	Action on TRANS (Child)
Insert	None.	Get a parent.
Modify key or foreign key	Prohibit—WORK uses a surrogate key.	Prohibit—WORK uses a surrogate key.
Delete	Prohibit—data related to a transaction is never deleted (business rule).	None.

(b) For the WORK-to-TRANS Relationship

Amazon Ex. 1008.0253

CUSTOMER Is Required Parent	Action on CUSTOMER (Parent)	Action on CUSTOMER_ARTITST_INT (Child)
Insert	None.	Get a parent.
Modify key or foreign key	Prohibit—CUSTOMER uses a surrogate key.	Prohibit—CUSTOMER uses a surrogate key.
Delete	Prohibit if TRANS exists—data related to a transaction is never deleted (business rule). Allow if no TRANS exists (business rule)—cascade delete children.	None.

(c) For the CUSTOMER-to-CUSTOMER_ARTIST_INT Relationship

ARTIST Is Required Parent	Action on ARTIST (Parent)	Action on CUSTOMER_ARTITST_INT (Child)
Insert	None.	Get a parent.
Modify key or foreign key	Prohibit—ARTIST uses a surrogate key.	Prohibit—ARTIST uses a surrogate key.
Delete	Prohibit if TRANS exists—data related to a transaction is never deleted (business rule). Allow if no TRANS exists (business rule)—cascade delete children.	None.

Figure 6-39 (Continued)

(d) For the ARTIST-to-CUSTOMER_ARTIST_INT Relationship

Minimum Cardinality Enforcement for the Required Child

As shown in the summary in Figure 6-38, TRANS is the only required child in the database design in Figure 6-37. The actions to enforce that required child are documented in Figure 6-40, which is based on the template in Figure 6-28(b).

TRANS Is Required Parent	Action on WORK (Parent)	Action on TRANS (Child)
Insert	INSERT trigger on WORK to create row in TRANS. TRANS will be given data for DateAcquired and AcquisitionPrice. Other columns will be null.	Will be created by INSERT trigger on WORK.
Modify key or foreign key	Prohibit—surrogate key.	Prohibit—TRANS must always refer to the WORK associated with it.
Delete	Prohibit—data related to a transaction is never deleted (business rule).	Prohibit—data related to a transaction is never deleted (business rule).

Figure 6-40 Actions to Enforce Minimum Cardinality for Required Children for the WORK-to-TRANS Relationship

Amazon Ex. 1008.0254

According to this document, an INSERT trigger on WORK will be written to create the required child. This trigger will be fired whenever a work is first introduced at the gallery. At that time, a new TRANS row will be created to store the values for DateAcquired and AcquisitionPrice.

Changes to the primary key in WORK will not occur because it has a surrogate key. Changes to the foreign key in TRANS will not be allowed because a TRANS never switches to another work. As stated earlier, the gallery has the policy that no transaction or related data will ever be deleted. Consequently, deletions of either WORK or TRANS are not allowed.

Column Properties for the View Ridge Database Design Tables

As we discussed at the beginning of this chapter, besides naming the columns in each table, we must specify the column properties summarized in Figure 6-1 for each column: null status, data type, default value (if any), and data constraints (if any). These are shown Figure 6-41.

With this step, we have completed our database design for the View Ridge Gallery database, and now we are ready to create it as an actual, functioning database in a DBMS product. We will do so in many of the following chapters, so be certain that you understand the View Ridge Gallery database design we have built.

Figure 6-41

Database Column Properties for the View Ridge Database Design

Column Name	Type	Key	NULL Status	Remarks
ArtistID	Int	Primary Key	NOT NULL	Surrogate Key IDENTITY (1,1)
LastName	Char (25)	Alternate Key	NOT NULL	Unique (AK1.1)
FirstName	Char (25)	Alternate Key	NOT NULL	Unique (AK1.2)
Nationality	Char (30)	No	NULL	IN ('Canadian', 'English', 'French', 'German', 'Mexican', 'Russian', 'Spanish', 'United States')
DateOfBirth	Numeric (4)	No	NULL	(DateOfBirth < DateDeceased) (BETWEEN 1900 and 2999)
DateDeceased	Numeric (4)	No	NULL	(BETWEEN 1900 and 2999)

(a) Column Characteristics for the ARTIST Table

Column Name	Type	Key	NULL Status	Remarks
WorkID	Int	Primary Key	NOT NULL	Surrogate Key IDENTITY (500,1)
Title	Char (35)	Alternate Key	NOT NULL	Unique (AK1.1)
Copy	Char (12)	Alternate Key	NOT NULL	Unique (AK1.2)
Medium	Char (35)	No	NULL	
Description	Varchar (1000)	No	NULL	DEFAULT value = 'Unknown provenance'
ArtistID	Int	Foreign Key	NOT NULL	

(b) Column Characteristics for the WORK Table

Amazon Ex. 1008.0255

Column Name	Type	Key	NULL Status	Remarks
TransactionID	Int	Primary Key	NOT NULL	Surrogate Key IDENTITY (100,1)
DateAcquired	Date	No	NOT NULL	
AcquisitionPrice	Numeric (8,2)	No	NOT NULL	
AskingPrice	Numeric (8,2)	No	NULL	
DateSold	Date	No	NULL	(DateAcquired <= DateSold)
SalesPrice	Numeric (8,2)	No	NULL	(SalesPrice > 0) AND (SalesPrice <=500000)
CustomerID	Int	Foreign Key	NULL	
WorkID	Int	Foreign Key	NOT NULL	

(c) Column Characteristics for the TRANS Table

Column Name	Type	Key	NULL Status	Remarks
CustomerID	Int	Primary Key	NOT NULL	Surrogate Key IDENTITY (100,1)
LastName	Char (25)	No	NOT NULL	
FirstName	Char (25)	No	NOT NULL	
Street	Char (30)	No	NULL	
City	Char (35)	No	NULL	
State	Char (2)	No	NULL	
ZipPostalCode	Char (9)	No	NULL	
Country	Char (50)	No	NULL	
AreaCode	Char (3)	No	NULL	
PhoneNumber	Char (8)	No	NULL	
Email	Varchar (100)	Alternate Key	NULL	Unique (AK 1.1)

(d) Column Characteristics for the CUSTOMER Table

Column Name	Type	Key	NULL Status	Remarks
ArtistID	Int	Primary Key, Foreign Key	NOT NULL	
CustomerID	Int	Primary Key, Foreign Key	NOT NULL	

(e) Column Characteristics for the CUSTOMER_ARTIST_INT Table

Figure 6-41
(Continued)

Amazon Ex. 1008.0256

Summary

Transforming a data model into a database design requires three major tasks: replacing each entity with a table and each attribute with a column; representing relationships and maximum cardinality by placing foreign keys; and representing minimum cardinality by defining actions to constrain activities on values of primary and foreign keys.

During database design, each entity is replaced by a table. The attributes of the entity become columns of the table. The identifier of the entity becomes the primary key of the table, and candidate keys in the entity become candidate keys in the table. A good primary key is short, numeric, and fixed. If a good primary key is not available, a surrogate key may be used instead. Some organizations choose to use surrogate keys for all of their tables. An alternate key is the same as a candidate key and is used to ensure unique values in a column. The notation AK*n.m* refers to the *n*th alternative key and the *m*th column in that key.

Four properties need to be specified for each table column: null status, data type, default value, and data constraints. A column can be NULL or NOT NULL. Primary keys are always NOT NULL; alternate keys can be NULL. Data types depend on the DBMS to be used. Generic data types include CHAR(n), VARCHAR(n), DATE, TIME, MONEY, INTEGER, and DECIMAL. A default value is a value to be supplied by the DBMS when a new row is created. It can be a simple value or the result of a function. Sometimes triggers are needed to supply values of more complicated expressions.

Data constraints include domain constraints, range constraints, intrarelation constraints, and interrelation constraints. Domain constraints specify a set of values that a column may have; range constraints specify an interval of allowed values; intrarelation constraints involve comparisons among columns in the same table; and interrelation constraints involve comparisons among columns in different tables. A referential integrity constraint is an example of an interrelation constraint.

Once the tables, keys, and columns have been defined, they should be checked against normalization criteria. Usually the tables will already be normalized, but they should be checked in any case. Also, it may be necessary to denormalize some tables.

The second step in database design is to create relationships by placing foreign keys appropriately. For 1:1 strong relationships, the key of either table can go in the other table as a foreign key; for 1:N strong relationships, the key of the parent must go in the child; and for N:M strong relationships, a new table, called an intersection table, is constructed that has the keys of both tables. Intersection tables never have nonkey data.

Four uses for ID-dependent entities are N:M relationships, association relationships, multivalued attributes, and archetype/instance relationships. An association relationship differs from an intersection table because the ID-dependent entity has nonkey data. In all ID-dependent entities, the key of the parent is already in the child. Therefore, no foreign key needs to be created. When an instance entity of the archetype/instance pattern is given a non-ID-dependent identifier, it changes from an ID-dependent entity to a weak entity. The tables that represent such entities must have the key of the parent as a foreign key. They remain weak entities, however. When the parent of an ID-dependent entity is given a surrogate key, the ID-dependent entity is also given a surrogate key. It remains a weak entity, however.

Mixed entities are represented by placing the key of the parent of the nonidentifying relationship into the child. The key of the parent of the identifying relationship will already be in the child. Subtypes are represented by copying the key from the supertype into the subtype(s) as a foreign key. Recursive relationships are represented in the same ways that 1:1, 1:N, and N:M relationships are represented. The only difference is that the foreign key references rows in the table in which it resides.

Ternary relationships are decomposed into binary relationships. However, sometimes binary constraints must be documented. Three such constraints are MUST, MUST NOT, and MUST COVER.

The third step in database design is to create a plan for enforcing minimum cardinality. Figure 6-28 shows the actions that need to be taken to enforce minimum cardinality for required parents and required children. The actions in Figure 6-28(a) must be taken for M-O and M-M relationships; the actions in Figure 6-28(b) must be taken for O-M and M-M relationships.

Enforcing mandatory parents can be done by defining the appropriate referential integrity constraint and by setting the foreign key to NOT NULL. The designer must specify whether updates to the parent's primary key will cascade or be prohibited, whether deletions to the parent will cascade or be prohibited, and what policy will be used for finding a parent when a new child is created.

Enforcing mandatory children is difficult and requires the use of triggers or application code. The particular actions that need to be taken are shown in Figure 6-28(b). Enforcing M-M relationships can be very difficult. Particular challenges concern the creation of the first parent/child rows and the deletion of the last parent/child rows. The triggers on the two tables interfere with one another. M-M relationships between strong and weak entities are not as problematic as those between strong entities.

In this text, the actions to enforce required parents are documented using referential integrity actions on the table design diagrams. The actions to enforce required children are documented by using Figure 6-28(b) as a boilerplate document. An additional complication is that a table can participate in many relationships. Triggers written to enforce the minimum cardinality on one relationship may

Amazon Ex. 1008.0257

interfere with triggers written to enforce the minimum cardinality on another relationship. This problem is beyond the scope of this text, but be aware that it exists. The principles for enforcing minimum cardinality are summarized in Figure 6-33.

A database design for the View Ridge Gallery is shown in Figures 6-37, 6-38, 6-39, 6-40, and 6-41. You should understand this design, because it will be used throughout the remainder of this book.

Key Terms

action
alternate key (AK)
candidate key
cascading deletion
cascading update
data constraint
DBMS reserved word
default value
domain constraint
interrelation constraint
intersection table
intrarelation constraint
minimum cardinality enforcement action
MUST constraint
MUST COVER constraint
MUST NOT constraint
null status
parent mandatory and child mandatory (M-M)
parent mandatory and child optional (M-O)
parent optional and child mandatory (O-M)
parent optional and child optional (O-O)
range constraint
referential integrity (RI) action
surrogate key
trigger
View Ridge Gallery (View Ridge or VRG)

Review Questions

6.1 Identify the three major tasks for transforming a data model into a database design.

6.2 What is the relationship between entities and tables? Between attributes and columns?

6.3 Why is the choice of the primary key important?

6.4 What are the three characteristics of an ideal primary key?

6.5 What is a surrogate key? What are its advantages?

6.6 When should you use a surrogate key?

6.7 Describe two disadvantages of surrogate keys.

6.8 What is the difference between an alternate key and a candidate key?

6.9 What does the notation LastName (AK2.2) mean?

6.10 Name four column properties.

6.11 Explain why primary keys may never be null, but alternate keys can be null.

6.12 List five generic data types.

6.13 Describe three ways that a default value can be assigned.

6.14 What is a domain constraint? Give an example.

6.15 What is a range constraint? Give an example.

6.16 What is an intrarelation constraint? Give an example.

6.17 What is an interrelation constraint? Give an example.

6.18 What tasks should be accomplished when verifying normalization of a database design?

Amazon Ex. 1008.0258

6.19 Describe two ways to represent a 1:1 strong entity relationship. Give an example other than one in this chapter.

6.20 Describe how to represent a 1:N strong entity relationship. Give an example other than one in this chapter.

6.21 Describe how to represent an N:M strong entity relationship. Give an example other than one in this chapter.

6.22 What is an intersection table? Why is it necessary?

6.23 What is the difference between the table that represents an ID-dependent association entity and an intersection table?

6.24 List four uses for ID-dependent entities.

6.25 Describe how to represent an association entity relationship. Give an example other than one in this chapter.

6.26 Describe how to represent a multivalued attribute entity relationship. Give an example other than one in this chapter.

6.27 Describe how to represent a version/instance entity relationship. Give an example other than one in this chapter.

6.28 What happens when an instance entity is given a non-ID-dependent identifier? How does this change affect relationship design?

6.29 What happens when the parent in an ID-dependent relationship is given a surrogate key? What should the key of the child become?

6.30 Describe how to represent a mixed entity relationship. Give an example other than one in this chapter.

6.31 Describe how to represent a supertype/subtype entity relationship. Give an example other than one in this chapter.

6.32 Describe two ways to represent a 1:1 recursive relationship. Give an example other than one in this chapter.

6.33 Describe how to represent a 1:N recursive relationship. Give an example other than one in this chapter.

6.34 Describe how to represent an N:M recursive relationship. Give an example other than one in this chapter.

6.35 In general, how are ternary relationships represented? Explain how a binary constraint may impact such a relationship.

6.36 Describe a MUST constraint. Give an example other than one in this chapter.

6.37 Describe a MUST NOT constraint. Give an example other than one in this chapter.

6.38 Describe a MUST COVER constraint. Give an example other than one in this chapter.

6.39 Explain in general terms what needs to be done to enforce minimum cardinality.

6.40 Explain the need for each of the actions in Figure 6-28(a).

6.41 Explain the need for each of the actions in Figure 6-28(b).

6.42 State which of the actions in Figure 6-28 must be applied for M-O relationships, O-M relationships, and M-M relationships.

6.43 Explain what must be done for the DBMS to enforce required parents.

6.44 What design decisions must be made to enforce required parents?

6.45 Explain why the DBMS cannot be used to enforce required children.

6.46 What is a trigger? How can triggers be used to enforce required children?

Amazon Ex. 1008.0259

6.47 Explain why the enforcement of M-M relationships is particularly difficult.

6.48 Explain the need for each of the design decisions in Figure 6-33.

6.49 Explain the implications of each of the minimum cardinality specifications in Figure 6-38.

6.50 Explain the rationale for each of the entries in the table in Figure 6-40.

Project Questions

6.51 Answer Project Question 5.58 if you have not already done so. Design a database for your model in Project Question 5.58. Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for a required parent, if any, and the form in Figure 6-28(b) for a required child, if any.

6.52 Answer Project Question 5.59 if you have not already done so. Design a database for your model. Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.

6.53 Answer Project Question 5.60 if you have not already done so. Design a database for your model in Project Question 5.60(c). Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.

6.54 Answer Project Question 5.61 if you have not already done so. Design a database for your model in Project Question 5.61(d). Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.

6.55 Answer Project Question 5.62 if you have not already done so. Design databases for your model in Project Question 5.62(a) and for the model in Figure 5-58. Your designs should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.

6.56 Answer Project Question 5-63 if you have not already done so. Design a database for your model in Project Question 5-63(e). Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.

6.57 Answer Project Question 5-64 if you have not already done so. Design a database for your model in Project Question 5-64(c). Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.

Amazon Ex. 1008.0260

6.58 Answer Project Question 5-65 if you have not already done so. Design a database for your model in Project Question 5-65(d). Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.



If you have not already done so, complete the Marcia's Dry Cleaning project at the end of Chapter 5. Design a database for the model in your answer. Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.



If you have not already done so, answer the Morgan Importing project at the end of Chapter 5. Design a database for the model in your answer. Your design should include a specification of tables and attributes as well as primary, candidate, and foreign keys. Also specify how you will enforce minimum cardinality. Document your minimum cardinality enforcement using referential integrity actions for required parents, if any, and the form in Figure 6-28(b) for required children, if any.

Amazon Ex. 1008.0261

Part 3

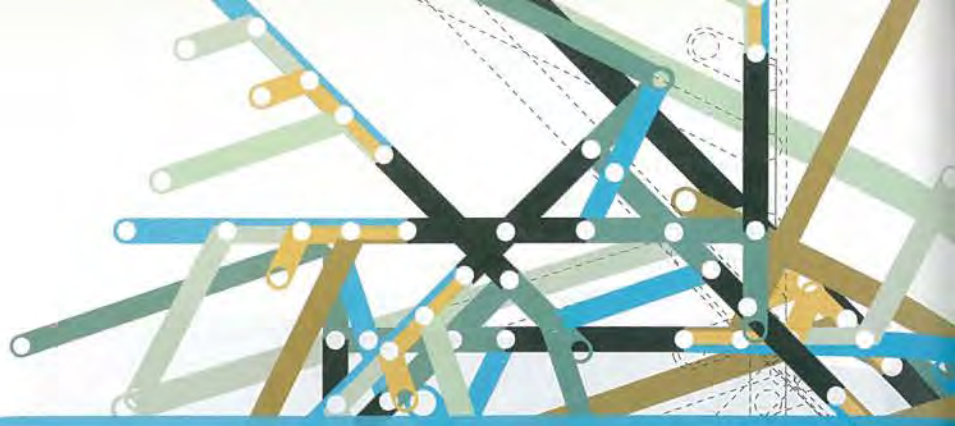
Database Implementation

Part 3 consists of two chapters. Chapter 7 presents SQL data definition language statements for constructing database components and describes the SQL data manipulation statements for inserting, updating, and deleting data. You will also learn how to construct and use SQL views and how to embed SQL statements into programs. The chapter concludes with a discussion of SQL triggers and stored procedures.

Chapter 8 presents the use of SQL statements to redesign databases. It presents SQL correlated subqueries and then introduces SQL statements using the SQL EXISTS and NOT EXISTS keywords. Both of these advanced SQL statements are needed for database redesign. Chapter 8 also describes database reverse engineering, surveys common database redesign problems, and shows how to use SQL to solve database redesign problems.



Amazon Ex. 1008.0262



7 SQL for Database Construction and Application Processing

Chapter Objectives

- To be able to create and manage table structures using SQL statements.
- To understand how referential integrity actions are implemented in SQL statements.
- To be able to create and use SQL constraints.
- To understand several uses for SQL views.
- To be able to use SQL statements to create and use views.
- To gain an understanding of how SQL is used in an application program.
- To understand how to create and use triggers.
- To understand how to create and use stored procedures.

In Chapter 2, we introduced SQL and classified SQL statements into two categories: data manipulation language (DML) statements, which are used for querying, inserting, updating and deleting data, and data definition language (DDL) statements, which are used for creating tables, relationships, and other database structures. In Chapter 2, we discussed only DML query statements.

This chapter describes and illustrates SQL DDL statements for constructing databases and SQL DML statements for inserting, modifying, and deleting data. We will also describe how to create SQL Views, how to embed SQL statements into application programs, and how to use SQL in triggers and stored procedures.

Amazon Ex. 1008.0263

The knowledge in this chapter is important whether you become a database administrator or an application programmer. Even if you will not construct SQL triggers or stored procedures yourself, it is important that you know what they are, how they work, and how they influence database processing.

The View Ridge Gallery Database

In Chapter 6, we introduced the View Ridge Gallery, a small art gallery that sells contemporary North American and European fine art and provides art framing services. We also developed a data model and database design for a database for the View Ridge Gallery. Our final database design for View Ridge is shown in Figure 7-1. In this chapter, we will use SQL to build the View Ridge database based on that design.

SQL DDL, DML, and a New Type of Join

Figure 7-2 summarizes the new SQL statements described in this chapter. We begin with SQL DDL statements for managing table structures, including CREATE TABLE, ALTER TABLE, and DROP TABLE. Using these statements, we will build the table structure for the View Ridge

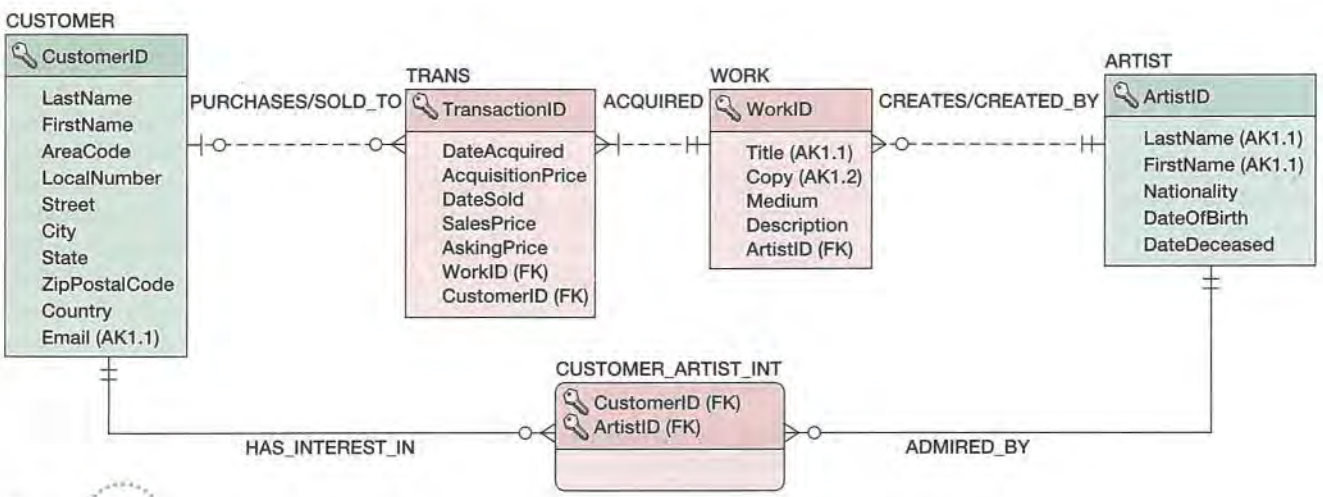


Figure 7-1
Final View Ridge Gallery Database Design

- SQL Data Definition Language (DDL)
  - CREATE TABLE
  - ALTER TABLE
  - DROP TABLE
- SQL Data Manipulation Language (DML)
  - INSERT
  - UPDATE
  - DELETE
- Additional join forms
  - Alternative join syntax
  - Outer joins

Figure 7-2
Chapter 7 SQL Statements

Amazon Ex. 1008.0264

database. Next, we present the three SQL DML statements for managing data: INSERT, UPDATE, and DELETE. Finally, we will add to the knowledge of joins you gained in Chapter 2 by describing a new format and offering a further discussion of SQL joins.

Managing Table Structure with SQL DDL

The **SQL CREATE TABLE statement** is used to construct tables, define columns and column constraints, and create relationships. Most DBMS products provide graphical tools for performing these tasks, and you may be wondering why you need to learn SQL to perform the same work. There are four reasons. First, creating tables and relationships with SQL is quicker than with graphical tools. Once you know how to use the SQL CREATE TABLE statement, you will be able to construct tables faster and more easily than by fussing around with buttons and graphical gimmickry. Second, some applications, particularly those for reporting, querying, and data mining, require you to create the same table repeatedly. You can do this efficiently if you create an SQL script text file with the necessary SQL CREATE TABLE statements. You then just execute the SQL script when you need to re-create a table. Third, some applications require you to create temporary tables during application work. Chapter 13 (pages 540–543) shows one such application. The only way to create tables from program code is to use SQL. Finally, SQL DDL is standardized and DBMS independent. With the exception of some data types, the same CREATE TABLE statement will work with SQL Server, Oracle, DB2, or MySQL.

Creating the View Ridge Database

Of course, before you can create any tables, you have to create the database. The SQL-92 and subsequent standards include an SQL statement for creating databases, but it is seldom used. Instead, most developers use special commands or graphical tools for creating a database. These techniques are DBMS specific, and we describe them in context in Chapters 10 (SQL Server 2008), 10A (Oracle Database 11g), 10B (MySQL 5.1), and Appendix A (Microsoft Access).

At this point, we highly recommend that you read the section on creating a new database in the DBMS product you are using and use the appropriate steps to create a new database named VRG. For illustrative purposes, we will use SQL Server 2008 in this chapter, and our SQL code will be the correct code for SQL Server 2008. The correct SQL statements for other DBMS products will be similar, but they will vary slightly. The correct SQL statements for Oracle and MySQL can be found in Chapters 10A and 10B, respectively. Figure 7-3 shows the VRG database is the SQL Server 2008 Management Studio.

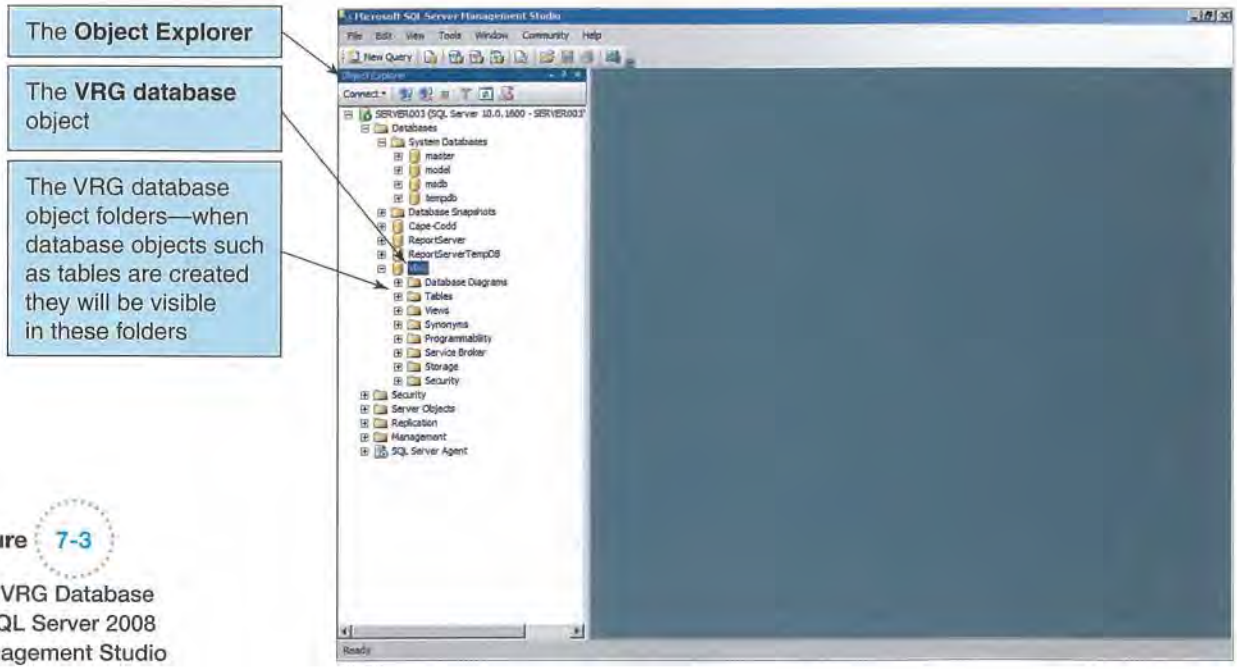


Figure 7-3
The VRG Database in SQL Server 2008 Management Studio

Amazon Ex. 1008.0265

Using the SQL CREATE TABLE Statement

The basic format of the SQL CREATE TABLE statement is:

```
CREATE TABLE NewTableName (
    three-part column definition,
    three-part column definition,
    . . .
    optional table constraints
    . . .
    );
```

The parts of the three-part column definition are the column name, the column data type, and, optionally, a constraint on column values. Thus, we can restate the CREATE TABLE format as:

```
CREATE TABLE NewTableName (
    ColumnName  DataType  OptionalConstraint,
    ColumnName  DataType  OptionalConstraint,
    . . .
    Optional table constraint
    . . .
    );
```

The column and table constraints we consider in this text are **PRIMARY KEY**, **FOREIGN KEY**, **NOT NULL**, **NULL**, **UNIQUE**, and **CHECK**. Additionally, the **DEFAULT keyword** (DEFAULT is not considered a column constraint) can be used to set initial values. Finally, most variants of SQL support a property to implement surrogate primary keys. For example, SQL Server 2008 uses the **IDENTITY({StartValue},{Increment}) property**. Oracle, MySQL, and Access use somewhat different techniques for creating surrogate keys. If you are using those products, see the discussion of surrogate keys for Oracle in Chapter 10A, MySQL in Chapter 10B, or Microsoft Access in Appendix A. We will explain each of these constraints, keywords, and properties as we meet them in the context of our discussion in this chapter.

Variations in SQL Data Types

Each DBMS product has its own variant or extension of SQL, where the extensions are usually **procedural language** extensions, which are additions that allow SQL to function similarly to a procedural programming language (for example, IF . . . THEN . . . ELSE structures). Some vendors have given their variants or extended versions specific names. SQL Server's extended version of SQL is called **Transact-SQL (T-SQL)**, whereas Oracle's extended version of SQL is called **Procedural Language/SQL (PL/SQL)**. MySQL's variant, even though it, too, contains procedural extensions, is just called SQL. We will point out specific SQL syntax differences as we encounter them in our discussion. For more on T-SQL, see the SQL Server 2008 Books Online section *Transact-SQL Reference* at *http://msdn.microsoft.com/en-us/library/bb510741.aspx*. For more on PL/SQL, see the Oracle Database PL/SQL User's Guide and Reference 10g Release 2 (10.2) at *http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14261/toc.htm*. For more on SQL in MySQL, see the MySQL 5.1 Reference Manual Chapter 12 on SQL Statement Syntax at *http://dev.mysql.com/doc/refman/5.1/en/sql-syntax.html*.

One source of variation in DBMS SQL stems from the different data types implemented by each vendor. The SQL standard defines a set of data types. Figure 7-4 shows a summary of common (but not all) data types for the DBMSs we have been discussing.

Amazon Ex. 1008.0266

BY THE WAY Even though Microsoft Access reads standard SQL and the SQL used by SQL Server 2008, the results may be a bit different. For example, Microsoft Access ANSI-89 SQL coverts both the Char and VarChar SQL data types to a fixed Text data type.

Creating the ARTIST Table

We will start by considering two of the tables in the View Ridge database design we developed at the end of Chapter 6, the ARTIST table and the WORK table. These tables are shown in Figure 7-1, and Figures 7-5 and 7-6 show the database column properties for these tables. Three new features are shown in these figures.

The first is the Microsoft SQL IDENTITY property, which is used to specify surrogate keys. In the ARTIST table, the expression IDENTITY(1, 1) means that ArtistID is to be a surrogate key with values starting at 1 and incremented by 1. Thus, the value of ArtistID for the second row in ARTIST will be (1 + 1) = 2. In the WORK table, the expression IDENTITY(500, 1) means that WorkID is to be a surrogate key with values starting at 500 and incremented by 1. Thus, the value of WorkID for the second row in WORK will be (500 + 1) = 501.

The second new feature is the designation of (LastName, FirstName) in ARTIST as an alternative key. This indicates that (LastName, FirstName) is a candidate key for the ARTIST table. Alternative keys are defined using the UNIQUE constraint.

Figure 7-4
SQL Data Types in DBMS Products

Data Type	Description
Binary	Binary, length 0 to 8,000 bytes.
Char	Character, length 0 to 8,000 bytes.
Datetime	8-byte datetime. Range from January 1, 1753, through December 31, 9999, with an accuracy of three-hundredths of a second.
Image	Variable length binary data. Maximum length 2,147,483,647 bytes.
Integer	4-byte integer. Value range from –2,147,483,648 through 2,147,483,647.
Money	8-byte money. Range from –922,337,203,685,477.5808 through +922,337,203,685,477.5807, with accuracy to a ten-thousandth of a monetary unit.
Numeric	Decimal – can set precision and scale. Range –10^38 +1 through 10^38 –1.
Smalldatetime	4-byte datetime. Range from January 1, 1900, through June 6, 2079, with an accuracy of one minute.
Smallint	2-byte integer. Range from –32,768 through 32,767.
Smallmoney	4-byte money. Range from 214,748.3648 through +214,748.3647, with accuracy to a ten-thousandth of a monetary unit.
Text	Variable length text, maximum length 2,147,483,647 characters.
Tinyint	1-byte integer. Range from 0 through 255.
Varchar	Variable-length character, length 0 to 8,000 bytes.

(a) Common Data Types in SQL Server

Data Type	Description
BLOB	Binary large object. Up to 4 gigabytes in length.
CHAR(n)	Fixed length character field of length *n*. Maximum 2,000 characters.
DATE	7-byte field containing both date and time.
INTEGER	Whole number of length 38.
NUMBER(n,d)	Numeric field of length *n*, *d* places to the right of the decimal.
VARCHAR(n) or VARCHAR2(n)	Variable length character field up to *n* characters long. Maximum value of *n* = 4,000.

(b) Common Data Types in Oracle

Amazon Ex. 1008.0267

NumericData Type	Description
BIT (M)	M = 1 to 64
TINYINT	–128 to 127
TINYINT UNSIGNED	0 to 255
BOOLEAN	0 = FALSE; 1 = TRUE
SMALLINT	–32,768 to 32,767
SMALLINT UNSIGNED	0 to 65535
MEDIUMINT	–8,388,608 to 8,388,607
MEDIUMINT UNSIGNED	0 to 16,777,215
INT or INTEGER	–2,147,483,648 to 2,147,483,647
INT UNSIGNED or INTEGER UNSIGNED	0 to 4,294,967,295
BIGINT	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
BIGINT UNSIGNED	0 to 1,844,674,073,709,551,615
FLOAT (P)	P = Precision; 0 to 24
FLOAT (M, D)	Small (single-precision) floating-point number: M = Display width D = Number of significant digits
DOUBLE (M, B)	Normal (double-precision) floating-point number: M = Display width B = Precision; 25 to 53
DEC (M[,D]) or DECIMAL (M[,D]) or FIXED (M[,D])	Fixed-point number: M = Total number of digits D = Number of decimals
Date and Time Data Types	**Description**
DATE	YYYY-MM-DD : 1000-01-01 to 9999-12-31
DATETIME	YYYY-MM-DD HH:MM:SS
	1000-01-01 00:00:00 to 9999-12-31 23:59:59
TIMESTAMP	See documentation.
TIME	HH:MM:SS—00:00:00 to 23:59:59
YEAR (M)	M = 2 or 4 (default)
	IF 2 = 1970 to 2069 (70 to 60)
	IF 4 = 1901 to 2155
String Data Types	**Description**
CHAR (M)	M = 0 to 255
VARCHAR (M)	M = 1 to 255
BLOB (M)	BLOB = Binary Large Object; maximum 65,535 characters
TEXT (M)	Maximum 65,535 characters
TINYBLOB MEDIUMBLOB LONGBLOB TINYTEXT MEDIUMTEXT LONGTEXT	See documentation.
ENUM ('value1', 'value2', . . .)	An enumeration. Only one value, but chosen from list. See documentation.
SET ('value1', 'value2', . . .)	A set. Zero or more values, all chosen from list. See documentation.

(c) Common Data Types in MySQL

Figure 7-4
(Continued)

Amazon Ex. 1008.0268

Column Name	Type	Key	NULL Status	Remarks
ArtistID	Int	Primary Key	NOT NULL	Surrogate Key IDENTITY (1,1)
LastName	Char (25)	Alternate Key	NOT NULL	AK1
FirstName	Char (25)	Alternate Key	NOT NULL	AK1
Nationality	Char (30)	No	NULL	
DateOfBirth	Numeric (4)	No	NULL	
DateDeceased	Numeric (4)	No	NULL	

Figure 7-5
Database Column Characteristics for the ARTIST Table

Column Name	Type	Key	NULL Status	Remarks
WorkID	Int	Primary Key	NOT NULL	Surrogate Key IDENTITY (500,1)
Title	Char (35)	No	NOT NULL	
Copy	Char (12)	No	NOT NULL	
Medium	Char (35)	No	NULL	
Description	Varchar (1000)	No	NULL	DEFAULT value = 'Unknown provenance'
ArtistID	Int	Foreign Key	NOT NULL	

Figure 7-6
Database Column Characteristics for the WORK Table

The third new feature is the use of the DEFAULT column constraint in the Description column of the WORK table. The DEFAULT constraint is used to set a value that will be inserted into each row unless some other value is specified.

Figure 7-7 describes in tabular form the M-O relationship between ARTIST and WORK shown in Figure 7-1, and Figure 7-8 [based on the template in Figure 6-28(a)] details the referential integrity actions that will be needed to enforce the minimum cardinalities in the ARTIST-to-WORK relationship.

Figure 7-9 shows the SQL CREATE TABLE statement for constructing the ARTIST table. (All of the SQL in this chapter runs on SQL Server. If you are using a different DBMS, you may need to make adjustments so consult the chapter or appendix for the DBMS you are using.) The format of CREATE TABLE is the name of the table followed by a list of all column definitions and constraints enclosed in parentheses and ending with the ubiquitous SQL semicolon (;).

As stated earlier, SQL has several column and table constraints: PRIMARY KEY, NULL, NOT NULL, UNIQUE, FOREIGN KEY, and CHECK. The PRIMARY KEY constraint is used to define the primary key of the table. Although it can be used as a column constraint, because it has to be used as a table constraint to define compound primary keys, we prefer to always use it as a table constraint, as shown in Figure 7-9. The NULL and NOT NULL column constraints are used to set the NULL status of a column, indicating whether data values are required in

Relationship		Cardinality		
Parent	Child	Type	MAX	MIN
ARTIST	WORK	Nonidentifying	1:N	M-O

Figure 7-7
The ARTIST-to-WORK Relationship

Amazon Ex. 1008.0269

ARTIST Is Required Parent	Action on ARTIST (Parent)	Action on WORK (Child)
Insert	None.	Get a parent.
Modify key or foreign key	Prohibit—ARTIST uses a surrogate key.	Prohibit—ARTIST uses a surrogate key.
Delete	Prohibit—data related to a transaction is never deleted (business rule).	None.

Figure 7-8

Actions to Enforce Minimum Cardinality for the ARTIST-to-WORK Relationship

```
CREATE TABLE ARTIST(
      ArtistID            Int              NOT NULL IDENTITY(1,1),
      LastName            Char(25)         NOT NULL,
      FirstName           Char(25)         NOT NULL,
      Nationality         Char(30)         NULL,
      DateOfBirth         Numeric(4)       NULL,
      DateDeceased        Numeric(4)       NULL,
      CONSTRAINT    ArtistPK            PRIMARY KEY(ArtistID),
      CONSTRAINT    ArtistAK1           UNIQUE(LastName, FirstName)
      );
```

Figure 7-9

SQL Statements to Create the Initial Version of the ARTIST Table

that column. The UNIQUE constraint is used to indicate that the values of a column or columns must not use repeated values. The FOREIGN KEY constraint is used to define referential integrity constraints, and the CHECK constraint is used to define data constraints.

In the first section of the CREATE TABLE statement for the ARTIST table, each column is defined by giving its name, data type, and null status. If you do not specify the null status using NULL or NOT NULL, then NULL is assumed.

In this database, DateOfBirth and DateDeceased are years. YearOfBirth and YearDeceased would have been better column names, but that is not how the gallery personnel refer to them. Because the gallery is not interested in the month and day of an artist's birth and death, those columns are defined as Numeric (4,0), which means a four-digit number with zero places to the right of the decimal point.

The last two expressions in the SQL table definition statement in Figure 7-9 are constraints that define the primary key and a candidate, or alternate, key. As stated in Chapter 6, the primary purpose of an alternate key is to ensure uniqueness of column values. Thus, in SQL, alternate keys are defined using the UNIQUE constraint.

The format of such constraints is the word CONSTRAINT followed by a constraint name provided by the developer followed by either the PRIMARY KEY or UNIQUE keyword and then one or more columns in parentheses. For example, the following statement defines a constraint named *MyExample* that ensures that the combination of first and last name is unique:

```
CONSTRAINT MyExample UNIQUE (FirstName, LastName),
```

As stated in Chapter 6, primary key columns must be NOT NULL, but candidate keys can be NULL or NOT NULL.

BY THE WAY SQL originated in the era of punch-card data processing. Punched cards had only uppercase letters, so there was no need to think about case sensitivity. When cards were replaced by regular keyboards, DBMS vendors chose to ignore the difference between uppercase and lowercase letters. Thus, CREATE TABLE, create table, and CReatE taBle are all the same in SQL. NULL, null, and Null are all the same as well.

Amazon Ex. 1008.0270

Notice that the last line of the SQL statement in Figure 7-9 is a closed parenthesis followed by a semicolon. These characters could be placed on the line above, but dropping them to a new line is a style convention that makes it easy to determine the boundaries of CREATE TABLE statements. Also notice that column descriptions and constraints are separated by commas but that there is no comma after the last one.

BY THE WAY Many organizations have developed SQL coding standards of their own. Such standards specify not only the format of SQL statements, but also conventions for naming constraints. For example, in the figures in this chapter we use the suffix PK on the names of all primary key constraints and the suffix FK for all foreign key constraints. Most organizations have standards that are more comprehensive. You should follow your organization's standards, even if you disagree with them. Consistent SQL coding improves organizational efficiency and reduces errors.

Creating the WORK Table and the 1:N ARTIST-to-WORK Relationship

Figure 7-10 shows SQL statements for creating the ARTIST and WORK tables and their relationship. Note that the column name Description is written as [Description], because Description is an SQL Server 2008 reserved keyword (see Chapter 10 on SQL Server 2008), and we must use the square brackets ([and]) to create a delimited identifier. This is the same reason that in Chapter 6 we decided to use the table name TRANS instead of TRANSACTION.

The only new syntax in Figure 7-10 is the FOREIGN KEY constraint at the end of WORK. Such constraints are used to define referential integrity constraints. The FOREIGN KEY constraint in Figure 7-10 is equivalent to the following referential integrity constraint:

ArtistID in WORK must exist in ArtistID in ARTIST

Note that the foreign key constraint contains two SQL clauses that implement the minimum cardinality enforcement requirements of Figure 7-8. The **SQL ON UPDATE clause** specifies whether updates should cascade form ARTIST to WORK, and the **SQL ON DELETE clause** specifies whether deletions in ARTIST should cascade to WORK.

Figure 7-10
SQL Statements to Create The ARTIST-to-WORK 1:N Relationship

```
CREATE TABLE ARTIST(
      ArtistID            Int              NOT NULL IDENTITY(1,1),
      LastName            Char(25)         NOT NULL,
      FirstName           Char(25)         NOT NULL,
      Nationality         Char(30)         NULL,
      DateOfBirth         Numeric(4)       NULL,
      DateDeceased        Numeric(4)       NULL,
      CONSTRAINT   ArtistPK            PRIMARY KEY(ArtistID),
      CONSTRAINT   ArtistAK1           UNIQUE(LastName, FirstName)
      );

CREATE TABLE WORK(
      WorkID              Int              NOT NULL IDENTITY(500,1),
      Title               Char(35)         NOT NULL,
      Copy                Char(12)         NOT NULL,
      Medium              Char(35)         NULL,
      [Description]       Varchar(1000)    NULL DEFAULT 'Unknown provenance',
      ArtistID            Int              NOT NULL,
      CONSTRAINT   WorkPK              PRIMARY KEY(WorkID),
      CONSTRAINT   WorkAK1             UNIQUE(Title, Copy),
      CONSTRAINT   ArtistFK            FOREIGN KEY(ArtistID)
                              REFERENCES ARTIST(ArtistID)
                                 ON UPDATE NO ACTION
                                 ON DELETE NO ACTION

      );
```

Amazon Ex. 1008.0271

The expression ON UPDATE NO ACTION indicates that updates to the primary key for a table that has children should be prohibited (this is the standard setting for surrogate keys that should never change). The expression ON UPDATE CASCADE would indicate that updates should cascade. UPDATE NO ACTION is the default.

Similarly, the expression ON DELETE NO ACTION indicates that deletions of rows that have children should be prohibited. The expression ON DELETE CASCADE would indicate that deletions should cascade. DELETE NO ACTION is the default.

In the present case, the ON UPDATE NO ACTION is meaningless because the primary key of ARTIST is a surrogate and will never be changed. The ON UPDATE action would need to be specified for nonsurrogate data keys, however, and we show the option here so you will know how to code it.



Note that you must define parent tables before child tables. In this case, you must define ARTIST before WORK. If you try to reverse the order of definition, the DBMS will generate an error message on the FOREIGN KEY constraint because it will not yet know about the ARTIST table.

Similarly, you must delete tables in the opposite order. You must DROP (described later) a child before a parent. Better SQL parsers would sort out all of this so that statement order would not matter, but, alas, that's not the way it's done! Just remember the following: *Parents are first in and last out.*

Implementing Required Parent Rows

In Chapter 6, you learned that to enforce a required parent constraint you must define the referential integrity constraint and set the foreign key to NOT NULL in the child table. The SQL CREATE TABLE statement for the WORK table in Figure 7-10 does both. In this case, ARTIST is the required parent table, and WORK is the child. Thus, ArtistID in the WORK table is specified as NOT NULL (using the NOT NULL column constraint), and the ArtistFK FOREIGN KEY table constraint is used to define the referential integrity constraint. Together, these specifications thus cause the DBMS to enforce the required parent.

If the parent were not required, then we would specify ArtistID in WORK as NULL. In that case, WORK would not need to have a value for ArtistID, and thus not need a parent. However, the FOREIGN KEY constraint would still ensure that all values of ArtistID in WORK would be present in the ArtistID in ARTIST.

Implementing 1:1 Relationships

SQL for implementing 1:1 relationships is almost identical to that for 1:N relationships, as just shown. The only difference is that the foreign key must be declared as unique. For example, if the relationship were 1:1 between ARTIST and WORK (i.e., each artist could have only one work at View Ridge), then in Figure 7-10 we would add the constraint:

```
CONSTRAINT UniqueWork UNIQUE (ArtistID),
```

Note that the ARTIST-to-WORK relationship in Figure 7-1 is of course *not* 1:1, so we will *not* specify this constraint to our current SQL statements. As before, if the parent is required, then the foreign key should be set to NOT NULL. Otherwise, it should be NULL.

Casual Relationships

Sometimes it is appropriate to create a foreign key column but not specify a FOREIGN KEY constraint. In that case, the foreign key value may or may not match a value of the primary key in the parent. If, for example, you define the column DepartmentName in EMPLOYEE but do not specify a FOREIGN KEY constraint, then a row may have a value of DepartmentName that does not match a value of DepartmentName in the DEPARTMENT table.

Amazon Ex. 1008.0272

Relationship Type	CREATE TABLE Constraints
1:N relationship, parent optional	Specify FOREIGN KEY constraint. Set foreign key NULL.
1:N relationship, parent required	Specify FOREIGN KEY constraint. Set foreign key NOT NULL.
1:1 relationship, parent optional	Specify FOREIGN KEY constraint. Specify foreign key UNIQUE constraint. Set foreign key NULL.
1:1 relationship, parent required	Specify FOREIGN KEY constraint. Specify foreign key UNIQUE constraint. Set foreign key NOT NULL.
Casual relationship	Create a foreign key column, but do not specify FOREIGN KEY constraint. If relationship is 1:1, specify foreign key UNIQUE.

Figure 7-11
Summary of Relationship Definitions Using the SQL CREATE TABLE Statement

Such relationships, which we call **casual relationships**, occur frequently in applications that process tables with missing data. For example, you might buy consumer data that includes names of consumers' employers. Assume that you have an EMPLOYER table that does not contain all of the possible companies for which the consumers might work. You want to use the relationship if you happen to have the values, but you do not want to require having those values. In that case, create a casual relationship by placing the key of EMPLOYER in the consumer data table but do *not* define a FOREIGN KEY constraint.

Figure 7-11 summarizes the techniques for creating relationships using FOREIGN KEY, NULL, NOT NULL, and UNIQUE constraints in 1:N, 1:1, and casual relationships.

Creating Default Values and Data Constraints with SQL

Figure 7-12 shows an example set of default value and example data constraints for the View Ridge database. The Description column in the WORK table is given the default value of 'Unknown provenance.' The ARTIST table and TRANS tables are assigned various data constraints.

Figure 7-12
Default Values and Data Constraints for the View Ridge Database

Table	Column	Default Value	Constraint
WORK	Description	'Unknown provenance'	
ARTIST	Nationality		IN ('Candian', 'English', 'French', 'German', 'Mexican', 'Russian', 'Spainish', 'United States'.
ARTIST	DateOfBirth		Less than DateDeceased.
ARTIST	DateOfBirth		Four digits—1 or 2 is first digit, 0 to 9 for remaining three digits.
ARTIST	DateDeceased		Four digits—1 or 2 is first digit, 0 to 9 for remaining three digits.
TRANS	SalesPrice		Greater than 0 and less than or equal to 500,000.
TRANS	DateAcquired		Less than or equal to DateSold.

Amazon Ex. 1008.0273

In the ARTIST table, Nationality is limited to the values in the domain constraint shown, and DateOfBirth is limited by the **intrarelation constraint** (within the same table) that DateOfBirth occurs before DateDeceased. Further, DateOfBirth and DateDeceased, which as noted earlier are years, are limited to the domain defined by specifying that the first digit be a 1 or a 2 and the remaining three digits be any decimal numbers. Thus, they can have any value between 1000 and 2999. SalesPrice in the TRANS table is limited by a range constraint to a value greater than 0 but less than or equal to $500,000, and PurchaseDate is limited by an intrarelation constraint that the DateSold be no earlier than the DateAcquired (i.e., DateAcquired is less than or equal to DateSold).

Figure 7-12 shows no **interrelation constraints** between tables. Although the SQL-92 specification defined facilities for creating such constraints, no DBMS vendor has implemented those facilities. Such constraints must be implemented in triggers. An example of this is shown later in this chapter. Figure 7-13 shows the SQL statements to create the ARTIST and WORK tables modified with the appropriate default values and data constraints.

Implementing Default Values

Default values are created by specifying the DEFAULT keyword in the column definition just after the NULL/NOT NULL specification. Note how in Figure 7-13 the Description column in the WORK table is given the default value of 'Unknown provenance' using this technique.

Implementing Data Constraints

The data constraints are created using the SQL CHECK Constraint. The format for the CHECK constraint is the word CONSTRAINT followed by a developer-provided constraint name followed by the word CHECK and then by the constraint specification in parentheses. Expressions in CHECK constraints are akin to those used in the WHERE clause of SQL

Figure 7-13

SQL Statements to Create the ARTIST and WORK Tables with Default Values and Data Constraints

```
CREATE TABLE ARTIST(
      ArtistID            Int              NOT NULL IDENTITY(1,1),
      LastName            Char(25)         NOT NULL,
      FirstName           Char(25)         NOT NULL,
      Nationality         Char(30)         NULL,
      DateOfBirth         Numeric(4)       NULL,
      DateDeceased        Numeric(4)       NULL,
      CONSTRAINT    ArtistPK              PRIMARY KEY(ArtistID),
      CONSTRAINT    ArtistAK1             UNIQUE(LastName, FirstName),
      CONSTRAINT    NationalityValues     CHECK
                        (Nationality IN ('Canadian', 'English', 'French',
                         'German', 'Mexican', 'Russian', 'Spanish',
                         'United States')),
      CONSTRAINT    BirthValuesCheck      CHECK (DateOfBirth < DateDeceased),
      CONSTRAINT    ValidBirthYear        CHECK
                        (DateOfBirth LIKE '[1-2][0-9][0-9][0-9]'),
      CONSTRAINT    ValidDeathYear        CHECK
                        (DateDeceased LIKE '[1-2][0-9][0-9][0-9]')
      );

CREATE TABLE WORK(
      WorkID              Int              NOT NULL IDENTITY(500,1),
      Title               Char(35)         NOT NULL,
      Copy                Char(12)         NOT NULL,
      Medium              Char(35)         NULL,
      [Description]       Varchar(1000)    NULL DEFAULT 'Unknown provenance',
      ArtistID            Int              NOT NULL,
      CONSTRAINT    WorkPK                PRIMARY KEY(WorkID),
      CONSTRAINT    WorkAK1               UNIQUE(Title, Copy),
      CONSTRAINT    ArtistFK              FOREIGN KEY(ArtistID)
                        REFERENCES ARTIST(ArtistID)
                           ON UPDATE NO ACTION
                           ON DELETE NO ACTION
      );
```

Amazon Ex. 1008.0274

statements. Thus, the SQL IN keyword is used to provide a list of valid values. The SQL NOT IN keyword also can be used for negatively expressed domain constraints (not shown in this example). The SQL LIKE keyword is used for the specification of decimal places. Range checks are specified using comparison operators such as the less than (<) and greater than (>) symbols. Because interrelation constraints are unsupported, comparisons can be made as intrarelation constraints between columns in the same table.



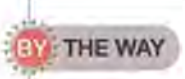
DBMS products are inconsistent in their implementation of CHECK constraints. The use of the SQL LIKE keyword in Figure 7-13, for example, will not work with Oracle. However, Oracle implements other types of constraints. Unfortunately, you must learn the peculiarities of the DBMS you use to know how best to implement constraints.

Creating the View Ridge Database Tables

Figure 7-14 shows SQL for creating all of the tables in the View Ridge database documented at the end of Chapter 6. Read each line and be certain that you understand its function and purpose. Notice that deletions cascade for the relationships between CUSTOMER and CUSTOMER_ARTIST_INT and between ARTIST and CUSTOMER_ARTIST_INT.

Any DBMS reserved words used as table or column names need to be enclosed in square brackets ([and]), and thus converted to delimited identifiers. We have already decided to use the table name TRANS instead of TRANSACTION so that we do not use the *transaction* reserved word. The table name WORK is also a potential problem; the word *work* is a reserved word in most DBMS products, as are the column names *Description* in the WORK table and *State* in the TRANS table. Enclosing such terms in brackets signifies to the SQL parser that these terms have been provided by the developer and are not to be used in the standard way. Ironically, SQL Server can process the word WORK without problem, but Oracle cannot, while SQL Server chokes on the word TRANSACTION, but Oracle has no problem with it. Because Figure 7-14 shows SQL Server 2008 T-SQL statements, we use WORK (no brackets), [Description], and [State].

You can find a list of reserved words in the documentation for the DBMS product that you use, and we deal with some specific cases in the chapters dedicated to SQL Server 2008, Oracle, and MySQL. Be assured that if you use any keyword from the SQL syntax, such as SELECT, FROM, WHERE, LIKE, ORDER, ASC, DESC, for table or column names, you will have problems. Enclose such words in square brackets. And, of course, your life will be easier if you can avoid using such terms for tables or columns altogether.



Every now and then, the DBMS might generate bizarre syntax-error messages. For example, suppose you define a table with the name ORDER. When you submit the statement SELECT * FROM ORDER, you will get very strange messages back from the DBMS.

If you do receive odd messages back from statements that you know are coded correctly, think about reserved words. If a term might be reserved, enclose it in brackets and see what happens when you submit it to the DBMS. No harm is done by enclosing SQL terms in brackets.

If you want to torture your DBMS, you can submit queries like SELECT [SELECT] FROM [FROM] WHERE [WHERE] < [NOT FIVE]. Most likely, you have better ways to spend your time, however. Without a doubt, the DBMS has better ways to spend its time!

Running the SQL statements in Figure 7-14 (or a specific variant in Chapter 10A for Oracle or Chapter 10B for MySQL) with your DBMS will generate all of the tables, relationships, and constraints for the View Ridge database. Figure 7-15 shows the completed table structure in SQL Server 2008 as a database diagram. It is far easier to create these tables and relationships using SQL code than by using GUI displays, which are discussed in Chapter 10 (SQL Server 2008), Chapter 10A (Oracle Database 11g), and Chapter 10B (MySQL 5.1).

Amazon Ex. 1008.0275

```
CREATE TABLE ARTIST(
      ArtistID            Int              NOT NULL IDENTITY(1,1),
      LastName            Char(25)         NOT NULL,
      FirstName           Char(25)         NOT NULL,
      Nationality         Char(30)         NULL,
      DateOfBirth         Numeric(4)       NULL,
      DateDeceased        Numeric(4)       NULL,
      CONSTRAINT    ArtistPK              PRIMARY KEY(ArtistID),
      CONSTRAINT    ArtistAK1             UNIQUE(LastName, FirstName),
      CONSTRAINT    NationalityValues     CHECK
                    (Nationality IN ('Canadian', 'English', 'French',
                     'German', 'Mexican', 'Russian', 'Spanish',
                     'United States')),
      CONSTRAINT    BirthValuesCheck      CHECK (DateOfBirth < DateDeceased),
      CONSTRAINT    ValidBirthYear        CHECK
                    (DateOfBirth LIKE '[1-2][0-9][0-9][0-9]'),
      CONSTRAINT    ValidDeathYear        CHECK
                    (DateDeceased LIKE '[1-2][0-9][0-9][0-9]')
      );

CREATE TABLE WORK(
      WorkID              Int              NOT NULL IDENTITY(500,1),
      Title               Char(35)         NOT NULL,
      Copy                Char(12)         NOT NULL,
      Medium              Char(35)         NULL,
      [Description]       Varchar(1000)    NULL DEFAULT 'Unknown provenance',
      ArtistID            Int              NOT NULL,
      CONSTRAINT    WorkPK                PRIMARY KEY(WorkID),
      CONSTRAINT    WorkAK1               UNIQUE(Title, Copy),
      CONSTRAINT    ArtistFK              FOREIGN KEY(ArtistID)
                       REFERENCES ARTIST(ArtistID)
                          ON UPDATE NO ACTION
                          ON DELETE NO ACTION
      );

CREATE TABLE CUSTOMER(
      CustomerID          Int              NOT NULL IDENTITY(1000,1),
      LastName            Char(25)         NOT NULL,
      FirstName           Char(25)         NOT NULL,
      Street              Char(30)         NULL,
      City                Char(35)         NULL,
      [State]             Char(2)          NULL,
      ZipPostalCode       Char(9)          NULL,
      Country             Char(50)         NULL,
      AreaCode            Char(3)          NULL,
      PhoneNumber         Char(8)          NULL,
      Email               Varchar(100)     Null,
      CONSTRAINT    CustomerPK            PRIMARY KEY(CustomerID),
      CONSTRAINT    EmailAK1              UNIQUE(Email)
      );
```

Figure 7-14

SQL Statements to Create the View Ridge Database Table Structure



Does Not Work with MS Access ANSI-89 SQL

Microsoft Access 2007 ANSI-89 SQL, unfortunately, does not support a number of standard SQL features we have examined in this discussion. However, you *can* run a basic SQL CREATE TABLE statement in ANSI-89 SQL, and then use the Access GUI display to finish building the tables and relationships. Specifically:

1. Although Microsoft Access supports a Number data type, it does not support the (*m, n*) extension to specify the number of digits and the number of digits to the right of the decimal place.

   Solution: You can set these values in the table Design view after the column is created.

(*continued*)

Amazon Ex. 1008.0276

```
CREATE TABLE TRANS(
      TransactionID        Int              NOT NULL IDENTITY(100,1),
      DateAcquired         Datetime         NOT NULL,
      AcquisitionPrice     Numeric(8,2)     NOT NULL,
      DateSold             Datetime         NULL,
      AskingPrice          Numeric(8,2)     NULL,
      SalesPrice           Numeric(8,2)     NULL,
      CustomerID           Int              NULL,
      WorkID               Int              NOT NULL,
      CONSTRAINT    TransPK              PRIMARY KEY(TransactionID),
      CONSTRAINT    TransWorkFK          FOREIGN KEY(WorkID)
                          REFERENCES WORK(WorkID)
                             ON UPDATE NO ACTION
                             ON DELETE NO ACTION,
      CONSTRAINT    TransCustomerFK      FOREIGN KEY(CustomerID)
                          REFERENCES CUSTOMER(CustomerID)
                             ON UPDATE NO ACTION
                             ON DELETE NO ACTION,
      CONSTRAINT    SalesPriceRange  CHECK
                          ((SalesPrice > 0) AND (SalesPrice <=500000)),
      CONSTRAINT    ValidTransDate       CHECK (DateAcquired <= DateSold),
      );

CREATE TABLE CUSTOMER_ARTIST_INT(
      ArtistID             Int              NOT NULL,
      CustomerID           Int              NOT NULL,
      CONSTRAINT    CAIntPK              PRIMARY KEY(ArtistID, CustomerID),
      CONSTRAINT    CAInt_ArtistFK       FOREIGN KEY(ArtistID)
                          REFERENCES ARTIST(ArtistID)
                             ON UPDATE NO ACTION
                             ON DELETE CASCADE,
      CONSTRAINT    CAInt_CustomerFK     FOREIGN KEY(CustomerID)
                          REFERENCES CUSTOMER(CustomerID)
                             on UPDATE NO ACTION
                             ON DELETE CASCADE
      );
```

Figure 7-14 (Continued)

2. Although Microsoft Access does support an AutoNumber data type, it *always* starts at 1 and increments by 1. Further, AutoNumber *cannot* be used as an SQL data type.

   Solution: Set AutoNumber data type manually after the table is created. Any other numbering system must be supported manually or by application code.

3. Microsoft Access ANSI-89 SQL does not support the UNIQUE and CHECK column constraints, nor the DEFAULT keyword.

   Solution: Equivalent constraints and initial values can be set in the GUI table Design view.

4. Microsoft Access does completely support foreign key CONSTRAINT phrases. While the basic referential integrity constraint can be created using SQL, the ON UPDATE and ON DELETE clauses are not supported.

   Solution: ON UPDATE and ON DELETE actions can be set manually after the relationship is created.

5. Unlike SQL Server, Oracle, and MySQL, Microsoft Access does not support SQL scripts.

   Solution: You can still create tables by using the SQL CREATE command and insert data by using the SQL INSERT command (discussed later in this chapter), but you must do so one command at a time.

Amazon Ex. 1008.0277

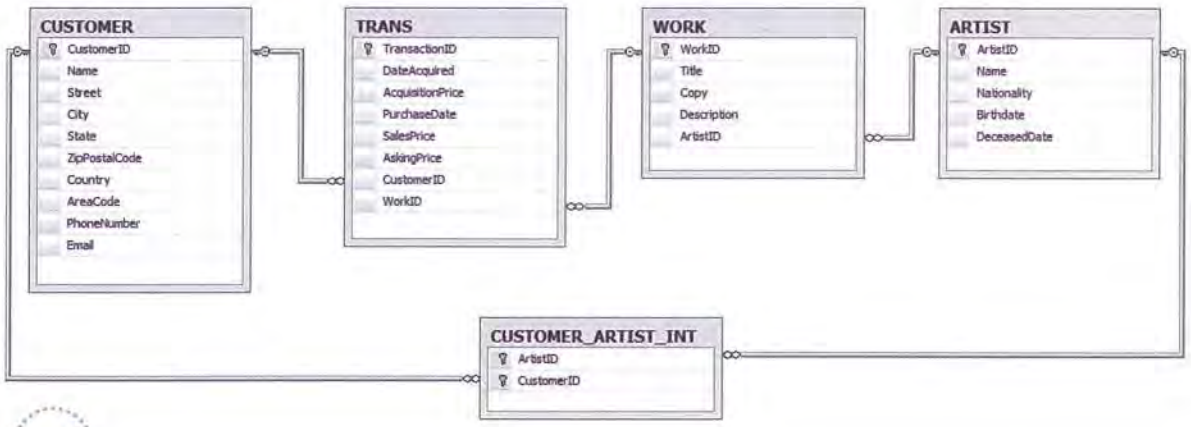


Figure 7-15
SQL Server 2008 View Ridge Database Diagram

The SQL ALTER Statement

The **SQL ALTER statement** is an SQL DDL statement that is used to change the structure of an existing table. It can be used to add, remove, or change columns. It also can be used to add or remove constraints.

Adding and Dropping Columns

The following statement will add a column named MyColumn to the CUSTOMER table:

```
ALTER TABLE CUSTOMER ADD MyColumn Char(5) NULL;
```

You can drop an existing column with the statement:

```
ALTER TABLE CUSTOMER DROP COLUMN MyColumn;
```

Note the asymmetry in syntax; the keyword COLUMN is used with DROP but not with ADD. You can also use the ALTER statement to change column properties as well, as you will see in the next three chapters.

Adding and Dropping Constraints

The ALTER statement can be used to add a constraint as follows:

```
ALTER TABLE CUSTOMER ADD CONSTRAINT MyConstraint CHECK
    (LastName NOT IN ('RobertsNoPay'));
```

You can also use the ALTER statement to DROP a constraint:

```
ALTER TABLE CUSTOMER DROP CONSTRAINT MyConstraint;
```

BY THE WAY The SQL ALTER statement can be used to add or drop any of the SQL constraints. You can use it to create primary keys and alternate keys, to set null status, to create referential integrity constraints, and to create data constraints. In fact, another SQL coding style uses CREATE TABLE only to declare the table's columns; all constraints are added using ALTER. We do not use that style in this text, but be aware that it does exist and that your employer might require it.

Amazon Ex. 1008.0278

The SQL DROP TABLE Statement

It is very easy to remove a table in SQL. In fact, it is far too easy. The following **SQL DROP TABLE statement** will drop the TRANS table *and all of its data:*

```
DROP TABLE TRANS;
```

Because this simple statement drops the table and all of its data, be very careful when using it. Do not code this statement on the wrong table!

The DBMS will not drop a table that is the parent in a FOREIGN KEY constraint. It will not do so even if there are no children, or even if you have coded DELETE CASCADE. Instead, to drop such a table, you must first either drop the foreign key constraint or drop the child table. Then you can delete the parent table. As mentioned earlier, parent tables must be first in and last out.

The following statements are needed to drop the CUSTOMER table:

```
DROP TABLE CUSTOMER_ARTIST_INT;
DROP TABLE TRANS;
DROP TABLE CUSTOMER;
```

Alternatively, you could drop CUSTOMER with:

```
ALTER TABLE CUSTOMER_ARTIST_INT
    DROP CONSTRAINT Customer_Artist_Int_CustomerFK;
ALTER TABLE TRANS
    DROP CONSTRAINT TransactionCustomerFK;
DROP TABLE CUSTOMER;
```

SQL DML Statements

At this point, you have learned how to query tables using SQL SELECT statements (in Chapter 2), and you know how to create, alter, and drop tables, columns, and constraints. You do not yet know, however, how to use SQL statements to insert, modify, and delete data. We consider those statements next.

The SQL INSERT Statement

The **SQL INSERT statement** is used to add rows of data to a table. The statement has a number of different options.

The SQL INSERT Statement Using Column Names

The standard version of the INSERT statement is to name the table, name the columns for which you have data, and then list the data in the following format:

```
INSERT INTO ARTIST
    (LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
    VALUES ('Miro', 'Joan', 'Spanish', 1893, 1983);
```

Note that both column names and values are enclosed in parenthesis, and DBMS populated surrogate keys are not included in the statement. If you are providing data for all of the columns, if that data is in the same order as the columns in the table, and if you have no surrogate keys, then you can omit the column list.

```
INSERT INTO ARTIST VALUES ('Miro', 'Joan', 'Spanish', 1893, 1983);
```

Amazon Ex. 1008.0279

Further, you need not provide the values in the same order as the columns in the table. If for some reason you want to provide Nationality first, you can revise the column names and the data value, as shown in the following example:

```
INSERT INTO ARTIST
    (Nationality, LastName, FirstName, DateOfBirth, DateDeceased)
    VALUES ('Spanish', 'Miro', 'Joan', 1893, 1983);
```

If you have partial values, just code the names of the columns for which you have data. For example, if you have only LastName, FirstName, and Nationality for an artist, you would use the SQL statement:

```
INSERT INTO ARTIST
    (LastName, FirstName, Nationality)
    VALUES ('Miro', 'Joan', 'Spanish');
```

You must, of course, have values for all NOT NULL columns.

Bulk INSERT

One of the most often used forms of INSERT uses an SQL SELECT statement to provide values. Suppose you have the names, nationalities, birth dates, and dates deceased of a number of artists in a table named IMPORTED_ARTIST. In this case, you can add those data to the ARTIST table with the following statement:

```
INSERT INTO ARTIST
  (LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
  SELECT   (LastName, FirstName, Nationality, DateOfBirth,
           DateDeceased)
  FROM     IMPORTED_ARTIST;
```

Note that the SQL keyword VALUES is not used with this form of insert.

This syntax should seem familiar. We used it for normalization and denormalization examples in Chapters 3 and 4.

Populating the View Ridge Database Tables

Now that we know how to use the SQL INSERT statement to add rows of data to a table, we can put data into the View Ridge database. Sample data for the View Ridge database is shown in Figure 7-16.

However, we need to be careful about exactly how we enter this data into the View Ridge database. Notice that in the SQL CREATE TABLE statements in Figure 7-14 that CustomerID, ArtistID, WorkID, and TransactionID are all surrogate keys with values automatically inserted by the DBMS. This will produce sequential numbers. For example, if we insert the ARTIST table data shown in Figure 7-16(b) using the automatic ArtistID numbering from IDENTITY(1, 1), the ArtistID numbers for the 9 artist will be (1, 2, 3, 4, 5, 6, 7, 8, 9). But in Figure 7-12(b) the ArtistID numbers are (1, 2, 3, 4, 5, 11, 17, 18, 19).

This happens because the View Ridge data shown in Figure 7-16 is sample data, not the complete data for the database. Therefore, the primary key numbers for CustomerID, ArtistID, WorkID, and TransactionID are *not* sequential.

This, of course, raises the question of how to override DBMS mechanisms that provide automatic surrogate key numbering. The answer to this question varies between DBMS products (as does the method for generating the surrogate values). A discussion of this topic and the complete set of SQL INSERT statements needed to enter the data for the DBMS products discussed in the text are in Chapters 10 (SQL Server 2008), 10A (Oracle Database 11g), and 10B (MySQL 5.1). At this point, we recommend that you read the appropriate section for the DBMS product you are using and populate the View Ridge database in your DBMS.

Amazon Ex. 1008.0280

CustomerID	LastName	FirstName	Street	City	State	ZipPostalCode
1000	Janes	Jeffrey	123 W. Elm St	Renton	WA	98055
1001	Smith	David	813 Tumbleweed Lane	Durango	CO	81201
1015	Twilight	Tiffany	88 1st Avenue	Langley	WA	98260
1033	Smathers	Fred	10899 88th Ave	Bainbridge Island	WA	98110
1034	Frederickson	Mary Beth	25 South Lafayette	Denver	CO	80201
1036	Warning	Selma	205 Burnaby	Vancouver	BC	V6Z 1W2
1037	Wu	Susan	105 Locust Ave	Atlanta	GA	30322
1040	Gray	Donald	55 Bodega Ave	Bodega Bay	CA	94923
1041	Johnson	Lynda	117 C Street	Washington	DC	20003
1051	Wilkens	Chris	87 Highland Drive	Olympia	WA	98508

CustomerID	LastName	FirstName	Country	AreaCode	PhoneNumber	Email
1000	Janes	Jeffrey	USA	425	543-2345	Jeffrey.James@somewhere.com
1001	Smith	David	USA	970	654-9876	David.Smith@somewhere.com
1015	Twilight	Tiffany	USA	360	765-5566	Tiffany.Twilight@somewhere.com
1033	Smathers	Fred	USA	206	876-9911	Fred.Smathers@somewhere.com
1034	Frederickson	Mary Beth	USA	303	513-8822	MaryBeth.Frederickson@somewhere.com
1036	Warning	Selma	Canada	604	988-0512	Selma.Warning@somewhere.com
1037	Wu	Susan	USA	404	653-3465	Susan.Wu@somewhere.com
1040	Gray	Donald	USA	707	568-4839	Donald.Gray@somewhere.com
1041	Johnson	Lynda	USA	202	438-5498	NULL
1051	Wilkens	Chris	USA	360	765-7766	Chris.Wilkens@somewhere.com

(a) CUSTOMER Table Data

Figure 7-16
Sample Data for the View Ridge Database

The SQL UPDATE Statement

The **SQL UPDATE statement** is used to change values of existing rows. For example, the following statement will change the value of City to 'New York City' for the View Ridge customer whose CustomerID is 1000 (Jeffrey Janes):

```
UPDATE      CUSTOMER
SET         City = 'New York City'
WHERE       CustomerID = 1000;
```

Amazon Ex. 1008.0281

ArtistID	LastName	FirstName	Nationality	DateOfBirth	DateDeceased
1	Miro	Joan	Spanish	1893	1983
2	Kandinsky	Wassily	Russian	1866	1944
3	Klee	Paul	German	1879	1940
4	Matisse	Henri	French	1869	1954
5	Chagall	Marc	French	1887	1985
11	Sargent	John Singer	United States	1856	1925
17	Tobey	Mark	United States	1890	1976
18	Horiuchi	Paul	United States	1906	1999
19	Graves	Morris	United States	1920	2001

(b) ARTIST Table Data

ArtistID	CustomerID
1	1001
1	1034
2	1001
2	1034
4	1001
4	1034
5	1001
5	1034
5	1036
11	1001
11	1015
11	1036
17	1000
17	1015

ArtistID	CustomerID
17	1033
17	1040
17	1051
18	1000
18	1015
18	1033
18	1040
18	1051
19	1000
19	1015
19	1033
19	1036
19	1040
19	1051

(c) CUSTOMER_ARTIST_INT Table Data

Figure 7-16
(Continued)

Amazon Ex. 1008.0282

WorkID	Title	Medium	Description	Copy	ArtistID
500	Memories IV	Casein rice paper collage	31 x 24.8 in.	Unique	18
511	Surf and Bird	High Quality Limited Print	Northwest School Expressionist style	142/500	19
521	The Tilled Field	High Quality Limited Print	Early Surrealist style	788/1000	1
522	La Lecon de Ski	High Quality Limited Print	Surrealist style	353/500	1
523	On White II	High Quality Limited Print	Bauhaus style of Kandinsky	435/500	2
524	Woman with a Hat	High Quality Limited Print	A very colorful Impressionist piece	596/750	4
537	The Woven World	Color lithograph	Signed	17/750	17
548	Night Bird	Watercolor on Paper	50 x 72.5 cm.—Signed	Unique	19
551	Der Blaue Reiter	High Quality Limited Print	"The Blue Rider"—Early Pointilism influence	236/1000	2
552	Angelus Novus	High Quality Limited Print	Bauhaus style of Klee	659/750	3
553	The Dance	High Quality Limited Print	An Impressionist masterpiece	734/1000	4
554	I and the Village	High Quality Limited Print	Shows Belarusian folk-life themes and symbology	834/1000	5
555	Claude Monet Painting	High Quality Limited Print	Shows French Impressionist influence of Monet	684/1000	11
561	Sunflower	Watercolor and ink	33.3 x 16.1 cm.—Signed	Unique	19
562	The Fiddler	High Quality Limited Print	Shows Belarusian folk-life themes and symbology	251/1000	5
563	Spanish Dancer	High Quality Limited Print	American realist style—From work in Spain	583/750	11
564	Farmer's Market #2	High Quality Limited Print	Northwest School Abstract Expressionist style	267/500	17

Figure 7-16
(Continued)

Amazon Ex. 1008.0283

565	Farmer's Market #2	High Quality Limited Print	Northwest School Abstract Expressionist style	268/500	17
566	Into Time	High Quality Limited Print	Northwest School Abstract Expressionist style	323/500	18
570	Untitled Number 1	Monotype with tempera	4.3 x 6.1 in.—Signed	Unique	17
571	Yellow covers blue	Oil and collage	71 x 78 in.—Signed	Unique	18
578	Mid Century Hibernation	High Quality Limited Print	Northwest School Expressionist style	362/500	19
580	Forms in Progress I	Color aquatint	19.3 x 24.4 in.—Signed	Unique	17
581	Forms in Progress II	Color aquatint	19.3 x 24.4 in.—Signed	Unique	17
585	The Fiddler	High Quality Limited Print	Shows Belarusian folk-life themes and symbology	252/1000	5
586	Spanish Dancer	High Quality Limited Print	American Realist style—From work in Spain	588/750	11
587	Broadway Boggie	High Quality Limited Print	Northwest School Abstract Expressionist style	433/500	17
588	Universal Field	High Quality Limited Print	Northwest School Abstract Expressionist style	114/500	17
589	Color Foating in Time	High Quality Limited Print	Northwest School Abstract Expressionist style	487/500	18
590	Blue Interior	Tempera on card	43.9 x 28 in.	Unique	17
593	Surf and Bird	Gouache	26.5 x 29.75 in.—Signed	Unique	19
594	Surf and Bird	High Quality Limited Print	Northwest School Expressionist style	366/500	19
595	Surf and Bird	High Quality Limited Print	Northwest School Expressionist style	366/500	19
596	Surf and Bird	High Quality Limited Print	Northwest School Expressionist style	366/500	19

(d) WORK Table Data

Figure 7-16

(Continued)

Amazon Ex. 1008.0284

TransactionID	DateAcquired	AcquisitionPrice	AskingPrice	DateSoldID	SalesPrice	CustomerID	WorkID
100	11/4/2005	$30,000.00	$45,000.00	12/14/2005	$42,500.00	1000	500
101	11/7/2005	$250.00	$500.00	12/19/2005	$500.00	1015	511
102	11/17/2005	$125.00	$250.00	1/18/2006	$200.00	1001	521
103	11/17/2005	$250.00	$500.00	12/12/2006	$400.00	1034	522
104	11/17/2005	$250.00	$250.00	1/18/2006	$200.00	1001	523
105	11/17/2005	$200.00	$500.00	12/12/2006	$400.00	1034	524
115	3/3/2006	$1,500.00	$3,000.00	6/7/2006	$2,750.00	1033	537
121	9/21/2006	$15,000.00	$30,000.00	11/28/2006	$27,500.00	1015	548
125	11/21/2006	$125.00	$250.00	12/18/2006	$200.00	1001	551
126	11/21/2006	$200.00	$400.00				552
127	11/21/2006	$125.00	$500.00	12/22/2006	$400.00	1034	553
128	11/21/2006	$125.00	$250.00	3/16/2007	$225.00	1036	554
129	11/21/2006	$125.00	$250.00	3/16/2007	$225.00	1036	555
151	5/7/2007	$10,000.00	$20,000.00	6/28/2007	$17,500.00	1036	561
152	5/18/2007	$125.00	$250.00	8/15/2007	$225.00	1001	562
153	5/18/2007	$200.00	$400.00	8/15/2007	$350.00	1001	563
154	5/18/2007	$250.00	$500.00	9/28/2007	$400.00	1040	564
155	5/18/2007	$250.00	$500.00				565
156	5/18/2007	$250.00	$500.00	9/27/2007	$400.00	1040	566
161	6/28/2007	$7,500.00	$15,000.00	9/29/2007	$13,750.00	1033	570
171	8/23/2007	$35,000.00	$60,000.00	9/29/2007	$55,000.00	1000	571
175	9/29/2007	$40,000.00	$75,000.00	12/18/2007	$72,500.00	1036	500
181	10/11/2007	$250.00	$500.00				578
201	2/28/2008	$2,000.00	$3,500.00	4/26/2008	$3,250.00	1040	580
202	2/28/2008	$2,000.00	$3,500.00	4/26/2008	$3,250.00	1040	581
225	6/8/2008	$125.00	$250.00	9/27/2008	$225.00	1051	585
226	6/8/2008	$200.00	$400.00				586
227	6/8/2008	$250.00	$500.00	9/27/2008	$475.00	1051	587
228	6/8/2008	$250.00	$500.00				588
229	6/8/2008	$250.00	$500.00				589
241	8/29/2008	$2,500.00	$5,000.00	9/27/2008	$4,750.00	1015	590
251	10/25/2008	$25,000.00	$50,000.00				593
252	10/27/2008	$250.00	$500.00				594
253	10/27/2008	$250.00	$500.00				595
254	10/27/2008	$250.00	$500.00				596

(e) TRANS Table Data

Figure 7-16
(Continued)

Amazon Ex. 1008.0285

To change the value of both City and State, we would use the SQL statement:

```
UPDATE      CUSTOMER
SET         City = 'New York City', State = 'NY'
WHERE       CustomerID = 1000;
```

The DBMS will enforce all referential integrity constraints when processing UPDATE commands. For the View Ridge database, all keys are surrogate keys, but for tables with data keys the DBMS will cascade or disallow (NO ACTION) updates according to the specification in the FOREIGN KEY constraint. Also, if there is a FOREIGN KEY constraint, the DBMS will enforce the referential integrity constraint on updates to a foreign key.

Bulk Updates

It is quite easy to make bulk updates with the UPDATE statement. It is so easy, in fact, that it is dangerous. Consider the SQL UPDATE statement:

```
UPDATE      CUSTOMER
SET         City = 'New York City';
```

This statement will change the value of City for every row of the CUSTOMER table. If we had intended to change just the value for customer 1000, we would have an unhappy result—every customer would have the value 'New York City'.

You can also perform bulk updates using an SQL WHERE clause that finds multiple rows. If, for example, we wanted to change the AreaCode for every customer who lives in Denver, we would code:

```
UPDATE      CUSTOMER
SET         AreaCode = '303'
WHERE       City = 'Denver';
```

Updating Using Values from Other Tables

The SQL UPDATE statement can set a column equal to the value of a column in a different table. The View Ridge database has no appropriate example for this operation, so suppose instead that we have a table named TAX_TABLE with columns (Tax, City), where Tax is the appropriate tax rate for the City.

Now suppose we have a table named PURCHASE_ORDER that includes the columns TaxRate and City. We can update all rows for purchase orders in the city of Bodega Bay with the following SQL statement:

```
UPDATE      PURCHASE_ORDER
SET         TaxRate =
            (SELECT     Tax
             FROM       TAX_TABLE
             WHERE      TAX_TABLE.City = 'Bodega Bay')
WHERE       PURCHASE_ORDER.City = 'Bodega Bay';
```

Amazon Ex. 1008.0286

More likely, we want to update the value of the tax rate for a purchase order without specifying the city. Say we want to update the TaxRate for purchase order number 1000. In that case, we use the slightly more complex SQL statement:

```
UPDATE      PURCHASE_ORDER
SET         TaxRate =
            (SELECT     Tax
             FROM       TAX_TABLE
             WHERE      TAX_TABLE.City = PURCHASE_ORDER.City)
WHERE       PURCHASE_ORDER.Number = 1000;
```

SQL SELECT statements can be combined with UPDATE statements in many different ways. We need to move on to other topics, but try these and other variations of UPDATE on your own.

The SQL DELETE Statement

The **SQL DELETE statement** is also quite easy to use. The following SQL statement will delete the row for a customer with a CustomerID of 1000:

```
DELETE      FROM CUSTOMER
WHERE       CustomerID = 1000;
```

Of course, if you omit the WHERE clause, you will delete *every* customer row, so be careful with this command as well.

The DBMS will enforce all referential integrity constraints when processing DELETE commands. For example, in the View Ridge database, you will be unable to delete a CUSTOMER row if that row has any TRANS children. Further, if a row with no TRANS children is deleted, any existing CUSTOMER_ARTIST_INT children will be deleted as well. This latter action occurs because of the CASCADE DELETE specification on the relationship between CUSTOMER and CUSTOMER_ARTIST_INT.

New Forms of Join

In Chapter 2, you learned how to perform SQL joins. Here we extend that discussion to show a different join syntax and to address ways of processing joins on tables with null values. We will be using the View Ridge database and the data in Figure 7-16 as our example, so if you want to try out these commands, you will need to have completed building and populating your VRG database based on the SQL CREATE TABLE statements and INSERT statements appropriate for your DBMS product.

The SQL JOIN ON Syntax

In Chapter 2, you learned to code joins using the following syntax:

```
SELECT      *
FROM        ARTIST, WORK
WHERE       ARTIST.ArtistID = WORK.ArtistID;
```

Another way to code this same join is:

```
SELECT      *
FROM        ARTIST JOIN WORK
     ON     ARTIST.ArtistID = WORK.ArtistID;
```

Amazon Ex. 1008.0287

These two joins are equivalent. Some people think that the second format, which uses the **SQL JOIN ON syntax** is easier to understand than the first.

You can use the SQL JOIN ON syntax as an alternate format for joins of three or more tables, as well. If, for example, you want to obtain a list of the names of customers and the names of the artists in which they are interested, code:

```
SELECT      CUSTOMER.LastName, CUSTOMER.FirstName,
            ARTIST.LastName AS ArtistName
FROM        CUSTOMER JOIN CUSTOMER_ARTIST_INT
     ON     CUSTOMER.CustomerID = CUSTOMER_ARTIST_INT.CustomerID
            JOIN    ARTIST
               ON   CUSTOMER_ARTIST_INT.ArtistID = ARTIST.ArtistID;
```

You can make that statement even simpler by using the **SQL AS keyword** to create table aliases as well as for naming output columns:

```
SELECT      C.LastName, C.FirstName,
            A.LastName AS ArtistName
FROM        CUSTOMER AS C JOIN CUSTOMER_ARTIST_INT AS CI
     ON     C.CustomerID = CI.CustomerID
            JOIN    ARTIST AS A
               ON   CI.ArtistID = A.ArtistID;
```

When a query produces a result table with many rows, we may want to limit the number of rows that we see. We can do this using the **SQL TOP NumberOfRows syntax**, which, along with an ORDER BY clause to sort the data, produces our final SQL query statement:

```
SELECT      TOP 10 C.LastName, C.FirstName,
            A.LastName AS ArtistName
FROM        CUSTOMER AS C JOIN CUSTOMER_ARTIST_INT AS CI
     ON     C.CustomerID = CI.CustomerID
            JOIN    ARTIST AS A
               ON  CI.ArtistID = A.ArtistID
ORDER BY    C.LastName, C.FirstName;
```

The result of this statement is:

	LastName	FirstName	ArtistName
1	Frederickson	Mary Beth	Chagall
2	Frederickson	Mary Beth	Kandinsky
3	Frederickson	Mary Beth	Matisse
4	Frederickson	Mary Beth	Miro
5	Gray	Donald	Graves
6	Gray	Donald	Horiuchi
7	Gray	Donald	Tobey
8	Janes	Jeffrey	Graves
9	Janes	Jeffrey	Horiuchi
10	Janes	Jeffrey	Tobey

Amazon Ex. 1008.0288

Outer Joins

The SQL query:

```
SELECT      C.LastName, C.FirstName, T.TransactionID, T.SalesPrice
FROM        CUSTOMER AS C JOIN TRANS AS T
      ON    C.CustomerID = T.CustomerID
ORDER BY    T.TransactionID;
```

produces the large result set shown in Figure 7-17.

This result is correct, but it only shows the names of only 8 of the 10 customers in the CUSTOMER table. What happened to the other two customers? Look closely at the data in Figure 7-16 and you will see that the two customers that do not appear in the results (Susan Wu and Lynda Johnson) are customers who have never made a purchase at the gallery. Therefore, the primary key value of these two customers does not match any foreign key value in the TRANS table, and because they have no match, they do not appear in the result of this join statement.

We can cause all of the rows in CUSTOMER to appear using what is called an **SQL outer join**. The SQL outer join syntax is shown in the query:

```
SELECT      C.LastName, C.FirstName, T.TransactionID, T.SalesPrice
FROM        CUSTOMER AS C LEFT JOIN TRANS AS T
      ON    C.CustomerID = T.CustomerID
ORDER BY    T.TransactionID;
```

Figure 7-17

Result of Join of CUSTOMER and TRANS

	LastName	FirstName	TransactionID	SalesPrice
1	Janes	Jeffrey	100	42500.00
2	Twilight	Tiffany	101	500.00
3	Smith	David	102	200.00
4	Frederickson	Mary Beth	103	400.00
5	Smith	David	104	200.00
6	Frederickson	Mary Beth	105	400.00
7	Smathers	Fred	115	2750.00
8	Twilight	Tiffany	121	27500.00
9	Smith	David	125	200.00
10	Frederickson	Mary Beth	127	400.00
11	Warning	Selma	128	225.00
12	Warning	Selma	129	225.00
13	Warning	Selma	151	17500.00
14	Smith	David	152	225.00
15	Smith	David	153	350.00
16	Gray	Donald	154	400.00
17	Gray	Donald	156	400.00
18	Smathers	Fred	161	13750.00
19	Janes	Jeffrey	171	55000.00
20	Warning	Selma	175	72500.00
21	Gray	Donald	201	3250.00
22	Gray	Donald	202	3250.00
23	Wilkens	Chris	225	225.00
24	Wilkens	Chris	227	475.00
25	Twilight	Tiffany	241	4750.00

Amazon Ex. 1008.0289

	LastName	FirstName	TransactionID	SalesPrice
1	Wu	Susan	NULL	NULL
2	Johnson	Lynda	NULL	NULL
3	Janes	Jeffrey	100	42500.00
4	Twilight	Tiffany	101	500.00
5	Smith	David	102	200.00
6	Frederickson	Mary Beth	103	400.00
7	Smith	David	104	200.00
8	Frederickson	Mary Beth	105	400.00
9	Smathers	Fred	115	2750.00
10	Twilight	Tiffany	121	27500.00
11	Smith	David	125	200.00
12	Frederickson	Mary Beth	127	400.00
13	Warning	Selma	128	225.00
14	Warning	Selma	129	225.00
15	Warning	Selma	151	17500.00
16	Smith	David	152	225.00
17	Smith	David	153	350.00
18	Gray	Donald	154	400.00
19	Gray	Donald	156	400.00
20	Smathers	Fred	161	13750.00
21	Janes	Jeffrey	171	55000.00
22	Warning	Selma	175	72500.00
23	Gray	Donald	201	3250.00
24	Gray	Donald	202	3250.00
25	Wilkens	Chris	225	225.00
26	Wilkens	Chris	227	475.00
27	Twilight	Tiffany	241	4750.00

Figure 7-18
Result of Left Outer Join of CUSTOMER and TRANS

The result of this query is shown in Figure 7-18. Notice that the values of TransactionID and SalesPrice are NULL for all customers (Susan Wu and Lynda Johnson) who have not made purchases.

Outer joins can be either from the left or the right. If the outer join is from the left, an **SQL left outer join which uses the SQL LEFT JOIN syntax**, then all of the rows on the table on the left (or first table in the join statement) will be included in the result. If the outer join is on the right, an **SQL right outerjoin which uses the SQL RIGHT JOIN syntax**, then all rows on the table on the right (or second table in the join statement) will be included in the result.

To illustrate the use of a right outer join, we will use a modification of the preceding query we used to associate customers with transactions. With the left outer join, the NULLs showed customers who had not purchased a work. With the right outer join, the NULL will show works that have not been purchased by customers (and are therefore still in inventory and available for sale):

```
SELECT      C.LastName, C.FirstName, T.TransactionID, T.SalesPrice
FROM        CUSTOMER AS C RIGHT JOIN TRANS AS T
      ON    C.CustomerID = T.CustomerID
ORDER BY    T.TransactionID;
```

Amazon Ex. 1008.0290

	LastName	FirstName	TransactionID	SalesPrice
1	Janes	Jeffrey	100	42500.00
2	Twilight	Tiffany	101	500.00
3	Smith	David	102	200.00
4	Frederickson	Mary Beth	103	400.00
5	Smith	David	104	200.00
6	Frederickson	Mary Beth	105	400.00
7	Smathers	Fred	115	2750.00
8	Twilight	Tiffany	121	27500.00
9	Smith	David	125	200.00
10	NULL	NULL	126	NULL
11	Frederickson	Mary Beth	127	400.00
12	Warning	Selma	128	225.00
13	Warning	Selma	129	225.00
14	Warning	Selma	151	17500.00
15	Smith	David	152	225.00
16	Smith	David	153	350.00
17	Gray	Donald	154	400.00
18	NULL	NULL	155	NULL
19	Gray	Donald	156	400.00
20	Smathers	Fred	161	13750.00
21	Janes	Jeffrey	171	55000.00
22	Warning	Selma	175	72500.00
23	NULL	NULL	181	NULL
24	Gray	Donald	201	3250.00
25	Gray	Donald	202	3250.00
26	Wilkens	Chris	225	225.00
27	NULL	NULL	226	NULL
28	Wilkens	Chris	227	475.00
29	NULL	NULL	228	NULL
30	NULL	NULL	229	NULL
31	Twilight	Tiffany	241	4750.00
32	NULL	NULL	251	NULL
33	NULL	NULL	252	NULL
34	NULL	NULL	253	NULL
35	NULL	NULL	254	NULL

Figure 7-19

Result of Right Outer Join of CUSTOMER and TRANS

The result for this right join query is shown in Figure 7-19.

Joins that are not outer joins are called **inner joins**. All of the joins we have presented up to this discussion of outer have been inner joins, though we did not use that term. Outer joins can be combined to any level, just as inner joins can. The following SQL statement will obtain a list of every customer and the artists in which they have an interest:

```
SELECT      C.LastName, C.FirstName, A.LastName AS ArtistName
FROM        CUSTOMER AS C LEFT JOIN CUSTOMER_ARTIST_INT AS CI
    ON      C.CustomerID = CI.CustomerID
            LEFT JOIN   ARTIST AS A
                ON      CI.ArtistID = A.ArtistID;
```

Amazon Ex. 1008.0291

	LastName	FirstName	ArtistName
3	Janes	Jeffrey	Graves
4	Smith	David	Miro
5	Smith	David	Kandinsky
6	Smith	David	Matisse
7	Smith	David	Chagall
8	Smith	David	Sargent
9	Twilight	Tiffany	Sargent
10	Twilight	Tiffany	Tobey
11	Twilight	Tiffany	Horiuchi
12	Twilight	Tiffany	Graves
13	Smathers	Fred	Tobey
14	Smathers	Fred	Horiuchi
15	Smathers	Fred	Graves
16	Frederickson	Mary Beth	Miro
17	Frederickson	Mary Beth	Kandinsky
18	Frederickson	Mary Beth	Matisse
19	Frederickson	Mary Beth	Chagall
20	Warning	Selma	Chagall
21	Warning	Selma	Sargent
22	Warning	Selma	Graves
23	Wu	Susan	NULL
24	Gray	Donald	Tobey
25	Gray	Donald	Horiuchi
26	Gray	Donald	Graves
27	Johnson	Lynda	NULL
28	Wilkens	Chris	Tobey
29	Wilkens	Chris	Horiuchi
30	Wilkens	Chris	Graves

Figure 7-20

Result of Nested Left Outer Joins

The result for this query is in Figure 7-20. Note that if we leave either LEFT keyword out of the expression the null rows will not appear in Figure 7-20. A variant of this query using right joins can also be run (if you are running these queries in a DBMS, then try it out), but note that because (as shown in the data in Figure 7-16) there is at least one customer interested in each artist, there will be no NULL values in the result. Of course, that is a significant result, showing that customers are interested in all the artists featured at the View Ridge Gallery.

BY THE WAY It is easy to forget that inner joins will drop nonmatching rows. Some years ago, one of the authors had a very large organization as a consulting client. The client had a budgetary-planning application that included a long sequence of complicated SQL statements. One of the joins in that sequence was an inner join that should have been an outer join. As a result, some 3,000 employees dropped out of the budgetary calculations. The mistake was discovered only months later when the actual salary expense exceeded the budget salary expense by a large margin. The mistake was an embarrassment all the way to the board of directors.

Amazon Ex. 1008.0292

Using 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 obtains data from tables or other views. Views are constructed from SQL SELECT statements using the **SQL CREATE VIEW statement**, and view names are then used just as table names would be in the FROM clause of other SQL SELECT statements. The only limitation on the SQL statements that are used to create views is that they may not contain an ORDER BY clause.[1] The sort order must be provided by the SELECT statement that processes the view.

> **BY THE WAY** Views are a standard and popular SQL construct. Access, however, does not support them. Instead, in Access you can create a view-equivalent *query*, name it, and then save it. You can then process the query in the same ways that we process views in the following discussion.
>
> SQL Server, Oracle, and MySQL all support views, and they are an important structure with many uses. Do not conclude from Access's lack of support that views are unimportant. Read on, and, if possible, use SQL Server, Oracle, or MySQL to process the statements in this section.

The following statement defines a view named CustomerNameView on the CUSTOMER table:

```
CREATE VIEW CustomerNameView AS
       SELECT   LastName AS CustomerLastName,
                FirstName AS CustomerFirstName
       FROM     CUSTOMER;
```

Note that the results from executing this statement will be only a system message stating the action completed. With GUI utilities such as SQL Server Management Studio, an appropriately named object will also be created.

> **BY THE WAY** The current versions of SQL Server, Oracle, and MySQL all process the CREATE VIEW statements as written here without difficulty. However, an earlier version of SQL Server, SQL Server 2000, has a quirk: to create views, you have to remove the semicolon from the CREATE VIEW statement. We have no idea why SQL Server 2000 accepts a semicolon for all other SQL statements but will not accept one for SQL statements that create views. If, by chance, you are still using SQL Server 2000, be aware that you must remove the semicolon when writing CREATE VIEW statements.

Once the view is created, it can be used in the FROM clause of SELECT statements, just like a table. The following obtains a list of customer names in sorted order:

```
SELECT      *
FROM        CustomerNameView
ORDER BY    CustomerLastName, CustomerFirstName;
```

[1] This limitation appears in the SQL-92 standard, but how views are actually implemented varies by DBMS product. For example, Oracle allows views to include ORDER BY, whereas SQL Server will only allow ORDER BY in very limited circumstances.

Amazon Ex. 1008.0293

The result for the sample data in Figure 7-16 is:

	CustomerLastName	CustomerFirstName
1	Frederickson	Mary Beth
2	Gray	Donald
3	Janes	Jeffrey
4	Johnson	Lynda
5	Smathers	Fred
6	Smith	David
7	Twilight	Tiffany
8	Waming	Selma
9	Wilkens	Chris
10	Wu	Susan

Note that the number of columns returned in the result depends on the number of columns in the view, not on the number of columns in the underlying table. In this example, the SELECT clause produces just two columns because CustomerNameView itself has just two columns.

Also notice that the columns LastName and FirstName in the CUSTOMER table have been renamed to CustomerLastName and CustomerFirstName in the view. Because of this, the ORDER BY phrase in the SELECT statement uses CustomerLastName and CustomerFirstName, not LastName and FirstName. Also, the DBMS uses the labels CustomerLastName and CustomerFirstName when producing results.

BY THE WAY If you need to change an SQL view after you have created it, use the **SQL ALTER VIEW statement**. For example, if you wanted to reverse the order of LastName and FirstName in the CustomerNameView, you would use the SQL statement:

```
ALTER VIEW CustomerNameView AS
SELECT     FirstName AS CustomerFirstName,
           LastName AS CustomerLastName
FROM       CUSTOMER;
```

Figure 7-21 lists the uses for SQL views. SQL views 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. Additionally, SQL views can provide an alias for table names and thus hide the

Figure 7-21
Uses of SQL Views

- Hide columns or rows
- Display results of computations
- Hide complicated SQL syntax
- Layer built-in functions
- Provide level of isolation between table data and users' view of data
- Assign different processing permissions to different views of the same table
- Assign different triggers to different views of the same table

Amazon Ex. 1008.0294

true table names from applications and users. SQL Views also are used to assign different processing permissions and different triggers to different views of the same table. We will show examples for each of these.

Using SQL Views to Hide Columns and Rows

SQL Views can be used to hide columns to simplify results or to prevent the display of sensitive data. For example, suppose the users at View Ridge want a simplified list of customers that has just names and phone numbers. The following SQL statement defines a view, BasicCustomerDataView, that will produce that list:

```
CREATE VIEW BasicCustomerDataView AS
     SELECT  LastName AS CustomerLastName,
             FirstName AS CustomerFirstName,
             AreaCode, PhoneNumber
       FROM  CUSTOMER;
```

To use this view, we can run the SQL statement:

```
SELECT        *
FROM          BasicCustomerDataView
ORDER BY      CustomerLastName, CustomerFirstName;
```

The result is:

	CustomerLastName	CustomerFirstName	AreaCode	PhoneNumber
1	Frederickson	Mary Beth	303	513-8822
2	Gray	Donald	707	568-4839
3	Janes	Jeffrey	425	543-2345
4	Johnson	Lynda	202	438-5498
5	Smathers	Fred	206	876-9911
6	Smith	David	970	654-9876
7	Twilight	Tiffany	360	765-5566
8	Waming	Selma	604	988-0512
9	Wilkens	Chris	360	876-8822
10	Wu	Susan	404	653-3465

If the management of the View Ridge Gallery wants to hide the columns AcquisitionPrice and SalesPrice in TRANS they can define a view that does not include those columns. One use for such a view is to populate a Web page.

SQL Views also can hide rows by providing a WHERE clause in the view definition. The next SQL statement defines a view of customer name and phone data for all customers with an address in Washington State:

```
CREATE VIEW BasicCustomerDataWAView AS
     SELECT     LastName AS CustomerLastName,
                FirstName AS CustomerFirstName,
                AreaCode, PhoneNumber
       FROM     CUSTOMER
       WHERE    State = 'WA';
```

Amazon Ex. 1008.0295

To use this view, we can run the SQL statement:

```
SELECT      *
FROM        BasicCustomerDataWAView
ORDER BY    CustomerLastName, CustomerFirstName;
```

The result is:

	CustomerLastName	CustomerFirstName	AreaCode	PhoneNumber
1	Janes	Jeffrey	425	543-2345
2	Smathers	Fred	206	876-9911
3	Twilight	Tiffany	360	765-5566
4	Wilkens	Chris	360	876-8822

As desired, only customers who live in Washington are shown in this view. This limitation is not obvious from the results because State is not included in the view. This characteristic is good or bad, depending on the use of the view. It is good if this view is used in a setting in which only Washington customers matter; it is bad if the view miscommunicates that these customers are the only View Ridge customers.

Using SQL Views to Display Results of Computed Columns

Another purpose of views is to show the results of computed columns without requiring the user to enter the computation expression. For example, the following view combines the AreaCode and PhoneNumber columns and formats the result:

```
CREATE VIEW CustomerPhoneView AS
    SELECT    LastName AS CustomerLastName,
              FirstName AS CustomerFirstName,
              ('(' + AreaCode + ')' + PhoneNumber) As CustomerPhone
     FROM     CUSTOMER;
```

When the view user executes the SQL statement:

```
SELECT      *
FROM        CustomerPhoneView
ORDER BY    CustomerLastName, CustomerFirstName;
```

the results[2] will be:

	CustomerLastName	CustomerFirstName	CustomerPhone
1	Frederickson	Mary Beth	(303) 513-8822
2	Gray	Donald	(707) 568-4839
3	Janes	Jeffrey	(425) 543-2345
4	Johnson	Lynda	(202) 438-5498
5	Smathers	Fred	(206) 876-9911
6	Smith	David	(970) 654-9876
7	Twilight	Tiffany	(360) 765-5566
8	Warning	Selma	(604) 988-0512
9	Wilkens	Chris	(360) 876-8822
10	Wu	Susan	(404) 653-3465

[2] As you might expect, different DBMS products use a different operator for the concatenation operation in the CustomerPhoneView definition. For example, in Oracle the plus sign (+) must be replaced by double vertical bars (||) for string concatenation. See the documentation for your DBMS for more details.

Amazon Ex. 1008.0296

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 each developer who uses a computation writes his or her own SQL expression, that developer may write it differently and obtain inconsistent results.

Using SQL Views to Hide Complicated SQL Syntax

Another use of SQL views is to hide complicated SQL syntax. Using a view, developers need not enter a complex SQL statement when they want a particular result. Also, such views give the benefits of complicated SQL statements to developers who do not know how to write such statements. This use of views also ensures consistency.

For example, suppose that the View Ridge Gallery salespeople want to see which customers are interested in which artists. To display these interests, two joins are necessary: one to join CUSTOMER to CUSTOMER_ARTIST_INT and another to join that result to ARTIST. We can code an SQL statement that constructs these joins and define it as an SQL view. In fact, we already have, and we will reuse a variant of our nested left outer join example discussed earlier [we simply use equijoins (i.e., inner joins) instead of outer joins, because we are not interested in NULL values in these results] to create the CustomerInterestsView:

```
CREATE VIEW CustomerInterestsView AS
    SELECT     C.LastName AS CustomerLastName,
               C.FirstName AS CustomerFirstName,
               A.LastName AS ArtistName
    FROM       CUSTOMER AS C JOIN CUSTOMER_ARTIST_INT AS CI
          ON   C.CustomerID = CI.CustomerID
               JOIN    ARTIST AS A
                  ON   CI.ArtistID = A.ArtistID;
```

Notice the aliasing of C.LastName to CustomerLastName and A.LastName to ArtistLastName. We *must* use at least one of these column aliases, for without them the resulting table has two columns named LastName. The DBMS would not be able to distinguish one LastName from the other and would generate an error when an attempt is made to create such a view.

This is a complicated SQL statement to write, but once the view is created the result of this statement can be obtained with a simple SELECT statement. For example, the following statement shows the results sorted by CustomerLastName and CustomerFirstName:

```
SELECT        *
FROM          CustomerInterestsView
ORDER BY      CustomerLastName, CustomerFirstName;
```

Figure 7-22 displays the fairly large result set. Clearly, using the view is much simpler than constructing the join syntax. Even developers who know SQL well will appreciate having a simpler SQL view with which to work.

Layering Built-in Functions

Recall from Chapter 2 that you cannot use a computation or a built-in function as part of an SQL WHERE clause. You can, however, construct a view that computes a variable and then write an

Amazon Ex. 1008.0297

	CustomerLastName	CustomerFirstName	ArtistName
1	Frederickson	Mary Beth	Chagall
2	Frederickson	Mary Beth	Kandinsky
3	Frederickson	Mary Beth	Miro
4	Frederickson	Mary Beth	Matisse
5	Gray	Donald	Tobey
6	Gray	Donald	Horiuchi
7	Gray	Donald	Graves
8	Janes	Jeffrey	Graves
9	Janes	Jeffrey	Horiuchi
10	Janes	Jeffrey	Tobey
11	Smathers	Fred	Tobey
12	Smathers	Fred	Horiuchi
13	Smathers	Fred	Graves
14	Smith	David	Chagall
15	Smith	David	Matisse
16	Smith	David	Kandinsky
17	Smith	David	Miro
18	Smith	David	Sargent
19	Twilight	Tiffany	Sargent
20	Twilight	Tiffany	Tobey
21	Twilight	Tiffany	Horiuchi
22	Twilight	Tiffany	Graves
23	Warning	Selma	Chagall
24	Warning	Selma	Graves
25	Warning	Selma	Sargent
26	Wilkens	Chris	Tobey
27	Wilkens	Chris	Graves
28	Wilkens	Chris	Horiuchi

Figure 7-22
Result of CustomerInterestsView

SQL statement on that view that uses the computed variable in a WHERE clause. To understand this, consider the SQL view definition for the ArtistWorkNetView:

```
CREATE VIEW ArtistWorkNetView AS
    SELECT     W.WorkID, LastName AS ArtistLastName,
               FirstName AS ArtistFirstName, Title, Copy, DateSold,
               AcquisitionPrice, SalesPrice,
               (SalesPrice - AcquisitionPrice) AS NetProfit
    FROM       TRANS AS T JOIN WORK AS W
          ON   T.WorkID = W.WorkID
               JOIN       ARTIST AS A
                  ON      W.ArtistID = A.ArtistID;
```

Amazon Ex. 1008.0298

This SQL view joins TRANS, WORK, and ARTIST and creates the computed column NetProfit. We can now use NetProfit in an SQL WHERE clause in a query, as follows:

```
SELECT      ArtistLastName, ArtistFirstName,
            WorkID, Title, Copy, DateSold, NetProfit
FROM        ArtistWorkNetView
WHERE       NetProfit > 5000
ORDER BY    DateSold;
```

Here we are using the result of a computation in a WHERE clause, something that is not allowed in a single SQL statement. The result of the SQL SELECT statement is:

	ArtistLastName	ArtistFirstName	WorkID	Title	Copy	DateSold	NetProfit
1	Horiuchi	Paul	500	Memories IV	Unique	2005-12-14 ...	12500.00
2	Graves	Morris	548	Night Bird	Unique	2006-11-28 ...	12500.00
3	Graves	Morris	561	Sunflower	Unique	2007-06-28 ...	7500.00
4	Tobey	Mark	570	Untitled Number 1	Unique	2007-09-29 ...	6250.00
5	Horiuchi	Paul	571	Yellow Covers Blue	Unique	2007-09-29 ...	20000.00
6	Horiuchi	Paul	500	Memories IV	Unique	2007-12-18 ...	32500.00

Such layering can be continued over many levels. We can define another view with another computation on the computation in the first view. For example, note that in the results above the Horiuchi work Memories IV has been acquired and sold more than once by View Ridge, and then consider the SQL view ArtistWorkTotalNetView, which will calculate the total net profit from *all* sales of each work:

```
CREATE VIEW ArtistWorkTotalNetView AS
    SELECT      ArtistLastName, ArtistFirstName,
                WorkID, Title, Copy,
                SUM(NetProfit) AS TotalNetProfit
    FROM        ArtistWorkNetView
    GROUP BY    ArtistLastName, ArtistFirstName,
                WorkID, Title, Copy;
```

Now we can use TotalNetProfit in an SQL WHERE clause on the ArtistWorkTotalNet view, as follows:

```
SELECT      *
FROM        ArtistWorkTotalNetView
WHERE       TotalNetProfit > 5000
ORDER       BY TotalNetProfit;
```

In this SELECT, we are using an SQL view on an SQL view and a built-in function on a computed variable in the WHERE clause. The results are as follows:

	ArtistLastName	ArtistFirstName	WorkID	Title	Copy	TotalNetProfit
1	Tobey	Mark	570	Untitled Number 1	Unique	6250.00
2	Graves	Morris	561	Sunflower	Unique	7500.00
3	Graves	Morris	548	Night Bird	Unique	12500.00
4	Horiuchi	Paul	571	Yellow Covers Blue	Unique	20000.00
5	Horiuchi	Paul	500	Memories IV	Unique	45000.00

Amazon Ex. 1008.0299

Using SQL Views for Isolation, Multiple Permissions, and Multiple Triggers

SQL Views have three other important uses. First, they can isolate source data tables from application code. To see how, suppose we define the view:

```
CREATE VIEW CustomerTable001View AS
    SELECT      *
    FROM        CUSTOMER;
```

This view assigns the alias CustomerTable001View to the CUSTOMER table. If all application code uses the CustomerTable001View in SQL statements, then the true source of the data is hidden from application programmers.

Such table isolation provides flexibility to the database administration staff. For example, suppose that at some future date the source of customer data is changed to a different table (perhaps one that is imported from a different database) named NEW_CUSTOMER. In this situation, all the database administrator needs to do is redefine CustomerTable001View using the SQL ALTER VIEW statement, as follows:

```
ALTER VIEW CustomerTable001View AS
    SELECT      *
    FROM        NEW_CUSTOMER;
```

All of the application code that uses CustomerTable001View will now run on the new data source without any problem.

The second important use for SQL views is to give different sets of processing permissions to the same table. We will discuss security in more detail in Chapters 9, 10, 10A, and 10B, but for now understand that it is possible to limit insert, update, delete, and read permissions on tables and views.

For example, an organization might define a view of CUSTOMER called CustomerReadView with read-only permissions on CUSTOMER and a second view of CUSTOMER called CustomerUpdateView with both read and update permissions. Applications that need not update the customer data would work with CustomerReadView, whereas those that need to update this data would work with CustomerUpdateView.

The final use of SQL views is to enable the definition of multiple sets of triggers on the same source data. This technique is commonly used for enforcing O-M and M-M relationships. In this case, one view has a set of triggers that prohibits the deletion of a required child and another view has a set of triggers that deletes a required child as well as the parent. The views are assigned to different applications, depending on the authority of those applications.

Updating SQL Views

Some views can be updated, others cannot. The rules by which this is determined are both complicated and dependent on the DBMS in use. To understand why this is so, consider the following two update requests on views defined in the last section:

```
UPDATE      CustomerTable001View
SET         Email = 'NewEmailAddress@elsewhere.com'
WHERE       CustomerID = 1000;
```

and

```
UPDATE      ArtistWorkTotalNetView
SET         TotalNetProfit = 23000
WHERE       ArtistLastName = 'Tobey';
```

The first request can be processed without problem because CustomerTable001View is just an alias for the CUSTOMER table. The second update, however, makes no sense at all. TotalNetProfit is a sum of a computed column. Nowhere in the database is there any such column to be updated.

Amazon Ex. 1008.0300

Updatable Views:

- View based on a single table with no computed columns and all non-null columns present in the view
- View based on any number of tables, with or without computed columns, and INSTEAD OF trigger defined for the view

Possibly Updatable Views:

- Based on a single table, primary key in view, some required columns missing from view, update and delete may be allowed. Insert is not allowed.
- Based on multiple tables, updates may be allowed on the most subordinate table in the view if rows of that table can be uniquely identified.

Figure 7-23

Guidelines for Updating SQL Views

Figure 7-23 shows general guidelines to determine if a view is updatable. Again, the specifics depend on the DBMS product in use. In general, the DBMS must be able to associate the column(s) to be updated with a particular row in a particular table. A way to approach this question is to ask yourself, "What would I do if I were the DBMS and I were asked to update this view? Would the request make sense, and, if so, do I have sufficient data to make the update?" Clearly, if the entire table is present and there are no computed columns, the view is updatable. Also, the DBMS will mark the view as updatable if it has an INSTEAD OF trigger defined for it, as described later.

However, if any of the required columns are missing, the view clearly cannot be used for inserts. It may be used for updates and deletes, however, as long as the primary key (or, for some DBMS products, a candidate key) is present in the view. Multitable views may be updatable on the most subordinate table. Again, this can only be done if the primary key or candidate key for that table is in the view. We will revisit this topic for SQL Server 2008 in Chapter 10, Oracle Database 11g in Chapter 10A, and MySQL 5.1 in Chapter 10B.

Embedding SQL in Program Code

SQL statements can be embedded in application programs, triggers, and stored procedures. Before we discuss those subjects, however, we need to explain the placement of SQL statements in program code.

In order to embed SQL statements in program code, two problems must be solved. The first problem is that some means of assigning the results of SQL statements to program variables must be available. Many different techniques are used. Some involve object-oriented programs, whereas others are simpler. For example, in Oracle's PL/SQL the following statement assigns the count of the number of rows in the CUSTOMER table to the variable named *rowCount*:

```
SELECT      Count(*) INTO rowCount
FROM        CUSTOMER;
```

A similar statement in SQL Server's T-SQL is:

```
SELECT      @rowCount = Count(*)
FROM        CUSTOMER;
```

In either case, the execution of this code will place the number of rows in CUSTOMER into the program variable *rowCount* or *@rowCount*.

The second problem to solve concerns a paradigm mismatch between SQL and application programming languages. SQL is table oriented; SQL SELECT statements start with one or more tables and produce a table as output. Programs, however, start with one or more variables, manipulate them, and store the result in a variable. Because of this difference, an SQL statement like the following makes no sense:

```
SELECT      LastName INTO custLastName
FROM        CUSTOMER;
```

Amazon Ex. 1008.0301

If there are 100 rows in the CUSTOMER table, there will be 100 values of LastName. The program variable custLastName, however, is expecting to receive just one value.

To avoid this problem, the results of SQL statements are treated as **pseudofiles**. When an SQL statement returns a set of rows, a **cursor**, which is a pointer to a particular row, is established. The application program can then place the cursor on the first, last, or some other row of the SQL statement output table. With the cursor placed, values of columns for that row can be assigned to program variables. When the application is finished with a particular row, it moves the cursor to the next, prior, or some other row, and continues processing.

The typical pattern for using a cursor is as follows:

```
Open SQL (SELECT * FROM CUSTOMER);
Move cursor to first row;
   While cursor not past end of result{
      Set custLastName = LastName;
      . . . other statements . . .
      Advance cursor to next row;
   };
   . . . continue processing . . .
```

In this way, the rows of an SQL SELECT are processed one at a time. You will see many examples of these techniques and others like them in the chapters that follow. For now, try to gain an intuitive understanding of how SQL is embedded in program code.

Using SQL Triggers

A **trigger** is a stored program that is executed by the DBMS whenever a specified event occurs. Triggers for Oracle are written in Java or in Oracle's PL/SQL. SQL Server triggers are written in Microsoft .NET common language runtime 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. We will discuss triggers written in DBMS-specific SQL variants in Chapters 10 (T-SQL), 10A (PL/SQL), and 10B (MySQL SQL).

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 7-24 summarizes the triggers available for SQL Server 2008, Oracle Database 11g, and MySQL 5.1.

Oracle 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. Altogether, 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 2008 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.

MySQL Server 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.

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

Amazon Ex. 1008.0302

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

Figure 7-24
Summary of SQL Triggers by DBMS

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. For now, assume that new values are supplied by prefixing a column name with the expression *new:*. Thus, during an insert on CUSTOMER, the variable new:LastName is the value of LastName for the row being inserted. For an update, new:LastName has the value of LastName after the update takes place. Similarly, assume that old values are supplied by prefixing a column name with the expression *old:*. Thus, for a deletion, the variable old:LastName has the value of LastName for the row being deleted. For an update, old:LastName has the value of Name prior to the requested update. (This, in fact, is the strategy used by Oracle. You will see the equivalent SQL Server strategy in Chapter 10 and the way MySQL works in Chapter 10B.)

Triggers have many uses. In this chapter, we consider the four uses summarized in Figure 7-25:

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

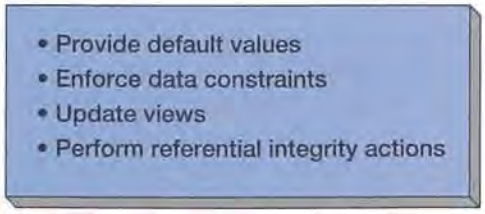
Using Triggers to Provide Default Values

Earlier in this chapter, you learned to use the SQL DEFAULT keyword to provide initial column values. DEFAULT works only for simple expressions, however. If the computation of a default value requires complicated logic, then an INSERT trigger must be used instead.

For example, suppose that there is a policy at View Ridge Gallery to set the value of AskingPrice equal either to twice the AcquisitionPrice or to the AcquisitionPrice plus the average net gain for sales of this art in the past, whichever is greater. The AFTER trigger in Figure 7-26 implements this policy. Note that the code in Figure 7-26 is generic. You will learn how to write specific code for SQL Server, Oracle, and MySQL in Chapters 10, 10A, and 10B respectively.

After declaring program variables, the trigger reads the TRANS table to find out how many TRANS rows exist for this work. Because this is an AFTER trigger, the new TRANS row for the

Figure 7-25
Uses for SQL Triggers



Amazon Ex. 1008.0303

```
CREATE TRIGGER TRANS_AskingPriceInitialValue
       AFTER INSERT ON TRANS

DECLARE
       rowcount as int;
       sumNetProfit as Numeric(10,2);
       avgNetProfit as Numeric(10,2);

BEGIN
       /* First find if work has been here before                          */

       SELECT    Count(*) INTO rowcount
       FROM      TRANS AS T
       WHERE     T.WorkID = new:WorkID;

       IF (rowcount = 1)
       THEN
          /* This is first time work has been in gallery                   */

          new:AskingPrice = 2 * new:AcquisitionPrice;

       ELSE
          IF rowcount > 1
          THEN
             /* Work has been here before                                  */

             SELECT    SUM(NetProfit) into sumNetProfit
             FROM      ArtistWorkNetView AWNV
             WHERE     AWNV.WorkID = new.WorkID
             GROUP BY  AWNV.WorkID;

             avgNetProfit = sumNetProfit / (rowcount - 1);

                /* Now choose larger value for the new AskingPrice         */

                IF ((new:AcquisitionPrice + avgNetProfit)
                      > (2 * new:AcquisitionPrice))
                THEN
                   new:AskingPrice = (new:AcquisitionPrice + avgNetProfit);
                ELSE
                   new:AskingPrice = (2 * new:AcquisitionPrice);
                END IF;
          ELSE
             /* Error, rowcount cannot be less than 1                      */
             /* Do something!                                              */
          END IF;
       END IF;
END;
```

Figure 7-26

Trigger Code to Insert a Default Value

work will have already been inserted. Thus, the count will be one if this is the first time the work has been in the gallery. If so, the new value of SalesPrice is set to twice the AcquisitionPrice.

If the rowcount is greater than one, then the work has been in the gallery before. To compute the average gain for this work, the trigger uses the ArtistWorkNetView described on page 271 to compute Sum(NetProfit) for this work. The sum is placed in the variable sumNetProfit. Notice that the WHERE clause limits the rows to be used in the view to this particular work. The average is then computed by dividing this sum by rowcount minus one.

You may be wondering, why not use AVG(NetProfit) in the SQL statement? The answer is that the default SQL average function would have counted the new row in the computation of the average. We do not want that row to be included, so we subtract one from rowcount when the average is computed. Once the value of avgNetProfit has been computed, it is compared with twice the AcquisitionPrice; the larger result is used for the new value of AskingPrice.

Amazon Ex. 1008.0304

Using Triggers to Enforce Data Constraints

A second purpose of triggers is to enforce data constraints. Although SQL CHECK constraints can be used to enforce domain, range, and intrarelation constraints, no DBMS vendor has implemented the SQL-92 features for interrelation CHECK constraints. Consequently, such constraints are implemented in triggers.

Suppose, for example, that the gallery has a special interest in Mexican painters and never discounts the price of their works. Thus, the SalesPrice of a work must always be at least the AskingPrice. To enforce this rule, the gallery database has an insert and update trigger on TRANS that checks to see if the work is by a Mexican painter. If so, the SalesPrice is checked against the AskingPrice. If it is less than the AskingPrice, the SalesPrice is reset to the AskingPrice. This, of course, must happen when the art work is actually being sold, and the customer charged the full amount! This is *not* a postsale accounting adjustment.

Figure 7-27 shows generic trigger code that implements this rule. This trigger will be fired after any insert or update on a TRANS row. The trigger first checks to determine if the work is by a Mexican artist. If not, the trigger is exited. Otherwise, the SalesPrice is checked against the AskingPrice; if it is less than the AskingPrice, the SalesPrice is set equal to the AskingPrice.

Figure 7-27

Trigger Code to Enforce an Interrelation Data Constraint

```
CREATE TRIGGER TRANS_CheckSalesPrice
       AFTER INSERT, UPDATE ON TRANS

DECLARE

       artistNationality      char (30);

BEGIN
       /* First determine if work is by a Mexican artist */

       SELECT    Nationality into artistNationality
       FROM      ARTIST AS A JOIN WORK AS W
             ON A.ArtistID = W.ArtistID
       WHERE     W.WorkID = new:WorkID;

       IF (artistNationality <> 'Mexican')
       THEN
          Exit Trigger;
       ELSE

          /* Work is by a Mexican artist - enforce constraint                */

          IF (new: SalesPrice < new:AskingPrice)
          THEN

             /* Sales Price is too low, reset it                             */

             UPDATE    TRANS
             SET       SalesPrice = new:AskingPrice;

             /* Note:  The above update will cause a recursive call on this */
             /* trigger. The recursion will stop the second time through    */
             /* because SalesPrice will be = AskingPrice.                   */

             /* At this point send a message to the user saying what's been */
             /* done so that the customer has to pay the full amount        */

          ELSE
             /* Error, artistNationality is either Mexican or it isn't      */
             /* Do something!                                               */
          END IF;
       END IF;
END;
```

Amazon Ex. 1008.0305

This trigger will be called recursively. The update statement in the trigger will cause an update on TRANS which will cause the trigger to be called again. The second time, however, the SalesPrice will be equal to the AskingPrice, no more updates will be made, and the recursion will stop.

Using Triggers to Update Views

As stated earlier, the DBMS can update some views but not others, depending on the way the view is constructed. Applications can sometimes update the views that the DBMS cannot update by applying logic that is particular to a given business setting. In this case, the application-specific logic for updating the view is placed in an INSTEAD OF trigger.

When an INSTEAD OF trigger is declared on a view, the DBMS performs no action other than to call the trigger. Everything else is up to the trigger. If you declare an INSTEAD OF INSERT trigger on view MyView, and if your trigger does nothing but send an e-mail message, then that e-mail message becomes the result of an INSERT on the view. INSERT MyView means "Send an e-mail," and nothing more.

More realistically, consider the SQL view CustomerInterestsView on page 270 and the result of that view in Figure 7-22. This view is the result of two joins across the intersection table between CUSTOMER and ARTIST. Suppose that this view populates a grid on a user form, and further suppose that users want to make customer name corrections, when necessary, on this form. If such changes are not possible, the users will say something like, "But, hey, the name is right there. Why can't I change it?" Little do they know the trials and tribulations the DBMS went through to display those data!

In any case, if, for example, the customer LastName value happens to be unique within the database, the view has sufficient information to update the customer's last name. Figure 7-28 shows generic trigger code for such an update. The code just counts the number of customers that have the old value of LastName. If only one customer has that value, then the update is made; otherwise, an error message is generated. Notice that the update activity is on one of the tables that underlie the view. The view, of course, has no real view data. Only actual tables can be updated.

Using Triggers to Implement Referential Integrity Actions

The fourth use of triggers is to implement referential integrity actions. Consider, for example, the 1:N relationship between DEPARTMENT and EMPLOYEE. Assume that the relationship is M-M and that EMPLOYEE.DepartmentName is a foreign key to DEPARTMENT.

To enforce this constraint, we will construct two views, both based on EMPLOYEE. The first view, DeleteEmployeeView, will delete an EMPLOYEE row only if that row is not the last child in the DEPARTMENT. The second view, DeleteEmployeeDepartmentView, will delete an EMPLOYEE row, and if that row is the last EMPLOYEE in the DEPARTMENT, it will also delete the DEPARTMENT row.

An organization would make the view DeleteEmployeeView available to applications that do not have permission to delete a row in DEPARTMENT. The view DeleteEmployeeDepartmentView would be given to applications that have permission to delete both employees and departments that have no employees.

Both of the views DeleteEmployeeView and DeleteEmployeeDepartmentView have the identical structure:

```
CREATE VIEW DeleteEmployeeView
    SELECT     *
    FROM       EMPLOYEE;
CREATE VIEW DeleteEmployeeDepartmentView
    SELECT     *
    FROM       EMPLOYEE;
```

Amazon Ex. 1008.0306

```
CREATE TRIGGER CustomerInterestView_UpdateCustomerLastName
       INSTEAD OF UPDATE ON CustomerInterestView

DECLARE

       rowcount  Int;

BEGIN

       SELECT    COUNT(*) into rowcount
       FROM      CUSTOMER
       WHERE     CUSTOMER.LastName = old:LastName

       IF (rowcount = 1)
       THEN

          /* If get here, then only one customer has this last name.      */
          /* Make the name change.                                        */

          UPDATE    CUSTOMER
          SET       CUSTOMER.LastName = new:LastName
          WHERE     CUSTOMER.LastName = old:LastName;

       ELSE

          IF (rowcount > 1 )
          THEN

             /* Send a message to the user saying cannot update because  */
             /* there are too many customers with this last name.        */

          ELSE
             /* Error, if rowcount <= 0 there is an error!               */
             /* Do something!                                            */
          END IF;
       END IF;
END;
```

Figure 7-28

Trigger Code to Update an SQL View

The trigger on DeleteEmployeeView, shown in Figure 7-29, determines if the employee is the last employee in the department. If not, the EMPLOYEE row is deleted. If however, the employee *is* the last employee in the department, *nothing* is done. Note again that the DBMS does nothing when an INSTEAD OF trigger is declared on the deletion. All activity is up to the trigger. If the employee is the last employee, then this trigger does nothing, which means that no change will be made to the database because the DBMS left all processing tasks to the INSTEAD OF trigger.

The trigger on DeleteEmployeeDepartment, shown in Figure 7-30, treats the employee deletion a bit differently. First, the trigger checks to determine if the employee is the last employee in the department. If so, the DEPARTMENT is deleted, and then, after that, the employee is deleted. Notice that the row in EMPLOYEE is deleted in either case.

Triggers such as those in Figures 7-29 and 7-30 are used to enforce the referential integrity actions for O-M and M-M relationships, as described at the end of Chapter 6. You will learn how to write them for Oracle in Chapter 10, for SQL Server in Chapter 11, and for MySQL in Chapter 12.

Amazon Ex. 1008.0307

```
CREATE TRIGGER EMPLOYEE_DeleteCheck
      INSTEAD OF DELETE ON DeleteEmployeeView

DECLARE

      rowcount  int;

BEGIN

      /*  First determine if this is the last employee in the department */

      SELECT    Count(*) into rowcount
      FROM      EMPLOYEE
      WHERE     EMPLOYEE.EmployeeNumber = old:EmployeeNumber;

      IF (rowcount > 1)
      THEN

         /* Not last employee, allow deletion                          */

         DELETE    EMPLOYEE
         WHERE     EMPLOYEE.EmployeeNumber = old:EmployeeNumber;

      ELSE

         /* Send a message to user saying that the last employee       */
         /* in a department cannot be deleted.                         */

      END IF;

END;
```

Figure 7-29

Trigger Code to Delete All but Last Child

Using Stored Procedures

A **stored procedure** is a program that is stored within the database and compiled when used. In Oracle, stored procedures can be written in PL/SQL or in Java. With SQL Server 2005 and SQL Server 2008, stored procedures are written in T-SQL or a .NET Common Language Runtime (CLR) languages such as VisualBasic.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 7-31.

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 or JavaScript. Ad hoc users can run them from DBMS management products such as SQL*Plus or SQL Developer in Oracle, SQL Server Management Studio in SQL Server, or the MySQL Query Browser in MySQL.

Advantages of Stored Procedures

The advantages of using stored procedures are listed in Figure 7-32. 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

Amazon Ex. 1008.0308

```
CREATE TRIGGER EMPLOYEE_DEPARTMENT_DeleteCheck
       INSTEAD OF DELETE ON DeleteEmployeeDepartmentView

DECLARE

       rowcount  int;

BEGIN

       /* Delete Employee row regardless of whether Department will be deleted  */

       DELETE    EMPLOYEE
       WHERE     EMPLOYEE.EmployeeNumber = old:EmployeeNumber;

       /*  First determine if this is the last employee in the department    */

       SELECT    Count(*) into rowcount
       FROM      EMPLOYEE
       WHERE     EMPLOYEE.EmployeeNumber = old:EmployeeNumber;

       IF (rowcount = 0)
       THEN

          /* Last employee in Department, delete Department                */

          DELETE    DEPARTMENT
          WHERE     DEPARTMENT.DepartmentName = old:DepartmentName;

       END IF;

END;
```

Figure 7-30

Trigger Code to Delete Last Child and Parent When Necessary

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.

Figure 7-31

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, and DELETE commands
  - Used for repetitive administration tasks or as part of an application

Amazon Ex. 1008.0309



Figure 7-32
Advantages of Stored Procedures

The WORK_AddWorkTransaction Stored Procedure

Figure 7-33 shows a stored procedure that records the acquisition of a work in the View Ridge database. Again, this code is generic, but the code style in Figure 7-33 is closer to that used in SQL Server rather than the more Oracle-like style that was used for the trigger examples in the prior section. If you compare the examples in both sections, you can gain a sense of the differences between code written in PL/SQL and T-SQL.

The WORK_addWorkTransaction procedure receives five input parameters and returns none. In a more realistic example, a return parameter would be passed back to the caller to indicate the success or failure of the operation. That discussion takes us away from database concepts, however, and we will omit it here. This code does not assume that the value of ArtistID that is passed to it is a valid ID. Instead, the first step in the stored procedure is to check whether the ArtistID value is valid. To do this, the first block of statements counts the number of rows that have the given ArtistID value. If the count is zero, then the ArtistID value is invalid, and the procedure writes an error message and returns.

Otherwise,[3] the procedure then checks to determine if the work has been in the View Ridge Gallery before. If so, the WORK table will already contain a row for this ArtistID, Title, and Copy. If no such row exists, the procedure creates a new WORK row. Once that has been done, it then uses a SELECT to obtain a value for the WorkID value. If the WORK row was just created, this statement is necessary to obtain the new value of the WorkID surrogate key. If the work was not created, the SELECT on WorkID is necessary to obtain the WorkID of the existing row. Once a value of WorkID has been obtained, the new row is inserted into TRANS. Notice that the system function GetDate() is used to supply a value for DateAcquired in the new row.

This procedure illustrates how SQL is embedded in stored procedures. It is not complete, because we need to do something to ensure that either all updates are made to the database or none of them are. You will learn how to do this in Chapter 9. For now, just concentrate on how SQL can be used as part of a database application.

Summary

SQL data definition language (DDL) statements are used to manage the structure of tables. This chapter presented three SQL DDL statements: CREATE TABLE, ALTER TABLE, and DROP TABLE. SQL is preferred over graphical tools for creating tables because it is faster, it can be used to create the same table repeatedly, tables can be created from program code, and it is standardized and DBMS independent.

The IDENTITY (*N*, *M*) data type is used to create surrogate key columns, where *N* is the starting value and *M* is the increment to be added. The SQL CREATE TABLE statement is used to define the name of the table, its columns, and constraints on columns. There are five types of constraints: PRIMARY KEY, UNIQUE, NULL/NOT NULL, FOREIGN KEY, and CHECK.

The purposes of the first three constraints are obvious. FOREIGN KEY is used to create referential integrity constraints; CHECK is used to create data constraints. Figure 7-11 summarizes techniques for creating relationships using SQL constraints.

Simple default values can be assigned using the DEFAULT keyword. Data constraints are defined using CHECK constraints. Domain, range, and intratable constraints can be defined. Although SQL-92 defined facilities for interrelation CHECK constraints, those facilities were not implemented by DBMS vendors. Instead, interrelation constraints are enforced using triggers.

[3] This code does not check for more than one row having the given ArtistID, because ArtistID is a surrogate key.

Amazon Ex. 1008.0310

```
CREATE PROCEDURE WORK_AddWorkTransaction
      (
      @ArtistID int,    /* Artist must already exist in database */
      @Title char(25),
      @Copy char(8),
      @Description varchar(1000),
      @AcquisitionPrice Numeric (6,2)
      )

/* Stored procedure to record the acquisition of a work.  If the work has    */
/* never been in the gallery before, add a new WORK row.  Otherwise, use     */
/* the existing WORK row.  Add a new TRANS row for the work and set           */
/* DateAcquired to the system date.                                          */

AS

      DECLARE @rowcount AS int
      DECLARE @WorkID AS int

      /* Check that the ArtistID is valid                                    */

      SELECT    @rowcount = COUNT(*)
      FROM      ARTIST AS A
      WHERE     A.ArtistID = @ArtistID

      IF (@rowcount = 0)
         /* The Artist does not exist in the database                        */
         BEGIN
            Print 'No artist with id of ' + Str(@artistID)
            Print 'Processing terminated.'
            RETURN
         END

      /* Check to see if the work is in the database                         */

      SELECT    @rowcount = COUNT(*)
      FROM      WORK AS W
      WHERE     W.ArtistID = @ArtistID and
                W.Title = @Title and
                W.Copy = @Copy

      IF (@rowcount = 0)
         /* The Work is not in database, so put it in.                       */
         BEGIN
            INSERT INTO WORK (Title, Copy, Description, ArtistID)
               VALUES (@Title, @Copy, @Description, @ArtistID)
         END

      /* Get the work surrogate key WorkID value                             */

      SELECT    @WorkID = W.WorkID
      FROM      WORK AS W
      WHERE     W.ArtistID = @ArtistID
         AND    W.Title = @Title
         AND    W.Copy = @Copy

      /* Now put the new TRANS row into database.                            */

      INSERT INTO TRANS (DateAcquired, AcquisitionPrice, WorkID)
         VALUES (GetDate(), @AcquisitionPrice, @WorkID)

      RETURN
```

Figure 7-33

Stored Procedure to Record the Acquisition of a Work

Amazon Ex. 1008.0311

The ALTER statement is used to add and remove columns and constraints. The DROP statement is used to drop tables. In SQL DDL, parents need to be created first and dropped last.

The data manipulation language (DML) SQL statements are INSERT, UPDATE, and DELETE. Each statement can be used on a single row, on a group of rows, or on the entire table.

Because of their power, both UPDATE and DELETE need to be used with care.

Some people believe the JOIN ON syntax is an easier form of join. Rows that have no match in the join condition are dropped from the join results. To keep such rows, use a LEFT OUTER or RIGHT OUTER join rather than a regular, or INNER, join.

An SQL view is a virtual table that is constructed from other tables and views. SQL SELECT statements are used to define views. The only restriction is that a view definition may not include an ORDER BY clause.

Views are used to hide columns or rows and to show the results of computed columns. They also can hide complicated SQL syntax, such as that used for joins and GROUP BY queries, and layer computations and built-in functions so that computations can be used in WHERE clauses. Some organizations use views to provide table aliases. Views also can be used to assign different sets of processing permissions to tables and to assign different sets of triggers as well. The rules for determining whether a view can be updated are both complicated and DBMS specific. Guidelines are shown in Figure 7-23.

SQL can be embedded in program code in triggers, stored procedures, and application code. To do so, there must be a way to associate SQL table columns with program variables. Also, there is a paradigm mismatch between SQL and programs. Most SQL statements return sets of rows; an application expects to work on one row at a time. To resolve this mismatch, the results of SQL statements are processed as pseudofiles using a cursor.

A trigger is a stored program that is executed by the DBMS whenever a specified event occurs on a specified table or view. In Oracle, triggers can be written in Java or in a proprietary Oracle language called PL/SQL. In SQL Server, triggers can be written in a propriety SQL Server language called TRANSACT-SQL, or T-SQL, and in Microsoft CLR languages such as Visual Basic .NET, C# .NET and C++ .NET. With MySQL, triggers can be written in MySQL's variant of SQL.

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. When a trigger is fired, the DBMS supplies old and new values for the update. New values are provided for inserts and updates, and old values are provided for updates and deletions. How these values are provided to the trigger depends on the DBMS in use.

Triggers have many uses. This chapter discussed four: setting default values, enforcing interrelation data constraints, updating views, and enforcing referential integrity actions.

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.

Stored procedures can be called from programs written in the same languages used for triggers. They also can be called from DBMS SQL utilities. The advantages of using stored procedures are summarized in Figure 7-32.

Key Terms

casual relationship
CHECK constraint
cursor
data definition language (DDL)
data manipulation language (DML)
DEFAULT keyword
FOREIGN KEY constraint
IDENTITY({StartValue},{Increment}) property
inner join
interrelation constraint
intrarelation constraint
NOT NULL constraint
NULL constraint
PRIMARY KEY constraint
procedural language
Procedural Language/SQL (PL/SQL)
pseudofile
SQL ALTER statement
SQL ALTER VIEW statement
SQL AS keyword
SQL CREATE TABLE statement
SQL CREATE VIEW statement
SQL DELETE statement
SQL DROP TABLE statement
SQL INSERT statement
SQL JOIN ON syntax
SQL LEFT JOIN syntax
SQL left outer join
SQL ON DELETE clause
SQL ON UPDATE clause
SQL outer join
SQL RIGHT JOIN syntax
SQL right outer join
SQL TOP NumberOfRows syntax
SQL UPDATE statement
SQL view
stored procedure
Transact-SQL (T-SQL)
trigger
UNIQUE constraint

Amazon Ex. 1008.0312

Review Questions

7.1 What does DDL stand for? List the SQL DDL statements.

7.2 What does DML stand for? List the SQL DML statements.

7.3 Explain the meaning of the following expression: IDENTITY (4000, 5).

For this set of Review Questions, we will create and use a database for the Review Wedgewood Pacific Corporation (WPC) that is similar to the Microsoft Access database we created and used in Chapter 1 and Chapter 2. Founded in 1957 in Seattle, Washington, WPC has grown into an internationally recognized organization. The company is located in two buildings. One building houses the Administration, Accounting, Finance, and Human Resources departments, and the second houses the Production, Marketing, and Information Systems departments. The company database contains data about employees; departments; projects; assets, such as computer equipment; and other aspects of company operations.

The database will be named WPC and will contain the following four tables:

DEPARTMENT (DepartmentName, BudgetCode, OfficeNumber, Phone)
EMPLOYEE (EmployeeNumber, FirstName, LastName, *Department*, Phone, Email)
PROJECT (ProjectID, Name, *Department*, MaxHours, StartDate, EndDate)
ASSIGNMENT (*ProjectID*, *EmployeeNumber*, HoursWorked)

EmployeeNumber is a surrogate key that starts at 1 and increments by 1. ProjectID is a surrogate key that starts at 1000 and increases by 100. DepartmentName is the text name of the department, and is therefore not a surrogate key.

The WPC database has the following referential integrity constraints:

Department in EMPLOYEE must exist in Department in DEPARTMENT
Department in PROJECT must exist in Department in DEPARTMENT
ProjectID in ASSIGNMENT must exist in ProjectID in PROJECT
EmployeeNumber in ASSIGNMENT must exist in EmployeeNumber in EMPLOYEE

The relationship from EMPLOYEE to ASSIGNMENT is 1:N, M-O and the relationship from PROJECT to ASSIGNMENT is 1:N, M-O.

The database also has the following business rules:

- If an EMPLOYEE row is to be deleted and that row is connected to any ASSIGNMENT, the EMPLOYEE row deletion will be disallowed.
- If a PROJECT row is deleted, then all the ASSIGNMENT rows that are connected to the deleted PROJECT row will also be deleted.

The business sense of these rules is as follows:

- If an EMPLOYEE row is deleted (for example, if the employee is transferred), then someone must take over that employee's assignments. Thus, the application needs someone to reassign assignments before deleting the employee row.
- If a PROJECT row is deleted, then the project has been canceled, and it is unnecessary to maintain records of assignments to that project.

Amazon Ex. 1008.0313

The column characteristics for these tables are shown in Figures 1-26 (DEPARTMENT), 1-28 (EMPLOYEE), 2-26 (PROJECT), and 2-28 (ASSIGNMENT). The data for these tables are shown in Figures 1-27 (DEPARTMENT), 1-29 (EMPLOYEE), 2-27 (PROJECT), and 2-29 (ASSIGNMENT).

If at all possible, you should run your SQL solutions to the following questions against an actual database. Because we have already created this database in Microsoft Access, you should use an SQL-oriented DBMS such as Oracle, SQL Server, or MySQL in these exercises. If that is not possible, create a new Access database named WPC-CH07.accdb, and use the SQL capabilities in these exercises. In all the exercises, use the data types appropriate for the DBMS you are using.

Answer Review Questions 7.4–7.13 without running them on your DBMS.

7.4 Write a CREATE TABLE statement for the DEPARTMENT table.

7.5 Write a CREATE TABLE statement for the EMPLOYEE table. Email is required and is an alternate key, and the default value of Department is Human Resources. Cascade updates but not deletions from DEPARTMENT to EMPLOYEE.

7.6 Write a CREATE TABLE statement for PROJECT table. The default value for MaxHours is 100. Cascade updates but not deletions from DEPARTMENT to EMPLOYEE.

7.7 Write a CREATE TABLE statement for the ASSIGNMENT table. Cascade only deletions from PROJECT to ASSIGNMENT; do not cascade either deletions or updates from EMPLOYEE to ASSIGNMENT.

7.8 Modify your answer to Review Question 7.7 to include the constraint that StartDate be prior to EndDate.

7.9 Write an alternate SQL statement that modifies your answer to question 7.7 to make the relationship between EMPLOYEE and ASSIGNMENT a 1:1 relationship.

7.10 Write an ALTER statement to add the column AreaCode to EMPLOYEE. Assume that AreaCode is not required.

7.11 Write an ALTER statement to remove the column AreaCode from EMPLOYEE.

7.12 Write an ALTER statement to make Phone an alternate key in EMPLOYEE.

7.13 Write an ALTER statement to drop the constraint that Phone is an alternate key in EMPLOYEE.

If you are using a DBMS, then at this point you should create a database named WPC and run the SQL statements from Review Questions 7.4 through 7.8 only. [Hint: write and test an SQL script, and then run the script. Save the script as *WPC-Create-Tables.sql* for future use.] Do not run your answer to Review Question 7.9! After the tables are created, run your answers to Review Questions 7.10 through 7.13. Note that after these four statements have been run, the table structure is exactly the same as it was before you ran them.

7.14 Write INSERT statements to add the data shown in Figure 1-27 to the DEPARTMENT table. Run these statements to populate the DEPARTMENT table. [Hint: write and test an SQL script, and then run the script. Save the script as *WPC-Insert-DEPARTMENT-Data.sql* for future use.]

7.15 Write INSERT statements to add the data shown in Figure 1-29 to the EMPLOYEE table. Run these statements to populate the EMPLOYEE table. [Hint: write and test an SQL script, and then run the script. Save the script as *WPC-Insert-EMPLOYEE-Data.sql* for future use.]

Amazon Ex. 1008.0314

7.16 Write INSERT statements to add the data shown in Figure 2-27 to the PROJECT table. Run these statements to populate the PROJECT table. [Hint: write and test an SQL script, and then run the script. Save the script as *WPC-Insert-PROJECT-Data.sql* for future use.]

7.17 Write INSERT statements to add the data shown in Figure 1-29 to the ASSIGNMENT table. Run these statements to populate the ASSIGNMENT table. [Hint: write and test an SQL script, and then run the script. Save the script as WPC-Insert-ASSIGNMENT-Data.sql for future use.]

7.18 Why were the tables populated in the order shown in Review Questions 7.14–7.17?

7.19 Assume that you have a table named NEW_EMPLOYEE that has the columns Department, Email, FirstName, and LastName, in that order. Write an INSERT statement to add all of the rows from the table NEW_EMPLOYEE to EMPLOYEE. Do not attempt to run this statement!

7.20 Write an UPDATE statement to change the phone number of employee with EmployeeNumber 11 to 360-287-8810. Run this SQL statement.

7.21 Write an UPDATE statement to change the department of employee with EmployeeNumber 5 to Finance. Run this SQL statement.

7.22 Write an UPDATE statement to change the phone number of employee with EmployeeNumber 5 to 360-287-8420. Run this SQL statement.

7.23 Combine your answers to Review Questions 7.21 and 7.22 into one SQL statement. Run this statement.

7.24 Write an UPDATE statement to set the HoursWorked to 60 for every row in ASSIGNMENT having the value 10 for EmployeeNumber. Run this statement.

7.25 Assume that you have a table named NEW_EMAIL, which has new values of Email for some employees. NEW_EMAIL has two columns: EmployeeNumber and NewEmail. Write an UPDATE statement to change the values of Email in EMPLOYEE to those in the NEW_EMAIL table. Do *not* run this statement.

7.26 Write one DELETE statement that will delete all data for project '2009 Q3 Product Plan' and all of its rows in ASSIGNMENT. Do *not* run this statement.

7.27 Write a DELETE statement that will delete the row for the employee named 'Smith'. Do *not* run this statement. What happens if this employee has rows in ASSIGNMENT?

7.28 Write an SQL statement to join EMPLOYEE, ASSIGNMENT, and PROJECT using the JOIN ON syntax. Run this statement.

7.29 Write an SQL statement to join EMPLOYEE and ASSIGNMENT and include all rows of EMPLOYEE in your answer, regardless of whether they have an ASSIGNMENT. Run this statement.

7.30 What is an SQL view? What purposes do views serve?

7.31 What is the limitation on SELECT statements used in SQL views?

7.32 Write an SQL statement to create a view named EmployeePhoneView that shows the values of EMPLOYEE.LastName as EmployeeLastName, EMPLOYEE.FirstName as EmployeeFirstName, and EMPLOYEE.Phone as EmployeePhone. Run this statement, and then test the view with an SQL SELECT statement.

7.33 Write an SQL statement to create a view named FinanceEmployeePhoneView that shows the values of EMPLOYEE.LastName as EmployeeLastName, EMPLOYEE.FirstName as EmployeeFirstName, and EMPLOYEE.Phone as EmployeePhone for employees

Amazon Ex. 1008.0315

who work in the Finance department. Run this statement, and then test the view with an SQL SELECT statement.

7.34 Write an SQL statement to create a view named CombinedNameEmployeePhoneView that shows the values of EMPLOYEE.LastName, EMPLOYEE.FirstName, and EMPLOYEE.Phone as EmployeePhone, but that combines EMPLOYEE.LastName and EMPLOYEE.FirstName into one column named EmployeeName that displays the employee name first name first. Run this statement, and then test the view with an SQL SELECT statement.

7.35 Write an SQL statement to create a view named EmployeeProjectAssignmentView that shows the values of EMPLOYEE.LastName as EmployeeLastName, EMPLOYEE.FirstName as EmployeeFirstName, EMPLOYEE.Phone as EmployeePhone, and PROJECT.Name as ProjectName. Run this statement, and then test the view with an SQL SELECT statement.

7.36 Write an SQL statement to create a view named DepartmentEmployeeProjectAssignmentView that shows the values of EMPLOYEE.LastName as EmployeeLastName, EMPLOYEE.FirstName as EmployeeFirstName, EMPLOYEE.Phone as EmployeePhone, DEPARTMENT.DepartmentName, Department.PHONE as DepartmentPhone, and PROJECT.Name as ProjectName. Run this statement, and then test the view with an SQL SELECT statement.

7.37 Write an SQL statement to create a view named ProjectHoursToDateView that shows the values of PROJECT.ProjectID, PROJECT.Name as ProjectName, PROJECT.MaxHours as ProjectMaxHour and the sum of ASSIGNMENT.HoursWorked as ProjectHoursWorkedToDate. Run this statement, and then test the view with an SQL SELECT statement.

7.38 Describe how views are used to provide an alias for tables. Why is this useful?

7.39 Explain how views can be used to improve data security.

7.40 Explain how views can be used to provide additional trigger functionality.

7.41 Give an example of a view that is clearly updatable.

7.42 Give an example of a view that is clearly not updatable.

7.43 Summarize the general idea for determining whether a view is updatable.

7.44 If a view is missing required items, what action on the view is definitely not allowed?

7.45 Explain the paradigm mismatch between SQL and programming languages.

7.46 How is the mismatch in your answer to Question 7.45 corrected?

7.47 What is a trigger?

7.48 What are PL/SQL and T-SQL? What is the MySQL equivalent?

7.49 What is the relationship between a trigger and a table or view?

7.50 Name nine possible trigger types.

7.51 Explain, in general terms, how new and old values are made available to a trigger.

7.52 Describe four uses for triggers.

7.53 Assume that the View Ridge Gallery will allow a row to be deleted from WORK if the work has never been sold. Explain, in general terms, how to use a trigger to accomplish such a deletion.

Amazon Ex. 1008.0316

7.54 Assume that the Wedgewood Pacific Corporation will allow a row to be deleted from EMPLOYEE if the employee has no project assignments. Explain, in general terms, how to use a trigger to accomplish such a deletion.

7.55 What is a stored procedure? How do they differ from triggers?

7.56 Summarize how to invoke a stored procedure.

7.57 Summarize the key advantages of stored procedures.

Project Questions

These Project Questions extend the Wedgewood Pacific Corporation database you created and used in the Review Questions with two new tables named COMPUTER and COMPUTER_ASSIGNMENT.

The data model for these modifications is shown in Figure 7-34. The column characteristics for the COMPUTER table are shown in Figure 7-35, and those for the

Figure 7-34

WPC Data Model Extension

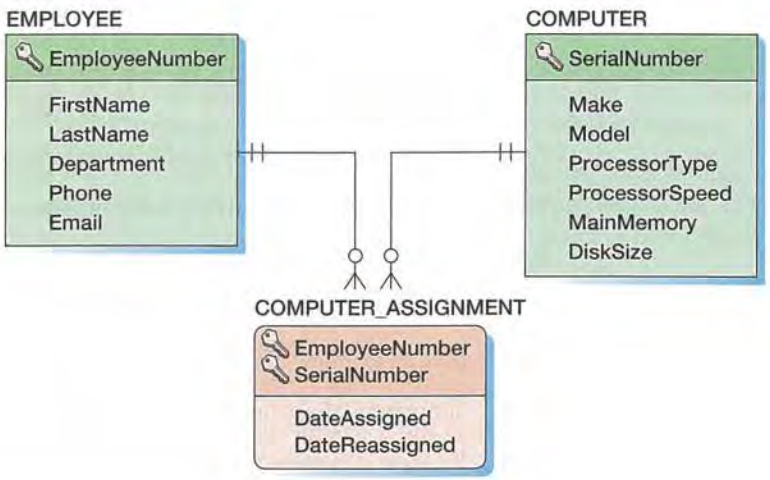


Figure 7-35

Database Column Characteristics for the COMPUTER Table

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

Amazon Ex. 1008.0317

Figure 7-36

Database Column Characteristics for the COMPUTER_ASSIGNMENT Table

ColumnName	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	No	Yes	Medium Date
DateReassigned	Date/Time	No	No	Medium Date

SerialNumber	Make	Model	ProcessorType	ProcessorSpeed	MainMemory	DiskSize
9871234	HP	Compaq DC5700	Intel Core 2 Duo	2.67	1.0 GByte	80 GBytes
9871245	HP	Compaq DC5700	Intel Core 2 Duo	2.67	1.0 GByte	80 GBytes
9871256	HP	Compaq DC5700	Intel Core 2 Duo	2.67	1.0 GByte	80 GBytes
9871267	HP	Compaq DC5700	Intel Core 2 Duo	2.67	1.0 GByte	80 GBytes
9871278	HP	Compaq DC5700	Intel Core 2 Duo	2.67	1.0 GByte	80 GBytes
9871289	HP	Compaq DC5700	Intel Core 2 Duo	2.67	1.0 GByte	80 GBytes
6541001	Dell	OptiPlex 755	Intel Core 2 Quad	2.4	2.0 GBytes	160 GBytes
6541002	Dell	OptiPlex 755	Intel Core 2 Quad	2.4	2.0 GBytes	160 GBytes
6541003	Dell	OptiPlex 755	Intel Core 2 Quad	2.4	2.0 GBytes	160 GBytes
6541004	Dell	OptiPlex 755	Intel Core 2 Quad	2.4	2.0 GBytes	160 GBytes
6541005	Dell	OptiPlex 755	Intel Core 2 Quad	2.4	2.0 GBytes	160 GBytes
6541006	Dell	OptiPlex 755	Intel Core 2 Quad	2.4	2.0 GBytes	160 GBytes

Figure 7-37

WPC COMPUTER Data

COMPUTER_ASSIGNMENT table are shown in Figure 7-36. Data for the COMPUTER table are shown in Figure 7-37, and data for the COMPUTER_ASSIGNMENT table are shown in Figure 7-38.

7.58 Describe the relationships in terms of type (identifying or nonidentifying) and maximum and minimum cardinality.

7.59 Explain the need for each of the foreign keys.

7.60 Define referential integrity actions (such as ON UPDATE CASCADE) for the COMPUTER-to-COMPUTER_ASSIGNMENT relationship only. Explain the need for these actions.

7.61 Assume that COMPUTER_ASSIGNMENT in the EMPLOYEE-to-COMPUTER_ASSIGNMENT relationship is now mandatory (i.e., every employee must have at least one computer). Use Figure 6-28(b) as a boilerplate to define triggers for enforcing the required child between EMPLOYEE and COMPUTER_ASSIGNMENT. Define the purpose of any necessary triggers.

7.62 Explain the interaction between the trigger in your answer to Question 7.61 and the COMPUTER-to-COMPUTER_ASSIGNMENT relationship. What, if any, cascading behavior do you want to occur? Explain how you can test to find out if it works the way that you want it to.

Amazon Ex. 1008.0318

SerialNumber	EmployeeNumber	DateAssigned	DateReassigned
9871234	11	15-Sep-08	21-Oct-08
9871245	12	15-Sep-08	21-Oct-08
9871256	4	15-Sep-08	
9871267	5	15-Sep-08	
9871278	8	15-Sep-08	
9871289	9	15-Sep-08	
6541001	11	21-Oct-08	
6541002	12	21-Oct-08	
6541003	1	21-Oct-08	
6541004	2	21-Oct-08	
6541005	3	21-Oct-08	
6541006	6	21-Oct-08	
9871234	7	21-Oct-08	
9871245	10	21-Oct-08	

Figure 7-38
WPC COMPUTER_ASSIGNMENT Data

7.63 Write CREATE TABLE statements for the COMPUTER and COMPUTER_ASSIGNMENT tables in Figure 7-34 using the column characteristics shown in Figures 7-35 and 7-36. Write CHECK constraints to ensure that Make is Dell, HP, Compaq, or Other. Also, write constraints to ensure that ProcessorSpeed is between 2.0 and 5.0 (these are units of Gigahertz). Run these statements on your WPC database to extend the database structure.

7.64 Using the sample data for the COMPUTER table shown in Figure 7-37 and the COMPUTER_ASSIGNMENT table shown in Figure 7-38, write INSERT statements to add this data to these tables in the WPC database. Run these INSERT statements to populate the tables.

7.65 Create a view of COMPUTER named ComputerView that displays SerialNumber and Make and Model as one attribute named ComputerType. Place a colon and a space between Make and Model in the format: Dell: 6200 Laptop. Run this statement, and then test with an SQL SELECT statement.

7.66 Create a view called ComputerMakeView that shows the Make and average ProcessorSpeed for all computers. Run this statement, and then test the view with an SQL SELECT statement.

7.67 Create a view called ComputerUserView that has all of the data of COMPUTER and ASSIGNMENT. Run this statement, and then test the view with an SQL SELECT statement.

7.68 Use the view you created called ComputerView to show the computer SerialNumber, ComputerType, and Employee name. Run this statement.

7.69 Suppose you want to use a stored procedure to store a new row in COMPUTER. List the minimum list of parameters that need to be in the procedure. Describe, in general terms, the logic of the stored procedure.

Amazon Ex. 1008.0319



Suppose that you have designed a database for Marcia's Dry Cleaning that has the following tables:

CUSTOMER (*CustomerID*, FirstName, LastName, Phone, Email)

ORDER (InvoiceNumber, *CustomerID*, DateIn, DateOut, Subtotal, Tax, TotalAmount)

ORDER_ITEM (*InvoiceNumber*, ItemNumber, *Service*, Quantity, UnitPrice, ExtendedPrice)

SERVICE (Service, Description, UnitPrice)

A. Specify NULL/NOT NULL constraints for each table column.

B. Specify alternate keys, if any.

C. State relationships as implied by foreign keys, and specify the maximum and minimum cardinality of each relationship. Justify your choices.

D. Explain how you will enforce the minimum cardinalities in your answer to part C. Use referential integrity actions for required parents, if any. Use Figure 6-28(b) as a boilerplate for required children, if any.

E. Write CREATE TABLE statements for each of the tables using your answers to parts A–D, as necessary. Set the first value of CustomerID to 100 and increment it by 5. Use FOREIGN KEY constraints to create appropriate referential integrity constraints. Set UPDATE and DELETE behavior in accordance with your referential integrity action design. Set the default value of Quantity to 1. Write a constraint that SERVICE.UnitPrice be between 1.50 and 10.00.

F. Explain how you would enforce the data constraint that ORDER_ITEM.UnitPrice be equal to SERVICE.UnitPrice, where ORDER_ITEM.Service = SERVICE.Service.

G. Write INSERT statements to insert at least three rows into each table. Use the data shown in Figures 2-33, 2-34, and 2-35 as a starting point, and create any other data needed for the revised set of tables used here.

H. Write an UPDATE statement to change values of SERVICE.Description from Mens Shirt to Mens' Shirts.

I. Write a DELETE statement(s) to delete an ORDER and all of the items on that ORDER.

J. Create a view called OrderSummaryView that contains ORDER.InvoiceNumber, ORDER.DateIn, ORDER.DateOut, ORDER_ITEM.ItemNumber, ORDER_ITEM. Service, and ORDER_ITEM.ExtendedPrice.

K. Create a view called CustomerOrderSummaryView that contains ORDER.InvoiceNumber, CUSTOMER.FirstName, CUSTOMER.LastName, CUSTOMER.Phone, ORDER.DateIn, ORDER.DateOut, ORDER.SubTotal, ORDER_ITEM.ItemNumber, ORDER_ITEM.Service, and ORDER_ITEM.ExtendedPrice.

L. Create a view called CustomerOrderHistoryView that (1) includes all columns of Customer-OrderSummaryView except ORDER_ITEM.ItemNumber and ORDER_ITEM. Service, (2) groups orders by CUSTOMER.LastName, CUSTOMER.FirstName and ORDER.InvoiceNumber in that order, and (3) sums and averages ORDER_ITEM.Extended-edPrice for each order for each customer.

M. Create a view called CustomerOrderCheckView that uses CustomerOrderHistoryView and that shows that any customers for whom the sum of ORDER_ITEM.ExtendedPrice is not equal to ORDER.SubTotal.

N. Explain, in general terms, how you will use triggers to enforce minimum cardinality actions as required by your design. You need not write the triggers, just specify which triggers you need and describe, in general terms, their logic.

Amazon Ex. 1008.0320



Suppose that you have designed a database for Morgan Importing that has the following tables:

STORE (StoreName, City, Country, Phone, Fax, Email, Contact)

PURCHASE (PurchaseID, *StoreName*, Date, Description, Category, PriceUSD)

SHIPMENT (ShipmentID, *ShipperID*, ShipperInvoiceNumber, Origin, Destination, DepartureDate, Arrival Date)

SHIPMENT_ITEM (*ShipmentID*, *PurchaseID*, InsuredValue)

SHIPPER (ShipperID, ShipperName, Phone, Fax, Email, Contact)

A. Do you think STORE should have a surrogate key? If so, create it and make required adjustments in the design. If not, explain why not or make other adjustments to STORE and other tables that you think are appropriate.

B. Specify NULL/NOT NULL constraints for each table column.

C. Specify alternate keys, if any.

D. State relationships as implied by foreign keys, and specify the maximum and minimum cardinality of each relationship. Justify your choices.

E. Explain how you will enforce the minimum cardinalities in your answer to part D. Use referential integrity actions for required parents, if any. Use Figure 6-28(b) as a boilerplate for required children, if any.

F. Write CREATE TABLE statements for each of the tables using your answers to the parts A–E, as necessary. Set the first value of PurchaseID to 500 and increment it by 5. Set the first value of ShipmentID to 100 and increment it by 1. Use FOREIGN KEY constraints to create appropriate referential integrity constraints. Set UPDATE and DELETE behavior in accordance with your referential integrity action design. Set the default value of InsuredValue to 100. Write a constraint that STORE.Country be limited to seven countries (you can pick the seven countries you want to purchase from, but be sure to include the countries for the ITEM.City locations shown in Figure 2-41).

G. Explain how you would enforce the rule that SHIPMENT_ITEM.InsuredValue be at least as great as PURCHASE.PriceUSD.

H. Write INSERT statements to insert at least three rows into each table. Use the data shown in Figures 2-39, 2-40, and 2-41 as a starting point, and create any other data needed for the revised set of tables used here.

I. Write an UPDATE statement to change values of STORE.City from New York City to NYC.

J. Write a DELETE statement(s) to delete a SHIPMENT and all of the items on that SHIPMENT.

K. Create a view called PurchaseSummaryView that shows only PURCHASE.PurchaseID, PURCHASE.Date, PURCHASE.Description, and PURCHASE.PriceUSD.

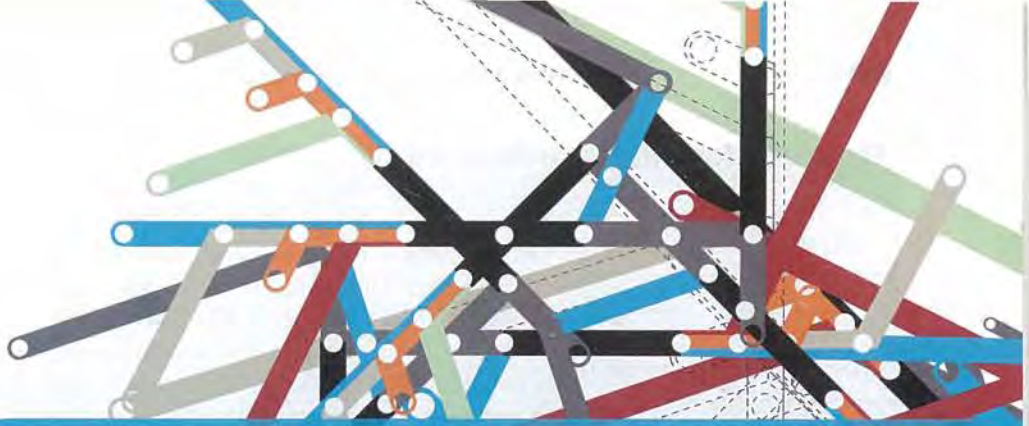
L. Create a view called StorePurchaseHistoryView that shows STORE.StoreName, STORE.Phone, STORE.Contact, PURCHASE.PurchaseID, PURCHASE.Date, PURCHASE.Description, and PURCHASE.PriceUSD.

M. Create a view called StoreHistoryView that sums the PriceUSD column of StorePurchaseHistoryView for each store into a column named TotalPurchases.

N. Create a view called MajorSources that uses StoreHistoryView and selects only those stores that have TotalPurchases greater than 100000.

O. Explain, in general terms, how you will use triggers to enforce minimum cardinality actions as required by your design. You need not write the triggers, just specify which triggers you need and describe, in general terms, their logic.

Amazon Ex. 1008.0321



8 Database Redesign

Chapter Objectives

- To understand the need for database redesign.
- To be able to use correlated subqueries.
- To be able to use the SQL EXISTS and NOT EXISTS keywords in correlated subqueries.
- To understand reverse engineering.
- To be able to use dependency graphs.
- To be able to change table names.
- To be able to change table columns.
- To be able to change relationship cardinalities.
- To be able to change relationship properties.
- To be able to add and delete relationships.

As stated in Chapter 1, databases arise from three sources. They can be created from existing tables and spreadsheets, they can be the result of a new systems development project, or they can be the outcome of database redesign. We have discussed the first two sources in Chapters 2 through 7. In this chapter, we will discuss the last source: database redesign.

We begin with a discussion of the need for database redesign, and then we will describe two important SQL statements: correlated subqueries and EXISTS. These statements play an important role when analyzing data prior to redesign. They also can be used for advanced queries and are important in their own right. After that discussion, we will turn to a variety of common database redesign tasks.

Amazon Ex. 1008.0322

The Need for Database Redesign

You may be wondering, "Why do we have to redesign a database? If we build it correctly the first time, why would we ever need to redesign it?" This question has two answers. First, it is not easy to build a database correctly the first time, especially databases that arise from the development of new systems. Even if we obtain all of the users' requirements and build a correct data model, the transformation of that data model into a correct database design is difficult. For large databases, the tasks are daunting and may require several stages of development. During those stages, some aspects of the database will need to be redesigned. Also, inevitably, mistakes will be made that must be corrected.

The second answer to this question is the more important one. Reflect for a moment on the relationship between information systems and the organizations that use them. It is tempting to say that they influence each other; that is, that information systems influence organizations and that organizations influence information systems.

In truth, however, the relationship is much stronger than that. Information systems and organizations do not just influence each other; they *create* each other. When a new information system is installed, the users can behave in new ways. As the users behave in those new ways, they will want changes to the information system to accommodate their new behaviors. As those changes are made, the users will have more new behaviors, they will request more changes to the information system, and so forth, in a never-ending cycle.

This circular process means that changes to an information system are not the sad consequence of a poor implementation, but rather are a natural outcome of information system use. Therefore, the need for change to information systems never goes away; it neither can nor should be removed by better requirements definition, better initial design, better implementation, or anything else. Instead, change is part and parcel of information systems use. Thus, we need to plan for it. In the context of database processing, this means we need to know how to perform database redesign.

SQL Statements for Checking Functional Dependencies

Database redesign is not terribly difficult if the database has no data. The serious difficulties arise when we have to change a database that has data and when we want to make changes with minimum impact on existing data. Telling the users that the system now works the way they want but that all of their data were lost while making the change is not acceptable.

Often, we need to know whether certain conditions or assumptions are valid in the data before we can proceed with a change. For example, we may know from user requirements that Department functionally determines DeptPhone, but we may not know whether that functional dependency is correctly represented in all of the data.

Recall from Chapter 3 that if Department determines DeptPhone, every value of Department must be paired with the same value of DeptPhone. If, for example, Accounting has a DeptPhone value of 834-1100 in one row, it should have that value in every row in which it appears. Similarly, if Finance has a DeptPhone of 834-2100 in one row, it should have that value in all rows in which it appears. Figure 8-1 shows data that violate this assumption. In the third row, the DeptPhone for Finance is different than for the other rows; it has too many zeroes. Most likely, someone made a keying mistake when entering DeptPhone. Such errors are typical.

Now, before we make a database change, we need to find all such violations and correct them. For the small table shown in Figure 8-1, we can just look at the data, but what if the EMPLOYEE table has 4,000 rows? Two SQL statements are particularly helpful in this regard: correlated subqueries and their cousin, EXISTS/NOT EXISTS. We will consider each of these in turn.

Amazon Ex. 1008.0323

EmployeeNumber	LastName	Email	Department	Phone
100	Johnson	JJ@somewhere.com	Accounting	834-1100
200	Abernathy	MA@somewhere.com	Finance	834-2100
300	Smathers	LS@somewhere.com	Finance	834-21000
400	Caruthers	TC@somewhere.com	Accounting	834-1100
500	Jackson	TJ@somewhere.com	Production	834-4100
600	Caldera	EC@somewhere.com	Legal	834-3100
700	Bandalone	RB@somewhere.com	Legal	834-3100

Figure 8-1
Table Showing Constraint Assumption Violation

Correlated Subqueries

A **correlated subquery** looks very much like the subqueries we discussed in Chapter 2, but, in actuality, correlated subqueries are very different. To understand the difference, consider the following subquery, which is like those in Chapter 2:

```
SELECT    A.FirstName, A.lastName
FROM      ARTIST AS A
WHERE     A.ArtistID IN
          (SELECT   W.ArtistID
           FROM     WORK W
           WHERE    W.Title = 'Blue Interior');
```

The DBMS can process such subqueries from the bottom up. That is, it can first find all of the values of ArtistID in WORK that have the title 'Blue Interior' 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 the artist Mark Tobey, as we would expect.

Searching for Multiple Rows with a Given Title

Now, to introduce correlated subqueries, suppose that someone at View Ridge Gallery proposes that the Title column of WORK be an alternate key. If you look at the data in Figure 7-16(d), you can see that while there is only one copy of 'Blue Interior', there are two or more copies of other titles, such as 'Surf and Bird'. Therefore Title cannot be an alternate key.

However, if the WORK table had 10,000 or more rows, this would be difficult to determine. In that case, we need a query that examines the WORK table and displays the Title and Copy of any works that share the same title.

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

Amazon Ex. 1008.0324

Then, we take the value of Title in the second row and compare it with all other rows in the WORK table, printing out the Title and Copy of any duplicate works. We proceed in this way until all rows of WORK have been examined.

A Correlated Subquery That Finds Rows with the Same Title

The following correlated subquery performs the action just described:

```
SELECT    W1.Title, W1.Copy
FROM      WORK AS W1
WHERE     W1.Title IN
          (SELECT    W2.Title
           FROM      WORK AS W2
           WHERE     W1.Title = W2.Title
                AND  W1.WorkID <> W2.WorkID);
```

The result of this query for the data in Figure 7-16(d) is shown in Figure 8-2.

This subquery looks deceptively similar to a regular subquery. To the surprise of many students, this subquery and the one above are drastically different. Their similarity is only superficial.

Before explaining why, first notice the notation in the correlated subquery. The WORK table is used in both the upper and the lower SELECT statements. In the upper statement, it is given the alias W1; in the lower SELECT statement, it is given the alias W2.

In essence, when we use this notation, it is as if we have made two copies of the WORK table. One copy is called W1, and the second copy is called W2. Therefore, in the last two lines of the correlated subquery, values in the W1 copy of WORK are compared with values in the W2 copy.

Figure 8-2
Results of the Check for Duplicate Titles

	Title	Copy
1	Farmer's Market #2	267/500
2	Farmer's Market #2	268/500
3	Spanish Dancer	583/750
4	Spanish Dancer	588/750
5	Surf and Bird	142/500
6	Surf and Bird	362/500
7	Surf and Bird	365/500
8	Surf and Bird	366/500
9	Surf and Bird	Unique
10	The Fiddler	251/1000
11	The Fiddler	252/1000

Amazon Ex. 1008.0325

The Difference Between Regular and Correlated Subqueries

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

```
WHERE     W1.Title = W2.Title
      AND W1.WorkID <> W2.WorkID);
```

In these expressions, W1.Title (from the top SELECT statement) is being compared with W2.Title (from the bottom SELECT statement). The same is true for W1.WorkID and W2.WorkID. Because of this fact, the DBMS cannot process the subquery portion independent 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 W1. Using that row, evaluate the second query. To do that, for each row in W2, compare W1.Title with W2.Title and W1.WorkID with W2.WorkID. If the titles are equal and the values of WorkID are not equal, return the value of W2.Title to the upper query. Do this for every row in W2.

Once all of the rows in W2 have been evaluated for the first row in W1, move to the second row in W1 and evaluate it against all the rows in W2. Continue in this way until all rows of W1 have been compared with all of the rows of W2.

If this is not clear to you, write out two copies of the WORK data from Figure 7-16(d) on a piece of scratch paper. Label one of them W1 and the second W2 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:

```
SELECT    W1.Title, W1.Copy
FROM      WORK AS W1
WHERE     W1.WorkID IN
          (SELECT     W2.WorkID
           FROM       WORK AS W2
           WHERE      W1.Title = W2.Title
                AND   W1.WorkID <> W2.WorkID);
```

The logic here seems correct, but it is not. No row will ever be displayed by this query, regardless of the underlying data. See if you can see why this is so before continuing.

The bottom query will indeed find all rows that have the same title and different WorkIDs. If one is found, it will produce the W2.WorkID of that row. But that value will then be compared with W1.WorkID. *These two values will always be different because of the condition* W1.WorkID W2.WorkID. No rows are returned because the values of the two unequal WorkIDs are used in the IN instead of the values of the two equal Titles.

Using Correlated Subqueries to Check Functional Dependencies

Correlated subqueries can be used to advantage during database redesign. As mentioned, one application of correlated subqueries is to verify functional dependencies. For example, suppose we have EMPLOYEE data like that in Figure 8-1, and we want to know whether the data conform to the functional dependency Department → DeptPhone. If so, every time a given value of Department occurs in the table, that value will be matched with the same value of DeptPhone.

Amazon Ex. 1008.0326

The following correlated subquery will find any rows that violate this assumption:

```
SELECT    E1.Department, E1.DeptPhone
FROM      EMPLOYEE AS E1
WHERE     E1.Department IN
          (SELECT     E2.Department
           FROM       EMPLOYEE AS E2
           WHERE      E1.Department = E2.Department
                AND   E1.DeptPhone <> E2.DeptPhone);
```

The results for the data in Figure 8-1 are:

	Department	DeptPhone
1	Finance	834-2100
2	Finance	834-21000

A listing like this can readily be used to find and fix any rows that violate the functional dependency.

EXISTS and NOT EXISTS

EXISTS and NOT EXISTS are another form of correlated subquery. We can write the last correlated subquery using the **SQL EXISTS keyword,** as follows:

```
SELECT    E1.Department, E1.DeptPhone
FROM      EMPLOYEE AS E1
WHERE     EXISTS
          (SELECT     E2.Department
           FROM       EMPLOYEE AS E2
           WHERE      E1.Department = E2.Department
                AND   E1.DeptPhone <> E2.DeptPhone);
```

Because EXISTS is 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 department names are the same and the department phone numbers are different, then the EXISTS is true and the Department and DeptPhone 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 Department and DeptPhone of the second row are selected. This process is repeated for all of the rows in E1.

Using NOT EXISTS in a Double Negative

The EXISTS keyword will be true if *any* row in the subquery meets the condition. The **SQL NOT EXISTS keyword** will be true only if *all* rows in the subquery fail the condition. Consequently, the double use of NOT EXISTS can be used to find rows that do not not match a condition. Because of the logic of a double negative, if a row does not *not match any row*, then it matches every row. For example, suppose that at View Ridge the users want to know the name of any artist that every customer is interested in. We can proceed as follows. First, produce the set of all customers who are interested in a particular artist. Then, take the complement of that set, which will be the customers who are not interested in that artist. If that complement is empty, then all customers are interested in the given artist.

Amazon Ex. 1008.0327

BY THE WAY The doubly nested NOT EXISTS pattern is famous in one guise or another among SQL practitioners. It is often used as a test of SQL knowledge in job interviews and in bragging sessions, and it can be used to advantage when assessing the desirability of certain database redesign possibilities, as you will see in the last section of this chapter. Therefore, even though this example involves some serious study, it is worth your while to understand it.

The Double NOT EXISTS Query

The following SQL statement implements the strategy just described:

```
SELECT    A.FirstName, A.LastName
FROM      ARTIST AS A
WHERE     NOT EXISTS
          (SELECT     C.CustomerID
           FROM       CUSTOMER AS C
           WHERE      NOT EXISTS
                      (SELECT      CAI.CustomerID
                       FROM        CUSTOMER_ARTIST_INT AS CAI
                       WHERE       C.CustomerID= CAI.CustomerID
                            AND    A.ArtistID = CAI.ArtistID));
```

The result of this query is an empty set, indicating that there is *no* artist that *every* customer is interested in:

FirstName	LastName

Let's see how this works. The bottom SELECT finds all of the customers who are interested in a particular artist. As you read this SELECT (the last SELECT in the query), keep in mind that this is a correlated subquery; this SELECT is nested inside the query on CUSTOMER, which is nested inside the query on ARTIST. C.CustomerID is coming from the SELECT on CUSTOMER in the middle, and A.ArtistID is coming from the SELECT on ARTIST at the top.

Now, the NOT EXISTS in the sixth line of the query will find the customers who are not interested in the given artist. If all customers are interested in the given artist, the result of the middle SELECT will be null. If the result of the middle SELECT is null, the NOT EXISTS in the third line of the query will be true, and the name of that artist will be produced, just as we want.

Consider what happens for artists who do not qualify in this query. Suppose that every customer except Tiffany Twilight is interested in the artist Joan Miro. (This is not the case for the data in Figure 7-16, but assume that it were true.) Now, for the preceding query, when Miro's row is considered, the bottom SELECT will retrieve every customer except Tiffany Twilight. In this case, because of the NOT EXISTS in the sixth line of the query, the middle SELECT will produce the CustomerID for Tiffany Twilight (because her row is the only one that does not appear in the bottom SELECT). Now, because there is a result from the middle SELECT, the NOT EXISTS in the top SELECT is false and the name Joan Miro will not be included in the output of the query. This is correct because there is a customer who is not interested in Joan Miro.

Again, take some time to study this pattern. It is a famous one, and if you become a database professional, you will certainly see it again in one form or another. In fact, you will not not see it again!

Amazon Ex. 1008.0328

Analyzing the Existing Database

Before we proceed with a discussion of database redesign, reflect for a moment on what this task means for a real company whose operations are dependent on the database. Suppose, for example, that you work for a company such as Amazon.com. Further suppose that you have been tasked with an important database redesign assignment, say to change the primary key of the vendor table.

To begin, you may wonder, why would Amazon want to do this? It could be that in the early days, when it only sold books, Amazon used company names for vendors. But, as Amazon began to sell more types of products, company name was no longer sufficient. Perhaps there are too many duplicates, and Amazon may have decided to switch to an Amazon-created VendorID.

Now, what does it mean to switch primary keys? Besides adding the new data to the correct rows, what else does it mean? Clearly, if the old primary key has been used as a foreign key, all of the foreign keys need to be changed as well. So we need to know all of the relationships in which the old primary key was used. But what about views? Do any views use the old primary key? If so, they will need to be changed. What about triggers and stored procedures? Do any of them use the old primary key? Not to mention any application code that may break when the old key is removed.

Now, to create a nightmare, what happens if you get partway through the change process and something fails? Suppose you encounter unexpected data, and you receive errors from the DBMS while trying to add the new primary key. Amazon cannot change its Web site to display "Sorry, our database is broken—come back tomorrow (we hope)!"

This nightmare brings up many topics, most of which relate to systems analysis and design. But with regard to database processing, three principles become clear. First, as the carpenters say, "Measure twice and cut once." Before we attempt any structural changes to a database, we must clearly understand the current structure and contents of the database, and we must know what depends on what. Second, before we make any structural changes to an operational database, we must test those changes on a realistically sized test database that has all of the important test data cases. Finally, if at all possible, we need to create a complete backup of the operational database prior to making any structural changes. If all goes awry, the backup can be used to restore the database while problems are corrected. We will consider each of these important topics next.

Reverse Engineering

Reverse engineering is the process of reading a database schema and producing a data model from that schema. The data model produced is not truly a logical model because entities will be generated for every table, including entities for intersection tables that have no nonkey data and should not appear in a logical model at all. The model generated by reverse engineering is a thing unto itself, a table-relationship diagram that is dressed in entity-relationship clothes. In this text, we will call it the **reverse engineered (RE) data model**.

Figure 8-3 shows the RE database design of the View Ridge Gallery database produced by Microsoft Visio 2007 from an SQL Server 2008 Express version of the View Ridge database created in Chapter 7. Note that due to the limitations of Visio, this is a physical database design rather than a logical data model. Nonetheless, it illustrates the reverse engineering technique we are discussing.

We used Microsoft Visio here because of its general availability. This means, however, that we have to work with Visio's nonstandard database modeling notation. Note that Visio uses a directed arrow to indicate relationships (the relationship line starts at the child entity and the arrowhead points to the parent entity). Although Visio does differentiate between identifying and nonidentifying relationships in the database properties, it does not do so visually, as we have been doing. Visio also stores cardinalities as relationship properties, but it does not display them as we do with the IE Crow's Foot notation. We have added color to indicate the strong and weak entities, because Visio only uses a table structure. Note the notations PK indicating a primary key, FK indicating a foreign key, and U indicating a UNIQUE constraint (this corresponds to the AK*m.n* notation we have been using). Columns set to NOT NULL are in bold; columns set to NULL are in nonbold text. All in all, however, this is a reasonable representation of the View Ridge schema.

Amazon Ex. 1008.0329

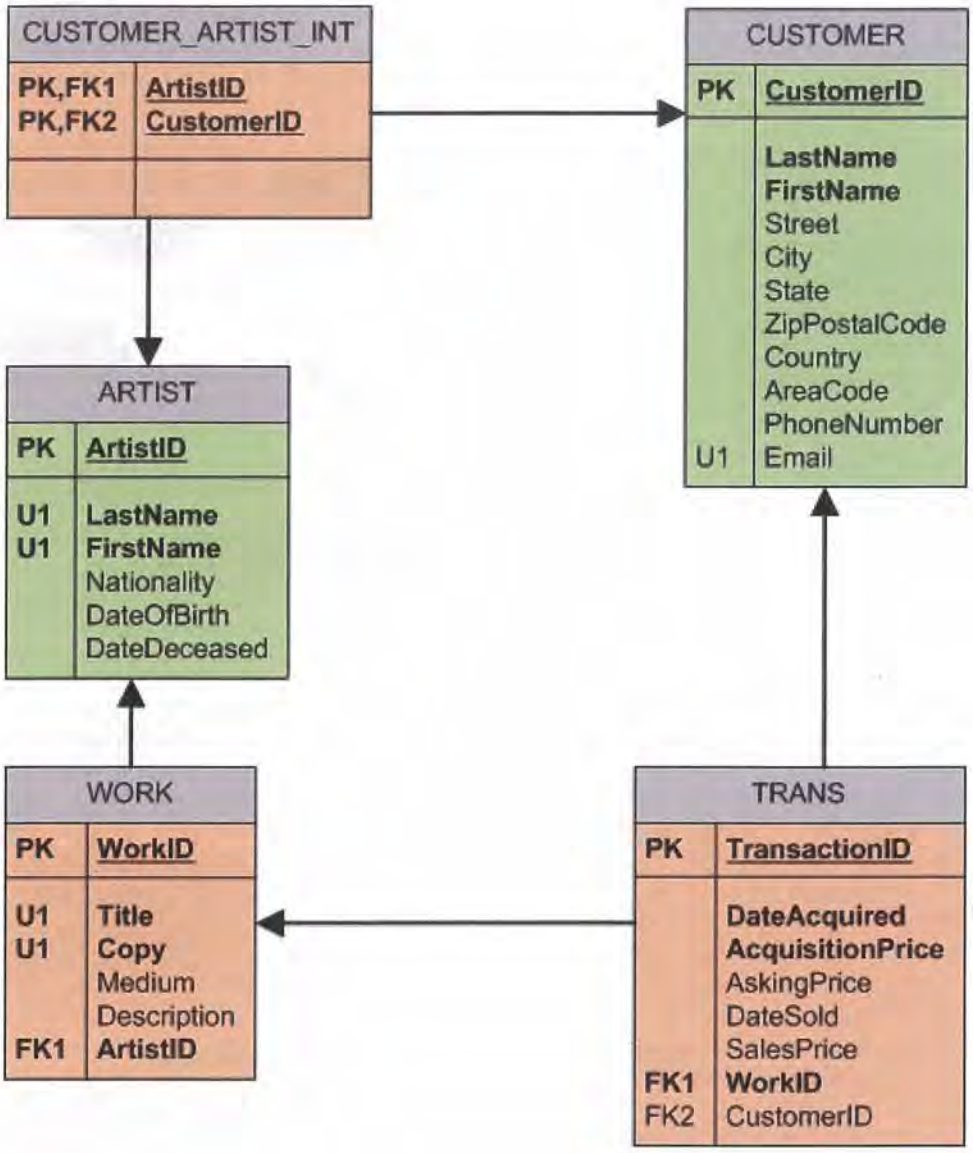


Figure 8-3
Reverse-Engineered Database Design

Although Visio produces only a database design, and not a data model, some other design software, such as Computer Associates' ERwin, can create both logical (data model) and physical (database design) versions of the database structure. In addition to tables and views, some data modeling products will capture constraints, triggers, and stored procedures from the database.

These constructs are not interpreted, but their text is imported into the data model. With some products, the relationship of the text to the items it references also is obtained. The redesign of constraints, triggers, and stored procedures is beyond the scope of our discussion here. You should realize that they, too, are part of the database, however, and are subject to redesign.

Dependency Graphs

Before making changes to database structures, it is vitally important to understand the dependencies of those structures. What changes will impact what? For example, consider changing the name of a table. Where is the table name used? In which triggers? In which stored procedures? In which relationships? Because of the need to know all of the dependencies, many database redesign projects begin by making a **dependency graph.**

The term *graph* arises from the mathematical topic of graph theory. Dependency graphs are not graphical displays like bar charts, rather they are diagrams that consist of nodes and arcs (or lines) that connect those nodes.

Amazon Ex. 1008.0330

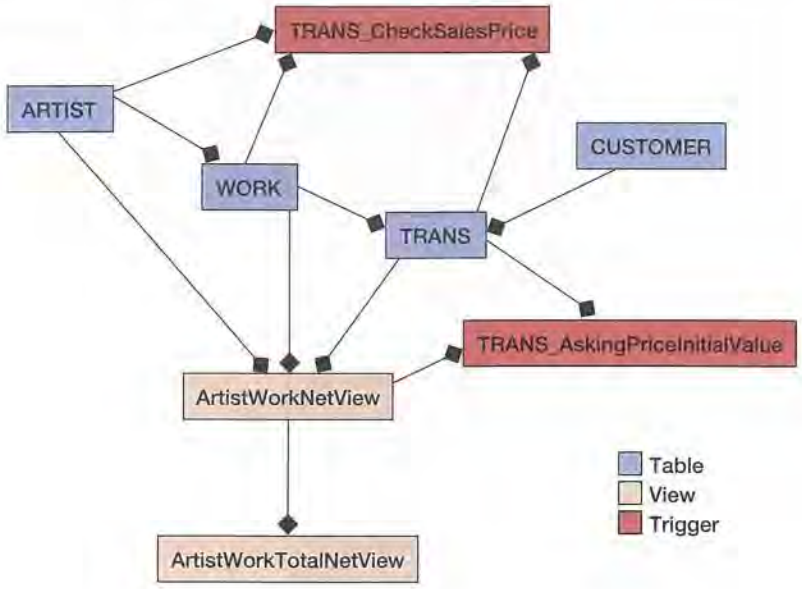


Figure 8-4
Example Dependency Graph (Partial)

Figure 8-4 shows a partial dependency graph that was drawn using the results of the RE model, but manually interpreting views and triggers we developed in Chapter 7. For simplicity, this graph does not show the views and triggers of CUSTOMER, nor does it show CUSTOMER_ARTIST_INT and related structures. Also, the stored procedure WORK_AddWorkTransaction is not shown, nor are the constraints.

Even this partial diagram reveals the complexity of dependencies among database constructs. You can see that it would be wise to tread lightly, for example, when changing anything in the TRANS table. The consequences of such a change need to be assessed against two relationships, three triggers, and three views. Again, measure twice and cut once!

Database Backup and Test Databases

Because of the potential damage that can be done to a database during redesign, a complete backup of the operational database should be made prior to making any changes. Equally important, it is essential that any proposed changes be thoroughly tested. Not only must structural changes proceed successfully, but all triggers, stored procedures, and applications must also run correctly on the revised database.

Typically, at least three different copies of the database schema are used in the redesign process. One is a small test database that can be used for initial testing. The second is a large test database, which may even be a full copy of the operational database. Sometimes, there are several large test databases. Finally, there is the operational database.

A means must be created to recover all test databases to their original state during the testing process. In that way, the test can be rerun as necessary against the same starting point. Depending on the facilities of the DBMS, backup and recovery or other means are used to restore the database after a test run.

Obviously, for enterprises with very large databases, it is not possible to have a test database that is a copy of the operational database. Instead, smaller test databases need to be created, but those test databases must have all the important data characteristics of the operational database; otherwise, they will not provide a realistic test environment. The construction of such test databases is in itself a difficult and challenging job. In fact, many interesting career opportunities are available for developing test databases and database test suites.

Finally, for organizations that have very large databases, it may not be possible to make a complete copy of the operational database prior to making structural changes. In this case, the database is backed up in pieces, and the changes are made in pieces as well. This task is very difficult and requires great knowledge and expertise. It also requires weeks or months

Amazon Ex. 1008.0331

of planning. You may participate as a junior member of a team to make such a change, but you should have years of database experience before you attempt to make structural changes to such large databases. Even then, it is a daunting task.

Changing Table Names and Table Columns

In this section, we will consider alterations to tables and their columns. To accomplish these changes, we will use only SQL statements. Many DBMS products have features to facilitate changing structures other than SQL. For example, some products have graphical design tools that simplify this process. But such features are not standardized, and you should not depend on them. The statements shown in this chapter will work with any enterprise-class DBMS product, and most will work with Access as well.

Changing Table Names

At first glance, changing a table name seems like an innocent and easy operation. A review of Figure 8-4, however, shows that the consequences of such a change are greater than you would think. If, for example, we want to change the name of the table WORK to WORK_VERSION2, several tasks are necessary. The constraint that defines the relationship from WORK to TRANS must be altered, ArtistWorkNetView view must be redefined, and then the TRANS_CheckSalesPrice trigger must be rewritten to use the new name.

Furthermore, there is no SQL command to change the name of the table. Instead, the table needs to be re-created under the new name, and the old table dropped. This requirement, however, suggests a good strategy for making table name changes. First, create the new table with all attendant structures and then drop the old one once everything is working with the new table. If the table to be renamed is too large to be copied, other strategies will have to be used, but they are beyond the scope of this discussion.

This strategy has one serious problem, however. WorkID is a surrogate key. When we create the new table, the DBMS will create new values of WorkID in the new table. The new values will not necessarily match the values in the old table, which means values of the foreign key TRANS.WorkID will be wrong. The easiest way to solve this problem is first to create the new version of the WORK table and not define WorkID as a surrogate key. Then, fill the table with the current values of WORK, including the current values of WorkID. Then, change WorkID to a surrogate key.

First, we create the table by submitting an SQL CREATE TABLE WORK_VERSION2 statement to the DBMS. We make WorkID an integer, but not a surrogate key. We also must give new names to the WORK constraints. The prior constraints still exist, and if new names are not used, the DBMS will issue a duplicate constraint error when processing the CREATE TABLE statements. Examples of new constraint names are:

```
CONSTRAINT WorkV2PK PRIMARY KEY(WorkID),
CONSTRAINT WorkV2AK1 UNIQUE(Title, Copy),
CONSTRAINT ArtistV2FK FOREIGN KEY(ArtistID)
                REFERENCES ARTIST(ArtistID)
                      ON DELETE NO ACTION
                      ON UPDATE NO ACTION
```

Next, copy the data into the new table with the following SQL statement:

```
INSERT INTO WORK_VERSION2 (WorkID, Copy, Title, Medium,
     Description, ArtistID)
     SELECT     WorkID, Copy, Title, Medium, Description, ArtistID
     FROM       WORK;
```

Amazon Ex. 1008.0332

At this point, alter the WORK_VERSION2 table to make WorkID a surrogate key. In SQL Server, the easiest way to do that is to open the graphical table designer and redefine WorkID as an IDENTITY column (there is no standard SQL for making this change). Set the Identity Seed property to the original value of 500, and SQL Server will set the next new value of WorkID to be the maximum largest value of WorkID plus one. A different strategy is used for surrogate keys with Oracle and MySQL, and these topics will be discussed in Chapters 10A and 10B.

Now all that remains is to define the two triggers. This can be done by copying the text of the old triggers and changing the name WORK to WORK_VERSION2.

At this point, test suites should be run against the database to verify that all changes have been made correctly. After that, stored procedures and applications that use WORK can be changed to run against the new table name.[1] If all is correct, then the foreign key constraint TransWorkFK and the WORK table can be dropped with the following:

```
ALTER TABLE TRANS DROP CONSTRAINT TransWorkFK;
DROP TABLE WORK;
```

The TransWorkFK constraint then can be added back to TRANS using the new name for the work table:

```
ALTER TABLE TRANS ADD CONSTRAINT TransWorkFK FOREIGN KEY(WorkID)
      REFERENCES WORK_VERSION2(WorkID)
            ON UPDATE NO ACTION
            ON DELETE NO ACTION;
```

Clearly, there is more to changing a table name than you would think. You now can see why some organizations do not allow programmers or users to employ the true name of a table. Instead, views are described that serve as table aliases, as explained in Chapter 7. If this were done here, only the views that define the aliases would need to be changed when the source table name is changed.

Adding and Dropping Columns

Adding null columns to a table is straightforward. For example, to add the null column DateCreated to WORK, we simply use the ALTER statement as follows:

```
ALTER TABLE WORK
      ADD DateCreated DateTime NULL;
```

If there are other column constraints, such as DEFAULT or UNIQUE, include them with the column definition, just as you would if the column definition were part of a CREATE TABLE statement. However, if you include a DEFAULT constraint, be aware that the default value will be applied to all new rows, but existing rows will have null values.

Suppose, for example, that you want to set the default value of DateCreated to 1/1/1900 to signify that the value has not yet been entered. In this case, you would use the ALTER statement:

```
ALTER TABLE WORK
      ADD DateCreated DateTime NULL DEFAULT '01/01/1900';
```

[1] The timing is important. The WORK_VERSION2 table was created from WORK. If triggers, stored procedures, and applications continue to run against WORK while the verification of WORK_VERSION2 is underway, then WORK_VERSION2 will be out-of-date. Some action will need to be taken to bring it up-to-date before switching the stored procedures and applications over to WORK_VERSION2.

Amazon Ex. 1008.0333

This statement causes DateCreated for new rows in WORK to be set to 1/1/1900 by default. To set existing rows, you would need to execute the following query:

```
UPDATE WORK
SET        DateCreated ='01/01/1900'
WHERE      DateCreated IS NULL;
```

Adding NOT NULL Columns

To add a new NOT NULL column, first add the column as NULL. Then, use an UPDATE statement like that just shown to give the column a value in all rows. After the update, the following SQL ALTER TABLE . . . ALTER COLUMN statement can be executed to change DateCreated from NULL to NOT NULL.

```
ALTER TABLE WORK
      ALTER COLUMN DateCreated DateTime NOT NULL;
```

Note that this statement will fail if DateCreated has not been given values in all rows.

Dropping Columns

Dropping nonkey columns is easy. For example, eliminating the DateCreated column from WORK can be done with the following:

```
ALTER TABLE WORK
      DROP COLUMN DateCreated;
```

To drop a foreign key column, the constraint that defines the foreign key must first be dropped. Making such a change is equivalent to dropping a relationship, and that topic is discussed later in this chapter.

To drop the primary key, the primary key constraint first needs to be dropped. To drop that, however, all foreign keys that use the primary key must first be dropped. Thus, to drop the primary key of WORK and replace it with the composite primary key (Title, Copy, ArtistID), the following steps are necessary:

- Drop the constraint WorkFK from TRANS.
- Drop the constraint WorkPK from WORK.
- Create a new WorkPK constraint using (Title, Copy, ArtistID).
- Create a new WorkFK constraint referencing (Title, Copy, ArtistID) in TRANS.
- Drop the column WorkID.

It is important to verify that all changes have been made correctly before dropping WorkID. Once it is dropped, there is no way to recover it except by restoring the WORK table from a backup.

Changing a Column Data Type or Column Constraints

To change a column data type or to change column constraints, the column is redefined using the ALTER TABLE ALTER COLUMN command. However, if the column is being changed from NULL to NOT NULL, then all rows must have a value in that column for the change to succeed.

Also, some data type changes may cause data loss. Changing Char(50) to Date, for example, will cause loss of any text field that the DBMS cannot successfully transform into a date value. Or, alternatively, the DBMS may simply refuse to make the column change. The results depend on the DBMS product in use.

Generally, converting numeric to char or varchar will succeed. Also, converting date or money or other more specific data types to char or varchar will usually succeed. Converting char or varchar back to date, money, or numeric is risky, and it may or may not be possible.

Amazon Ex. 1008.0334

In the View Ridge schema, if DateOfBirth had been defined as Char(4), then a risky but sensible data type change would be to modify DateOfBirth in the ARTIST table to numeric (4,0).

This would be a sensible change because all of the values in this column are numeric. Recall the check constraint that was used to define DateOfBirth (refer to Figure 7-14). The following makes that change and simplifies the CHECK constraint.

```
ALTER TABLE ARTIST
      ALTER COLUMN DateOfBirth Numeric(4,0) NULL;
ALTER TABLE ARTIST
      ADD CONSTRAINT NumericBirthYearCheck
            CHECK (Birthdate > 1900 AND Birthdate < 2100);
```

The prior check constraints on DateOfBirth should now be deleted.

Adding and Dropping Constraints

As already shown, constraints can be added and removed using the ALTER TABLE ADD CONSTRAINT and ALTER TABLE DROP CONSTRAINT statements.

Changing Relationship Cardinalities and Properties

Changing cardinalities is a common database redesign task. Sometimes, the need is to change minimum cardinalities from zero to one or from one to zero. Another common task is to change the maximum cardinality from 1:1 to 1:N or from 1:N to N:M. Another possibility, which is less common, is to decrease maximum cardinality from N:M to 1:N or from 1:N to 1:1. This latter change can only be made with data loss, as you will see.

Changing Minimum Cardinalities

The action to be taken in changing minimum cardinalities depends on whether the change is on the parent side or on the child side of the relationship.

Changing Minimum Cardinalities on the Parent Side

If the change is on the parent side, meaning that the child will or will not be required to have a parent, making the change is a matter of changing whether null values are allowed for the foreign key that represents the relationship. For example, suppose that in the 1:N relationship from DEPARTMENT to EMPLOYEE the foreign key DepartmentNumber appears in the EMPLOYEE table. Changing whether an employee is required to have a department is simply a matter of changing the null status of DepartmentNumber.

If the change is from a minimum cardinality of zero to one, then the foreign key, which would have been null, must be changed to NOT NULL. Changing a column to NOT NULL can only be done if all the rows in the table have a value. In the case of a foreign key, this means that every record must already be related. If not, all records must be changed so that all have a relationship before the foreign key can be made NOT NULL. In the previous example, every employee must be related to a department before DepartmentNumber can be changed to NOT NULL.

Depending on the DBMS product in use, the foreign key constraint that defines the relationship may have to be dropped before the change is made to the foreign key. Then the foreign key constraint can be re-added. The following SQL will work for the preceding example:

```
ALTER TABLE EMPLOYEE
      DROP CONSTRAINT DepartmentFK;
ALTER TABLE EMPLOYEE
      ALTER COLUMN DepartmentNumber Int NOT NULL;
```

Amazon Ex. 1008.0335

```
ALTER TABLE EMPLOYEE
      ADD CONSTRAINT DepartmentFK FOREIGN KEY (DepartmentNumber)
            REFERENCES DEPARTMENT (DepartmentNumber)
                  ON UPDATE CASCADE;
```

Also, cascade behavior for UPDATE and DELETE must be specified when changing the minimum cardinality from zero to one. In this example, updates are to cascade, but deletions will not (recall that the default behavior is NO ACTION).

Changing the minimum cardinality from one to zero is simple. Just change Department-Number from NOT NULL to NULL. You also may want to change the cascade behavior on updates and deletions, if appropriate.

Changing Minimum Cardinalities on the Child Side

As noted in Chapter 6, the only way to enforce a minimum cardinality other than zero on the child side of a relationship is to write triggers or application code that enforce the constraint. So, to change the minimum cardinality from zero to one, it is necessary to write the appropriate triggers. Design the trigger behavior using Figure 6-28, and then write the triggers. To change the minimum cardinality from one to zero, just drop the triggers that enforce that constraint.

In the DEPARTMENT-to-EMPLOYEE relationship example, to require each DEPARTMENT to have an EMPLOYEE triggers would need to be written on INSERT of DEPARTMENT and on UPDATE and DELETE of EMPLOYEE. The trigger code in DEPARTMENT ensures that an EMPLOYEE is assigned to the new DEPARTMENT, and the trigger code in EMPLOYEE ensures that the employee being moved to a new department or the employee being deleted is not the last employee in the relationship to its parent.

This discussion assumes that the required child constraint is enforced by triggers. If the required child constraint is enforced by application programs, then all of those programs also must be changed. Dozens of programs may need to be changed, which is one reason why it is better to enforce such constraints using triggers rather than application code.

Changing Maximum Cardinalities

The only difficulty when increasing cardinalities from 1:1 to 1:N or from 1:N to N:M is preserving existing relationships. This can be done, but it requires a bit of manipulation, as you will see. When reducing cardinalities, relationship data will be lost. In this case, a policy must be created for deciding which relationships to lose.

Changing a 1:1 Relationship to a 1:N Relationship

Figure 8-5 shows a 1:1 relationship between EMPLOYEE and PARKING_PERMIT. As we discussed in Chapter 6, the foreign key can be placed in either table for a 1:1 relationship. The action taken depends on whether EMPLOYEE is to be the parent entity in the 1:N relationship or whether PARKING_PERMIT is to be the parent.

If EMPLOYEE is to be the parent (employees are to have multiple parking permits), then the only change necessary is to drop the constraint that PARKING_PERMIT.EmployeeNumber be unique. The relationship will then be 1:N.

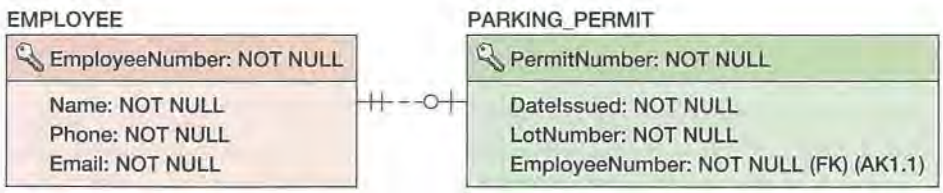


Figure 8-5 The EMPLOYEE-to-PARKING_PERMIT 1:1 Relationship

Amazon Ex. 1008.0336

If PARKING_PERMIT is to be the parent (for example, if parking permits are to be allocated to many employees, say, for a carpool), then the foreign key and appropriate values must be moved from PARKING_PERMIT to EMPLOYEE. The following SQL will accomplish this:

```
ALTER TABLE EMPLOYEE
      ADD PermitNumber Int NULL;
UPDATE EMPLOYEE
      SET EMPLOYEE.PermitNumber =
            (SELECT   PP.PermitNumber
             FROM     PARKING_PERMIT AS PP
             WHERE    PP.EmployeeNumber = EMPLOYEE.EmployeeNumber);
```

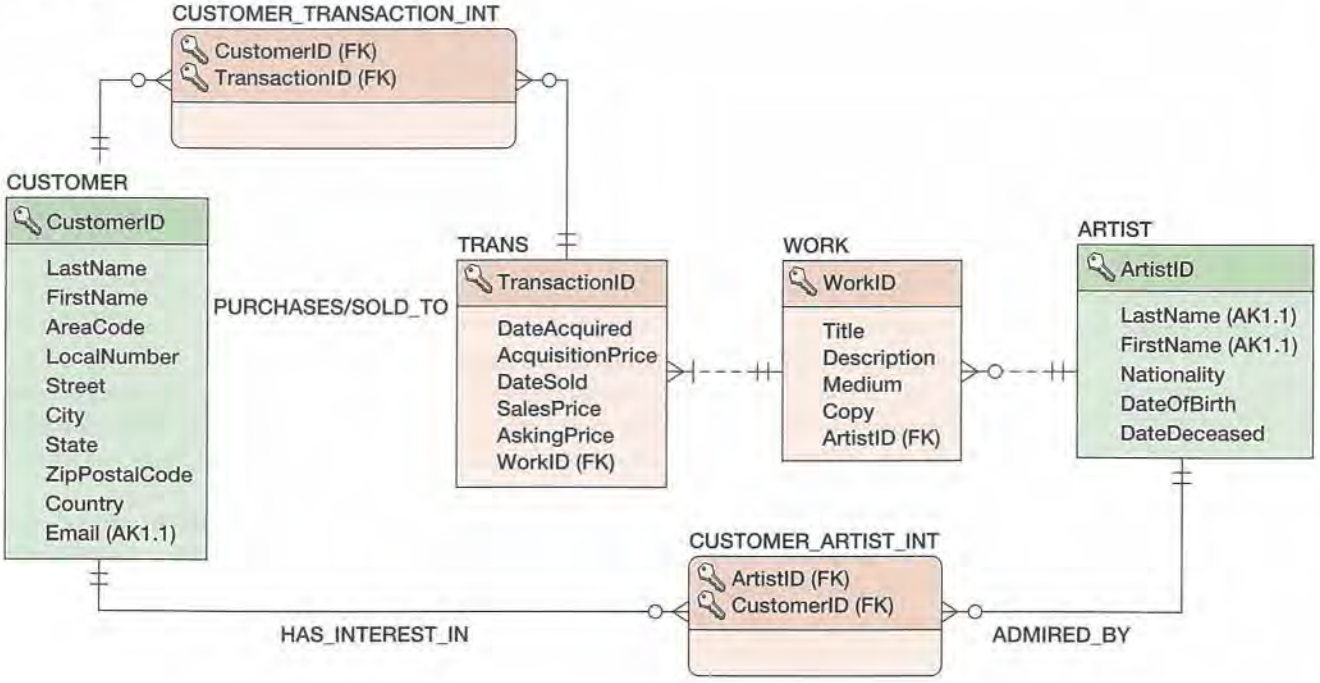
Once the foreign key has been moved over to EMPLOYEE, the EmployeeNumber column of PARKING_PERMIT should be dropped. Next, create a new foreign key constraint to define referential integrity. So that multiple employees can relate to the same parking permit, the new foreign key must not have a UNIQUE constraint.

Changing a 1:N Relationship to an N:M Relationship

Suppose that View Ridge Gallery decides that it wants to record multiple purchasers for a given transaction. It may be that some of its art is co-owned between a customer and a bank or a trust account, for example; or perhaps it may want to record the names of both owners when a couple purchases art. For whatever reason, this change will require that the 1:N relationship between CUSTOMER and TRANS be changed to an N:M relationship.

Changing a 1:N relationship to an N:M relationship is surprisingly easy.[2] Just create the new intersection table, fill it with data, and drop the old foreign key column. Figure 8-6 shows the View Ridge database design with a new intersection table to support the N:M relationship.

Figure 8-6
View Ridge Gallery Database Design with New N:M Relationship



[2] Making the data change is easy. Dealing with the consequences of the data change with regards to views, triggers, stored procedures, and application code will be more difficult. All of these will need to be rewritten to join across a new intersection table. All forms and reports also will need to be changed to portray multiple customers for a transaction; this will mean changing text boxes to grids, for example. All of this work is time consuming, and hence expensive.

Amazon Ex. 1008.0337

We need to create this table and then copy the values of TransactionID and CustomerID from TRANS for rows in which CustomerID is not null. First, create the new intersection table using the following SQL:

```
CREATE TABLE CUSTOMER_TRANSACTION_INT(
      CustomerID       Int      NOT NULL,
      TransactionID    Int      NOT NULL,
      CONSTRAINT       CustomerTransaction
            PK PRIMARY KEY(CustomerID, TransactionID),
      CONSTRAINT Customer_Transaction_Int_Trans_FK
            FOREIGN KEY (TransactionID) REFERENCES TRANS(TransactionID),
      CONSTRAINT Customer_Transaction_Int_Customer_FK
            FOREIGN KEY (CustomerID) REFERENCES CUSTOMER(CustomerID)
      );
```

Note that there is no cascade behavior for updates because CustomerID is a surrogate key. There is no cascade behavior for deletions because of the business policy never to delete data that involve transactions. The next task is to fill the table with data from the TRANS table using the following SQL statement:

```
INSERT INTO CUSTOMER_TRANSACTION_INT (CustomerID, TransactionID)
      SELECT    CustomerID, TransactionID
      FROM      TRANS
      WHERE     CustomerID IS NOT NULL;
```

Once all of these changes have been made, the CustomerID column of TRANS can be dropped.

Reducing Cardinalities (with Data Loss)

It is easy to make the structural changes to reduce cardinalities. To reduce an N:M relationship to 1:N, we just create a new foreign key in the relation that will be the child and fill it with data from the intersection table. To reduce a 1:N relationship to 1:1, we just make the values of the foreign key of the 1:N relationship unique and then define a unique constraint on the foreign key. In either case, the most difficult problem is deciding which data to lose.

Consider the reduction of N:M to 1:N. Suppose, for example, that the View Ridge Gallery decides to keep just one artist interest for each customer. Thus, the relationship will then be 1:N from ARTIST to CUSTOMER. Accordingly, we add a new foreign key column ArtistID to CUSTOMER and set up a foreign key constraint to ARTIST on that customer. The following SQL will accomplish this:

```
ALTER TABLE CUSTOMER
      ADD ArtistID Int NULL;
ALTER TABLE CUSTOMER
      ADD CONSTRAINT ArtistInterestFK FOREIGN KEY (ArtistID)
            REFERENCES ARTIST(ArtistID);
```

Updates need not cascade because of the surrogate key, and deletions cannot cascade because the customer may have a valid transaction and ought not to be deleted just because an artist interest goes away.

Amazon Ex. 1008.0338

Now which of a customer's potentially many artist interests should be preserved in the new relationship? The answer depends on the business policy at the gallery. Here, suppose we decide simply to take the first artist interest:

```
UPDATE      CUSTOMER
SET         ArtistID =
            (SELECT      Top 1 ArtistID
             FROM        CUSTOMER_ARTIST_INT AS CAI
             WHERE       CUSTOMER.CustomerID = CAI.CustomerID);
```

The SQL Top 1 phrase is used to return the first qualifying row.

All views, triggers, stored procedures, and application code need to be changed to account for the new 1:N relationship. Then the constraints defined on CUSTOMER_ ARTIST_INT can be dropped. Finally, the table CUSTOMER_ARTIST_INT can be dropped.

To change a 1:N to a 1:1 relationship, we just need to remove any duplicate values of the foreign key of the relationship and then add a unique constraint on the foreign key. See question 8.51.

Adding and Deleting Tables and Relationships

Adding new tables and relationships is straightforward. Just add the tables and relationships using CREATE TABLE statements with FOREIGN KEY constraints, as shown before. If an existing table has a child relationship to the new table, add a FOREIGN KEY constraint using the existing table.

For example, if a new table, COUNTRY, were added to the View Ridge database with the primary key Name and if CUSTOMER.Country is to be used as a foreign key in the new table, a new FOREIGN KEY constraint would be defined in CUSTOMER:

```
ALTER TABLE CUSTOMER
      ADD CONSTRAINT CountryFK FOREIGN KEY (Country)
            REFERENCES COUNTRY(Name)
                  ON UPDATE CASCADE;
```

Deleting relationships and tables is just a matter of dropping the foreign key constraints and then dropping the tables. Of course, before this is done, dependency graphs must be constructed and used to determine which views, triggers, stored procedures, and application programs will be affected by the deletions.

As described in Chapter 4, another reason to add new tables and relationships or to compress existing tables into fewer tables is for normalization and denormalization. We will not address that topic further in this chapter, except to say that normalization and denormalization are common tasks during database redesign.

Forward Engineering(?)

You can use a variety of different data modeling products to make database changes on your behalf. To do so, you first reverse engineer the database, make changes to the RE data model, and then invoke the forward-engineering functionality of the data modeling tool.

We will not consider forward engineering here because it hides the SQL that you need to learn. Also, the specifics of the forward-engineering process are product dependent.

Because of the importance of making data model changes correctly, many professionals are skeptical about using an automated process for database redesign. Certainly, it is necessary to test the results thoroughly before using forward engineering on operational data. Some

Amazon Ex. 1008.0339

products will show the SQL they are about to execute for review before making the changes to the database.

Database redesign is one area in which automation may not be the best idea. Much depends on the nature of the changes to be made and the quality of the forward-engineering features of the data modeling product. Given the knowledge you have gained in this chapter, you should be able to make most redesign changes by writing your own SQL. There is nothing wrong with that approach!

Summary

Database redesign is the third way in which databases can arise. Redesign is necessary both to fix mistakes made during the initial database design and also to adapt the database to changes in system requirements. Such changes are common because information systems and organizations do not just influence each other—they create each other. Thus, new information systems cause changes in systems requirements.

Correlated subqueries and EXISTS/NOT EXISTS are important SQL statements. They can be used to answer advanced queries. They also are useful during database redesign for determining whether specified data conditions exist. For example, they can be used to determine whether possible functional dependencies exist in the data.

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 distinction of a correlated subquery is that the lower-level SELECT statements use columns from upper-level statements.

EXISTS and NOT EXISTS are 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. NOT EXISTS is useful for queries that involve conditions that must be true for all rows, such as a "customer who has purchased all products." The double use of NOT EXISTS is a famous SQL pattern that often is used to test a person's knowledge of SQL.

Before redesigning a database, the existing database needs to be carefully examined to avoid making the database unusable by partially processing a database change. The rule is to measure twice and cut once. Reverse engineering is used to create a data model of the existing database. This is done to better understand the database structure before proceeding with a change. The data model produced, called a reverse engineered (RE) data model, is not a true data model, but is a thing unto itself. Most data modeling tools can perform reverse engineering. The RE data model almost always has missing information; such models should be carefully reviewed.

All of the elements of a database are interrelated. Dependency graphs are used to portray the dependency of one element on another. For example, a change in a table can potentially impact relationships, views, indexes, triggers, stored procedures, and application programs. These impacts need to be known and accounted for before making database changes.

A complete backup must be made to the operational database prior to any database redesign changes. Additionally, such changes must be thoroughly tested, initially on small test databases and later on larger test databases that may even be duplicates of the operational databases. The redesign changes are made only after such extensive testing has been completed.

Database redesign changes can be grouped into different types. One type involves changing table names and table columns. Changing a table name has a surprising number of potential consequences. A dependency graph should be used to understand these consequences before proceeding with the change. Nonkey columns are readily added and deleted. Adding a NOT NULL column must be done in three steps: first, add the column as NULL; then add data to every row; and then alter the column constraint to NOT NULL. To drop a column used as a foreign key, the foreign key constraint must first be dropped.

Column data types and constraints can be changed using the ALTER TABLE . . . ALTER COLUMN statement. Changing the data type to char or varchar from a more specific type, such as date, is usually not a problem. Changing a data type from char or varchar to a more specific type can be a problem. In some cases, data will be lost or the DBMS may refuse the change.

Constraints can be added or dropped using the ALTER TABLE . . . ADD/DROP CONSTRAINT statement. Use of this statement is easier if the developers have provided their own names for all constraints.

Changing minimum cardinalities on the parent side of a relationship is simply a matter of altering the constraint on the foreign key from NULL to NOT NULL or from NOT NULL to NULL. Changing minimum cardinalities on the child side of a relationship can be accomplished only by adding or dropping triggers that enforce the constraint.

Amazon Ex. 1008.0340

Changing maximum cardinality from 1:1 to 1:N is simple if the foreign key resides in the correct table. In that case, just remove the unique constraint on the foreign key column. If the foreign key resides in the wrong table for this change, move the foreign key to the other table and do not place a unique constraint on that table.

Changing a 1:N relationship to an N:M relationship requires building a new intersection table and moving the primary key and foreign key values to the intersection table. This aspect of the change is relatively simple. It is more difficult to change all of the views, triggers, stored procedures, application programs, and forms and reports to use the new intersection table.

Reducing cardinalities is easy, but such changes may result in data loss. Prior to making such reductions, a policy must be determined to decide which data to keep. Changing N:M to 1:N involves creating a foreign key in the parent table and moving one value from the intersection table into that foreign key. Changing 1:N to 1:1 requires first eliminating duplicates in the foreign key and then setting a uniqueness constraint on that key. Adding and deleting relationships can be accomplished by defining new foreign key constraints or by dropping existing foreign key constraints.

Most data modeling tools have the capacity to perform forward engineering, which is the process of applying data model changes to an existing database. If forward engineering is used, the results should be thoroughly tested before using it on an operational database. Some tools will show the SQL that they will execute during the forward-engineering process. Any SQL generated by such tools should be carefully reviewed. All in all, there is nothing wrong with writing database redesign SQL statements by hand rather than using forward engineering.

Key Terms

correlated subquery
dependency graph
reverse engineered (RE) data model
SQL EXISTS keyword
SQL NOT EXISTS keyword

Review Questions

8.1 Explain, one more time, the three ways that databases arise.

8.2 Describe why database redesign is necessary.

8.3 Explain the following statement in your own words: "Information systems and organizations create each other." How does this relate to database redesign?

8.4 Suppose that a table contains two nonkey columns: AdviserName and AdviserPhone. Further suppose that you suspect that AdviserPhone → AdviserName. Explain how to examine the data to determine if this supposition is true.

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

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

8.7 Write a correlated subquery, other than one in this chapter.

8.8 Explain how the query in your answer to Review Question 8.5 differs from the query in your answer to Review Question 8.7.

8.9 Explain what is wrong with the correlated subquery on page 299.

8.10 Write a correlated subquery to determine whether the data support the supposition in Review Question 8.4.

8.11 Explain the meaning of the SQL keyword EXISTS.

8.12 Answer Review Question 8.10, but use the SQL EXISTS keyword.

8.13 Explain how the words *any* and *all* pertain to the SQL keywords EXISTS and NOT EXISTS.

Amazon Ex. 1008.0341

8.14 Explain the processing of the query on page 301.

8.15 Using the View Ridge Gallery database, write a query that will display the names of any customers who are interested in all artists.

8.16 Explain how the query in your answer to Review Question 8.15 works.

8.17 Why is it important to analyze the database before implementing database redesign tasks? What can happen if this is not done?

8.18 Explain the process of reverse engineering.

8.19 Why is it important to carefully evaluate the results of reverse engineering?

8.20 What is a dependency graph? What purpose does it serve?

8.21 Explain the dependencies for WORK in the graph in Figure 8-4.

8.22 What sources are used when creating a dependency graph?

8.23 Explain two different types of test databases that should be used when testing database redesign changes.

8.24 Explain the problems that can occur when changing the name of a table.

8.25 Describe the process of changing a table name.

8.26 Considering Figure 8-4, describe the tasks that need to be accomplished to change the name of the table WORK to WORK_VERSION2.

8.27 Explain how views can simplify the process of changing a table name.

8.28 Under what conditions is the following SQL statement valid?

```
INSERT    INTO T1    (A, B)
          SELECT     (C, D) FROM T2;
```

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

8.30 Extend your answer to Review Question 8.29 to add C1 when C1 is to be NOT NULL.

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

8.32 Describe the process for dropping primary key C1 and making the new primary key C2.

8.33 Which data type changes are the least risky?

8.34 Which data type changes are the most risky?

8.35 Write an SQL statement to change a column C1 to Char(10) NOT NULL. What conditions must exist in the data for this change to be successful?

8.36 Explain how to change the minimum cardinality when a child that was required to have a parent is no longer required to have one.

8.37 Explain how to change the minimum cardinality when a child that was not required to have a parent is now required to have one. What condition must exist in the data for this change to work?

8.38 Explain how to change the minimum cardinality when a parent that was required to have a child is no longer required to have one.

8.39 Explain how to change the minimum cardinality when a parent that was not required to have a child is now required to have one.

8.40 Describe how to change the maximum cardinality from 1:1 to 1:N. Assume that the foreign key is on the side of the new child in the 1:N relationship.

Amazon Ex. 1008.0342

8.41 Describe how to change the maximum cardinality from 1:1 to 1:N. Assume that the foreign key is on the side of the new parent in the 1:N relationship.

8.42 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.

8.43 Explain how to transform a 1:N relationship into an N:M relationship.

8.44 Suppose that tables T1 and T2 have a 1:N relationship. Show the SQL statements necessary to fill an intersection T1_T2_INT. Make up your own names for the primary and foreign keys.

8.45 Explain how the reduction of maximum cardinalities causes data loss.

8.46 Using the tables in your answer to Review Question 8.44, show the SQL statements necessary to change the relationship back to 1:N. Assume that the first row in the qualifying rows of the intersection table is to provide the foreign key. Use the keys and foreign keys from your answer to Review Question 8.44.

8.47 Using the results of your answer to Review Question 8.46, explain what must be done to convert this relationship to 1:1. Use the keys and foreign keys from your answer to Review Question 8.46.

8.48 In general terms, what must be done to add a new relationship?

8.49 Suppose that tables T1 and T2 have a 1:N relationship, with T2 as the child. Show the SQL statements necessary to remove table T1. Make your own assumptions about the names of keys and foreign keys.

8.50 What are the risks and problems of forward engineering?

Project Questions

8.51 Suppose that the table EMPLOYEE has a 1:N relationship to the table PHONE_NUMBER. Further suppose that the primary key of EMPLOYEE is EmployeeID and the columns of PHONE_NUMBER are PhoneNumberID (a surrogate key), AreaCode, LocalNumber, and EmployeeID (a foreign key to EMPLOYEE). Alter this design so that EMPLOYEE has a 1:1 relationship to PHONE_NUMBER. For employees having more than one phone number, keep only the first one.

8.52 Suppose that the table EMPLOYEE has a 1:N relationship to the table PHONE_NUMBER. Further suppose that the key of EMPLOYEE is EmployeeID and the columns of PHONE_NUMBER are PhoneNumberID (a surrogate key), AreaCode, LocalNumber, and EmployeeID (a foreign key to EMPLOYEE). Write all SQL statements necessary to redesign this database so that it has just one table. Explain the difference between the result of question 8.51 and the result of this question.

8.53 Consider the following table:

TASK (EmployeeID, EmpLastName, EmpFirstName, Phone, OfficeNumber, ProjectName, Sponsor, WorkDate, HoursWorked)

with the following possible functional dependencies:

EmployeeID → (EmpLastName, EmpFirstName, Phone, OfficeNumber)

ProjectName → Sponsor

Amazon Ex. 1008.0343

A. Write SQL statements to display the values of any rows that violate these functional dependencies.

B. If no data violate these functional dependencies, can we assume that they are valid? Why or why not?

C. Assume that these functional dependencies are true and that the data have been corrected, as necessary, to reflect them. Write all SQL statements necessary to redesign this table into a set of tables in BCNF and 4NF. Assume that the table has data values that must be appropriately transformed to the new design.



Assume that Marcia has created a database with the tables described at the end of Chapter 7:

CUSTOMER (CustomerID, FirstName, LastName, Phone, Email)

ORDER (InvoiceNumber, *CustomerID*, DateIn, DateOut, Subtotal, Tax, TotalAmount)

ORDER_ITEM (*InvoiceNumber*, ItemNumber, *Service*, Quantity, UnitPrice, ExtendedPrice)

SERVICE (Service, Description, UnitPrice)

Assume that all relationships have been defined, as implied by the foreign keys in this table list.

A. Create a dependency graph that shows dependencies among these tables. Explain how you need to extend this graph for views and other database constructs such as triggers and stored procedures.

B. Using your dependency graph, describe the tasks necessary to change the name of the ORDER table to CUST_ORDER.

C. Write all SQL statements to make the name change described in part B.

D. Suppose that Marcia decides to allow multiple customers per order (for customers' spouses, for example). Modify the design of these tables to accommodate this change.

E. Code SQL statements necessary to redesign the database, as described in your answer to part D.

F. Suppose that Marcia considers changing the primary key of CUSTOMER to (FirstName, LastName). Write correlated subqueries to display any data that indicate that this change is not justifiable.

G. Suppose that (FirstName, LastName) can be made the primary key of CUSTOMER. Make appropriate changes to the table design with this new primary key.

H. Code all SQL statements necessary to implement the changes described in part G.

Amazon Ex. 1008.0344



Assume that Morgan has created a database with the tables described at the end of Chapter 7 (Note that STORE uses the surrogate key StoreID):

STORE (StoreID, StoreName, City, Country, Phone, Fax, Email, Contact)

PURCHASE (PurchaseID, *StoreID*, Date, Description, Category, PriceUSD)

SHIPMENT (ShipmentID, *ShipperID*, ShipperInvoiceNumber, Origin, Destination, DepartureDate, ArrivalDate)

SHIPMENT_ITEM (*ShipmentID*, *PurchaseID*, InsuredValue)

SHIPPER (ShipperID, ShipperName, Phone, Fax, Email, Contact)

Assume that all relationships have been defined as implied by the foreign keys in this table list.

A. Create a dependency graph that shows dependencies among these tables. Explain how you need to extend this graph for views and other database constructs, such as stored procedures.

B. Using your dependency graph, describe the tasks necessary to change the name of the SHIPMENT table to MORGAN_SHIPMENT.

C. Write all SQL statements to make the name change described in part B.

D. Suppose that Morgan decides to allocate some purchases to more than one shipment. Make design changes in accordance with this new fact. You will need to make assumptions about how purchases are divided and allocated to shipments. State your assumptions.

E. Code SQL statements to implement your redesign recommendations in your answer to part D.

F. Suppose that Morgan considers changing the primary key of PURCHASE to (StoreID, Date). Write correlated subqueries to display any data that indicate that this change is not justifiable.

G. Suppose that (StoreID, Date) can be made the primary key of PURCHASE. Make appropriate changes to the table design.

H. Code all SQL statements necessary to implement the changes described in part G.

Amazon Ex. 1008.0345

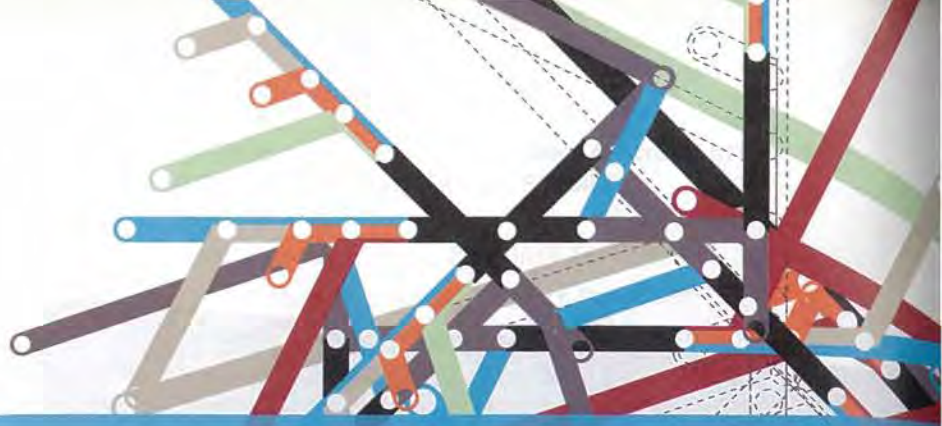
Part 4

Multiuser Database Processing

The four chapters in Part 4 introduce and discuss the major problems of multiuser database processing and describe the features and functions for solving those problems offered by two important DBMS products. We begin in Chapter 9 with a description of database administration and the major tasks and techniques for multiuser database management. The next three chapters illustrate the implementation of these concepts using SQL Server 2008 (Chapter 10), Oracle Database 11g (Chapter 10A), and MySQL 5.1 (Chapter 10B).



Amazon Ex. 1008.0346



9 Managing Multiuser Databases

Chapter Objectives

- To understand the need for and importance of database administration.
- To understand the need for concurrency control, security, and backup and recovery.
- To learn about typical problems that can occur when multiple users process a database concurrently.
- To understand the use of locking and the problem of deadlock.
- To learn the difference between optimistic and pessimistic locking.
- To know the meaning of an ACID transaction.
- To learn the four 1992 ANSI standard isolation levels.
- To understand the need for security and specific tasks for improving database security.
- To know the difference between recovery via reprocessing and recovery via rollback/rollforward.
- To understand the nature of the tasks required for recovery using rollback/rollforward.
- To know basic administrative and managerial DBA functions.

Although multiuser databases offer great value to the organizations that create and use them, they also pose difficult problems for those same organizations. For one, multiuser databases are complicated to design and develop because they support many overlapping user views.

Additionally, as discussed in the last chapter, requirements change over time, and those changes necessitate other changes to the database structure. Such structural changes must be carefully planned and controlled so that a change made for one group does not cause problems for another. In addition,

Amazon Ex. 1008.0347

when users process a database concurrently, special controls are needed to ensure that the actions of one user do not inappropriately influence the results for another. This topic is both important and complicated, as you will see.

In large organizations, processing rights and responsibilities need to be defined and enforced. What happens, for example, when an employee leaves the firm? When can the employee's records be deleted? For the purposes of payroll processing, records can be deleted after the last pay period. For the purposes of quarterly reporting, they can be deleted at the end of the quarter. For the purposes of end-of-year tax record processing, they can be deleted at the end of the year. Clearly, no department can unilaterally decide when to delete that data. Similar comments pertain to the insertion and changing of data values. For these and other reasons, security systems need to be developed that enable only authorized users to take authorized actions at authorized times.

Databases have become key components of organizational operations, and even key components of an organization's value. Unfortunately, database failures and disasters do occur. Thus, effective backup and recovery plans, techniques, and procedures are essential.

Finally, over time, the DBMS itself will need to be changed to improve performance by incorporating new features and releases and to conform with changes made in the underlying operating system. All of this requires attentive management.

To ensure that these problems are addressed and solved, most organizations have a database administration office. We begin with a description of the tasks of that office. We then describe the combination of software and manual practices and procedures that are used to perform those tasks. In the next three chapters, we will discuss and illustrate features and functions of SQL Server 2008, Oracle Database 11g, and MySQL 5.1, respectively, for dealing with these issues.

Database Administration

The terms **data administration** and **database administration** are both used in practice. In some cases, the terms are considered to be synonymous; in other cases, they have different meanings. Most commonly, the term *data administration* refers to a function that applies to an entire organization; it is a management-oriented function that concerns corporate data privacy and security issues. In contrast, the term *database administration* refers to a more technical function that is specific to a particular database, including the applications that process that database. This chapter addresses database administration.

Databases vary considerably in size and scope, from single-user personal databases to large interorganizational databases, such as airline reservation systems. All of these databases have a need for database administration, although the tasks to be accomplished vary in complexity. For personal databases, individuals follow simple procedures for backing up their data, and they keep minimal records for documentation. In this case, the person who uses the database also performs the database administration functions, even though he or she is probably unaware of it.

Amazon Ex. 1008.0348

Summary of Database Administration Tasks
• Manage database structure
• Control concurrent processing
• Manage processing rights and responsibilities
• Develop database security
• Provide for database recovery
• Manage the DBMS
• Maintain the data repository

Figure 9-1
Summary of Database Administration Tasks

For multiuser database applications, database administration becomes both more important and more difficult. Consequently, it generally has formal recognition. For some applications, one or two people are given this function on a part-time basis. For large Internet or intranet databases, database administration responsibilities are often too time consuming and too varied to be handled even by a single full-time person. Supporting a database with dozens or hundreds of users requires considerable time as well as both technical knowledge and diplomatic skills. Such support usually is handled by an office of database administration. The manager of the office is often known as the **database administrator**. In this case, the acronym **DBA** refers to either the office or the manager.

The overall responsibility of the DBA is to facilitate the development and use of the database. Usually, this means balancing the conflicting goals of protecting the database and maximizing its availability and benefit to users. Specific tasks are shown in Figure 9-1. We consider each of these tasks in the following sections.

Managing the Database Structure

Managing the database structure includes participating in the initial database design and implementation as well as controlling and managing changes to the database. Ideally, the DBA is involved early in the development of the database and its applications; participates in the requirements study; helps evaluate alternatives, including the DBMS to be used; and helps design the database structure. For large organizational applications, the DBA usually is a manager who supervises the work of technically oriented database design personnel.

Creating the database involves several different tasks. First, the database is created and disk space is allocated for database files and logs. Then tables are generated, indexes are created, and stored procedures and triggers are written. We will discuss examples of all of these tasks in the next three chapters. Once the database structures are created, the database is filled with data.

Configuration Control

After a database and its applications have been implemented, changes in requirements are inevitable, as described in Chapter 8. Such changes can arise from new needs, from changes in the business environment, from changes in policy, and from changes in business processes that evolve with system use. When changes to requirements necessitate changes to the database structure, great care must be used, because changes to the database structure seldom involve just one application.

Hence, effective database administration includes procedures and policies by which users can register their needs for changes, the entire database community can discuss the impacts of the changes, and a global decision can be made whether to implement proposed changes. Because of the size and complexity of a database and its applications, changes sometimes have unexpected results. Thus, the DBA must be prepared to repair the database and to gather sufficient information to diagnose and correct the problem that caused the damage. The database is most vulnerable to failure after its structure has been changed.

Amazon Ex. 1008.0349

Participate in Database and Application Development
• Assist in requirements stage and data model creation
• Play an active role in database design and creation
Facilitate Changes to Database Structure
• Seek communitywide solutions
• Assess impact on all users
• Provide configuration control forum
• Be prepared for problems after changes are made
• Maintain documentation

Figure 9-2
Summary of DBA's Responsibilities for Managing Database Structure

Documentation

The DBA's final responsibility in managing the database structure is documentation. It is extremely important to know what changes have been made, how they were made, and when they were made. A change in the database structure may cause an error that is not revealed for six months; without proper documentation of the change, diagnosing the problem is next to impossible. Considerable work may be required to identify the point at which certain symptoms first appeared. For this reason, it also is important to maintain a record of the test procedures and test runs made to verify a change. If standardized test procedures, test forms, and recordkeeping methods are used, recording the test results does not have to be time consuming.

Although maintaining documentation is tedious and unfulfilling, the effort pays off when disaster strikes and the documentation is the difference between a quick problem solution and a confused muddle of activity. Today, several products are emerging that ease the burden of documentation. Many CASE tools, for example, can be used to document logical database designs. Version-control software can be used to track changes. Data dictionaries provide reports and other outputs that present database data structures.

Another reason for carefully documenting changes in the database structure is so that historical data are used properly. If, for example, marketing wants to analyze three-year-old sales data that have been in the archives for two years, it will be necessary to know what structure was current at the time the data were last active. Records that show the changes in the structure can be used to answer that question. A similar situation arises when a six-month-old backup copy of data must be used to repair a damaged database (although this should not happen, it sometimes does). The backup copy can be used to reconstruct the database to the state it was in at the time of the backup. Then, transactions and structural changes can be made in chronological order to restore the database to its current state. Figure 9-2 summarizes the DBA's responsibilities for managing the database structure.

Concurrency Control

Concurrency control measures are taken to ensure that one user's work does not inappropriately influence another user's work. In some cases, these measures ensure that a user gets the same result when processing with other users that he or she would have received if processing alone. In other cases, it means that the user's work is influenced by other users, but in an anticipated way. For example, in an order-entry system, a user should be able to enter an order and get the same result, regardless of whether there are no other users or hundreds of other users. In contrast, a user who is printing a report of the most current inventory status may want to obtain in-process data changes from other users, even if there is a danger that those changes may later be cancelled.

Amazon Ex. 1008.0350

Unfortunately, no concurrency control technique or mechanism is ideal for every circumstance. All involve trade-offs. For example, a program can obtain very strict concurrency control by locking the entire database, but no other programs will be able to do anything while it runs. This is strict protection, but at a high cost. As you will see, other measures are available that are more difficult to program or enforce but that allow more throughput. Still other measures are available that maximize throughput but have a low level of concurrency control. When designing multiuser database applications, you will need to choose among these trade-offs.

The Need for Atomic Transactions

In most database applications, users submit work in the form of **transactions**, which are also known as **logical units of work (LUWs)**. A transaction (or LUW) is a series of actions to be taken on the database so that either all of them are performed successfully or none of them are performed at all, in which case the database remains unchanged. Such a transaction is sometimes called *atomic* because it is performed as a unit.

Consider the following sequence of database actions that could occur when recording a new order:

1. Change a customer's row, increasing AmountDue.
2. Change a salesperson's row, increasing CommissionDue.
3. Insert a new order row into the database.

Suppose that the last step failed, perhaps because of insufficient file space. Imagine the confusion if the first two changes were made but the third one was not. The customer would be billed for an order never received, and a salesperson would receive a commission on an order that was never sent to the customer. Clearly, these three actions need to be taken as a unit—either all of them should be done or none of them should be done.

Figure 9-3 compares the results of performing these activities as a series of independent steps [Figure 9-3(a)] and as an atomic transaction [Figure 9-3(b)]. Notice that when the steps are carried out atomically and one fails, no changes are made in the database. Also note that the commands Start Transaction, Commit Transaction, and Rollback Transaction are issued by the application program to mark the boundaries of the transaction logic. You will learn more about these commands later in this chapter and in Chapters 10, 10A, and 10B.

Concurrent Transaction Processing

When two transactions are being processed against a database at the same time, they are termed **concurrent transactions**. Although it may appear to the users that concurrent transactions are being processed simultaneously, this cannot be true because the CPU of the machine processing the database can execute only one instruction at a time. Usually, transactions are interleaved, which means that the operating system switches CPU services among tasks so that some portion of each transaction is carried out in a given interval. This switching among tasks is done so quickly that two people seated at browsers side by side, processing the same database, may believe that their two transactions are completed simultaneously; in reality, however, the two transactions are interleaved.

Figure 9-4 shows two concurrent transactions. User A's transaction reads Item 100, changes it, and rewrites it in the database. User B's transaction takes the same actions, but on Item 200. The CPU processes User A's transactions until it encounters an I/O interrupt or some other delay for User A. The operating system shifts control to User B. The CPU now processes User B's transactions until an interrupt, at which point the operating system passes control back to User A. To the users, the processing appears to be simultaneous, but it is interleaved, or concurrent.

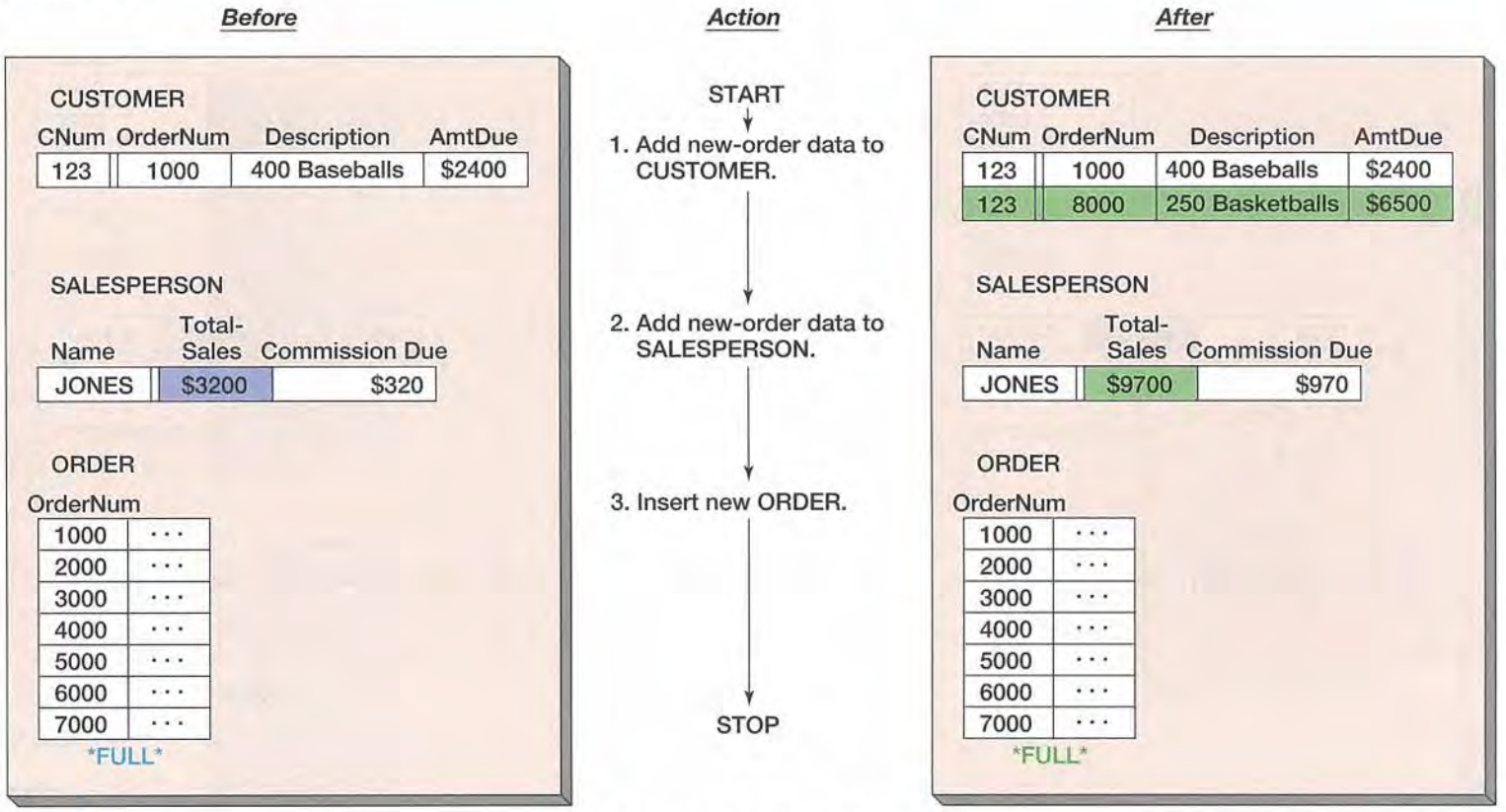
The Lost Update Problem

The concurrent processing illustrated in Figure 9-4 poses no problems because the users are processing different data. But suppose that both users want to process Item 100. For example,

Amazon Ex. 1008.0351

Figure 9-3

Need for Transaction Processing

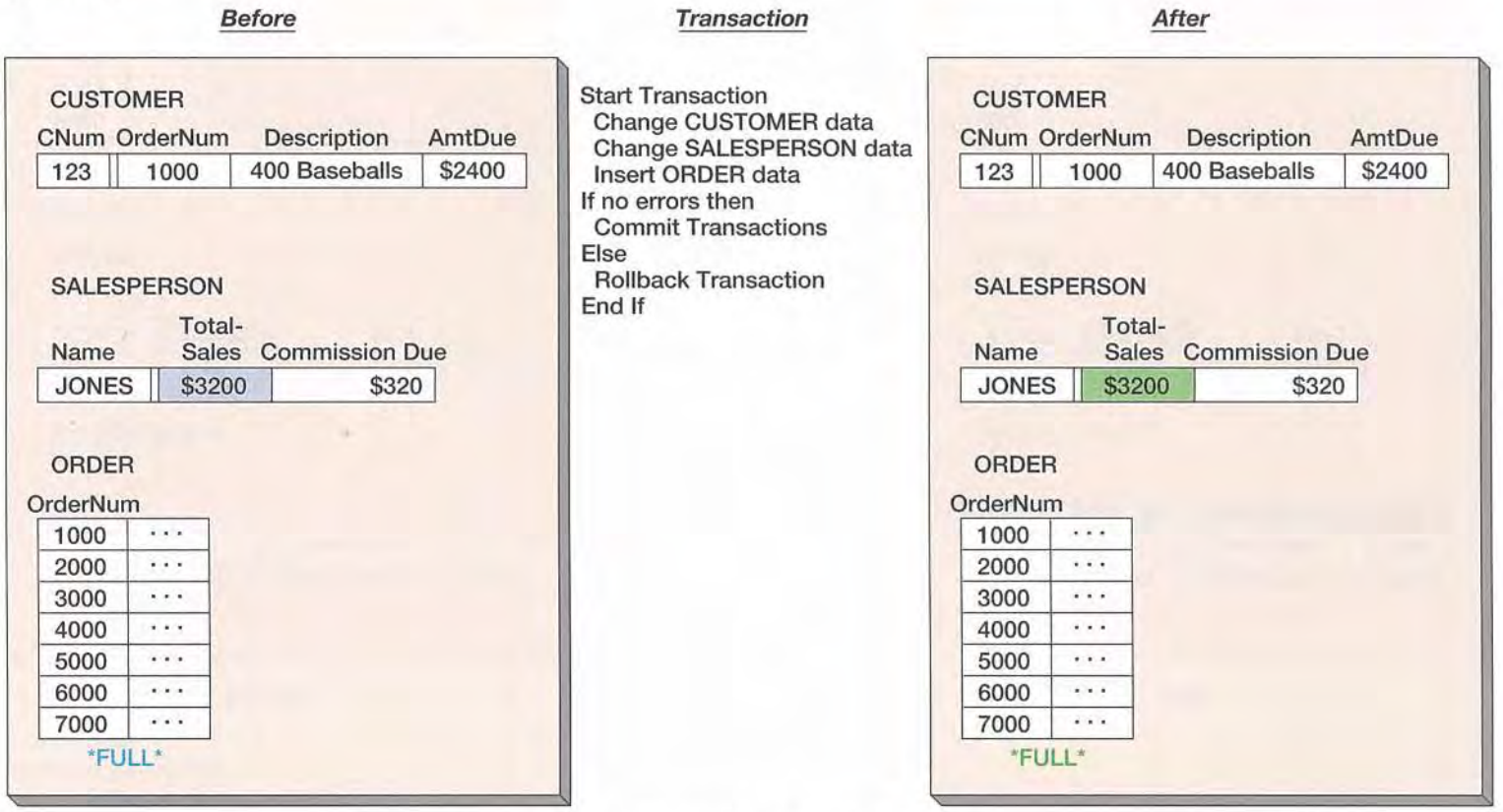


(a) Errors Introduced Without Transaction

Amazon Ex. 1008.0352

Figure 9-3

(Continued)



(b) Atomic Transaction Prevents Errors

Amazon Ex. 1008.0353

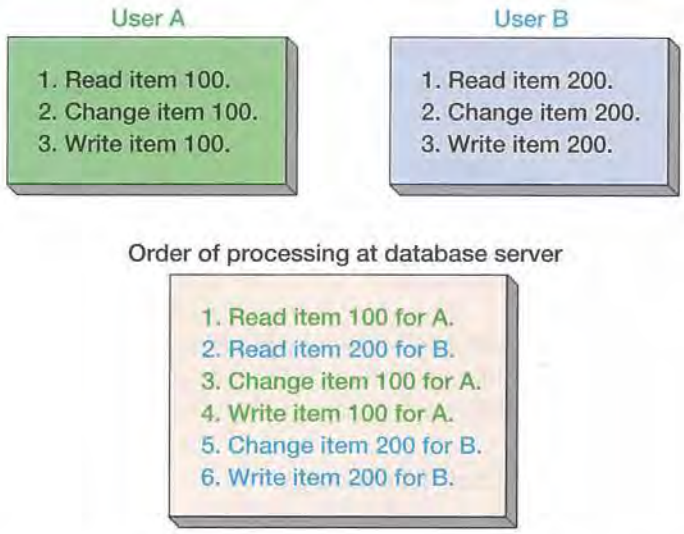


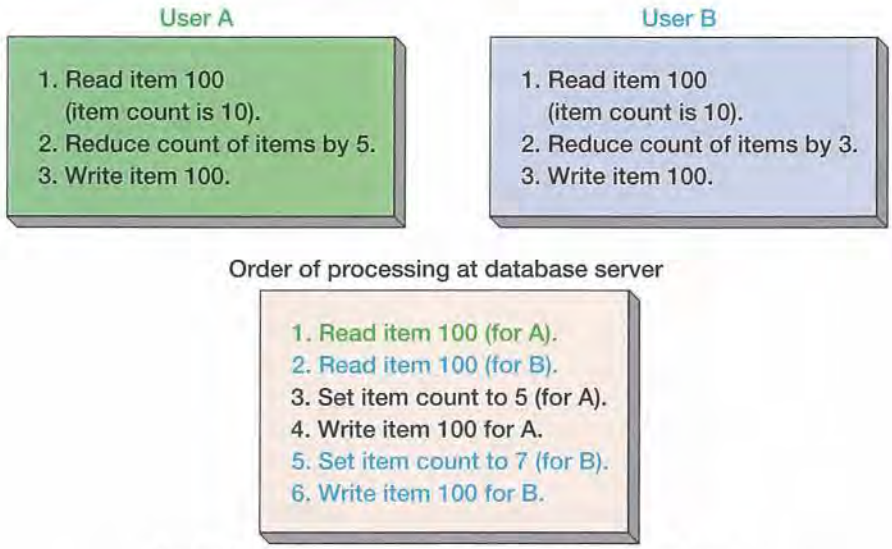
Figure 9-4
Concurrent-Processing Example

User A wants to order five units of Item 100, and User B wants to order three units of the same item. Figure 9-5 illustrates the problem.

User A reads a copy of Item 100's record into memory. According to the record, there are 10 items in inventory. Then User B reads another copy of Item 100's record into a different section of memory. Again, according to the record, there are 10 items in inventory. Now User A takes five, decrements the count of items in its copy of the data to five, and rewrites the record for Item 100. Then User B takes three, decrements the count in its copy of the data to seven, and rewrites the record for Item 100. The database now shows, incorrectly, that there are seven Item 100s in inventory. To review: We started with 10 in inventory, User A took five, User B took three, and the database shows that seven are in inventory. Clearly, this is a problem.

Both users obtained data that were correct at the time they obtained them. But when User B read the record, User A already had a copy that it was about to update. This situation is called the **lost update problem**, or the **concurrent update problem**. A similar problem is the **inconsistent read problem**. With this problem, User A reads data that have been processed by a portion of a transaction from User B. As a result, User A reads incorrect data.

Figure 9-5
Lost-Update Problem



Note: The change and write in steps 3 and 4 are lost.

Amazon Ex. 1008.0354

One remedy for the inconsistencies caused by concurrent processing is to prevent multiple applications from obtaining copies of the same record when the record is about to be changed. This remedy is called **resource locking**.

Resource Locking

One way to prevent concurrent processing problems is to disallow sharing by locking data that are retrieved for update. Figure 9-6 shows the order of processing using a **lock** command.

Because of the lock, User B's transaction must wait until User A is finished with the Item 100 data. Using this strategy, User B can read Item 100's record only after User A has completed the modification. In this case, the final item count stored in the database is two, as it should be. (We started with ten, User A took five, and User B took three, leaving two.)

Lock Terminology

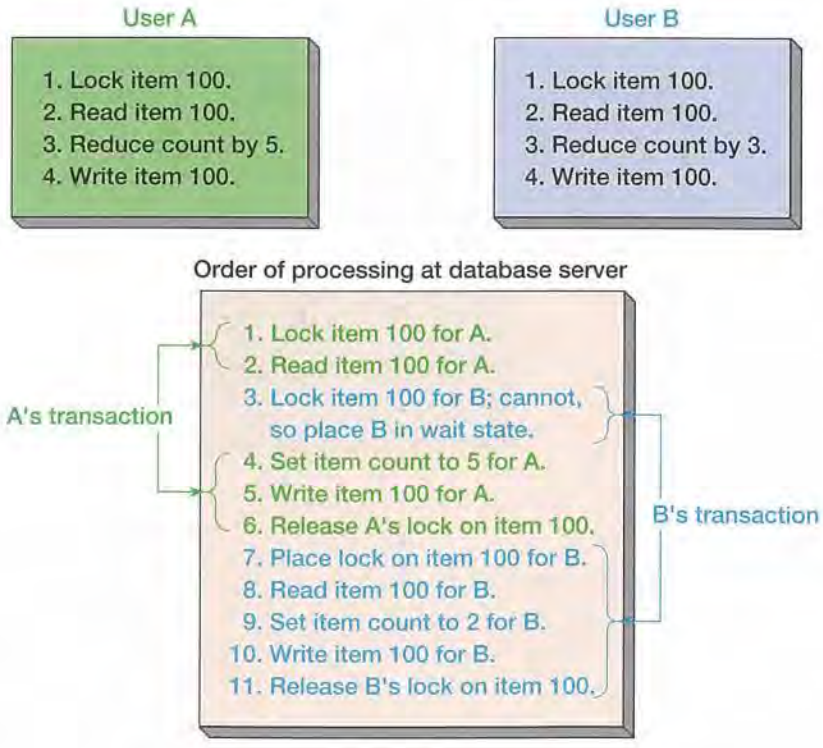
Locks can be placed either automatically by the DBMS or by a command issued to the DBMS from the application program. Locks placed by the DBMS are called **implicit locks**; those placed by command are called **explicit locks**. Today, almost all locking is implicit. The program declares the behavior it wants, and the DBMS places locks accordingly. You will learn how to do that later in this chapter.

In the preceding example, the locks were applied to rows of data. Not all locks are applied at this level, however. Some DBMS products lock groups of rows within a table, some lock entire tables, and some lock the entire database. The size of a lock is referred to as **lock granularity**. Locks with large granularity are easy for the DBMS to administer, but frequently cause conflicts. Locks with small granularity are difficult to administer (the DBMS has to track and check many more details), but conflicts are less common.

Locks also vary by type. An **exclusive lock** locks the item from any other access. No other transaction can read or change the data. A **shared lock** locks the item from change but not from read. That is, other transactions can read the item as long as they do not attempt to alter it.

Figure 9-6

Concurrent Processing with Explicit Locks



Amazon Ex. 1008.0355

Serializable Transactions

When two or more transactions are processed concurrently, the results in the database should be logically consistent with the results that would have been achieved had the transactions been processed in an arbitrary, serial fashion. A scheme for processing concurrent transactions in this way is said to be **serializable**.

Serializability can be achieved by a number of different means. One way is to process the transaction using **two-phased locking**. With this strategy, transactions are allowed to obtain locks as necessary, but once the first lock is released, no other lock can be obtained. Transactions thus have a **growing phase**, during which the locks are obtained, and a **shrinking phase**, during which the locks are released.

A special case of two-phased locking is used with a number of DBMS products. With it, locks are obtained throughout the transaction, but no lock is released until the COMMIT or ROLLBACK command is issued. This strategy is more restrictive than two-phase locking requires, but it is easier to implement.

Consider an order-entry transaction that processes data in the CUSTOMER, SALESPERSON, and ORDER tables. To avoid concurrency problems, the order-entry transaction issues locks on CUSTOMER, SALESPERSON, and ORDER, as needed; makes all database changes; and then releases all locks.

Deadlock

Although locking solves one problem, it introduces another. Consider what can happen when two users want to order two items from inventory. Suppose that User A wants to order some paper, and if she can get the paper, she wants to order some pencils. Then suppose that User B wants to order some pencils, and if he can get the pencils, he wants to order some paper. The order of processing is shown in Figure 9-7.

In this figure, Users A and B are locked in a condition known as **deadlock**, or sometimes as the **deadly embrace**. Each user is waiting for a resource that the other has locked. This problem can be solved either by preventing the deadlock from occurring or by allowing the deadlock to occur and then breaking it.

Deadlock can be prevented in several ways. One way is to require users to issue all lock requests at one time. In Figure 9-7, if User A had locked both the paper and the pencil records at the beginning, deadlock would not occur. A second way to prevent deadlock is to require all application programs to lock resources in the same order.

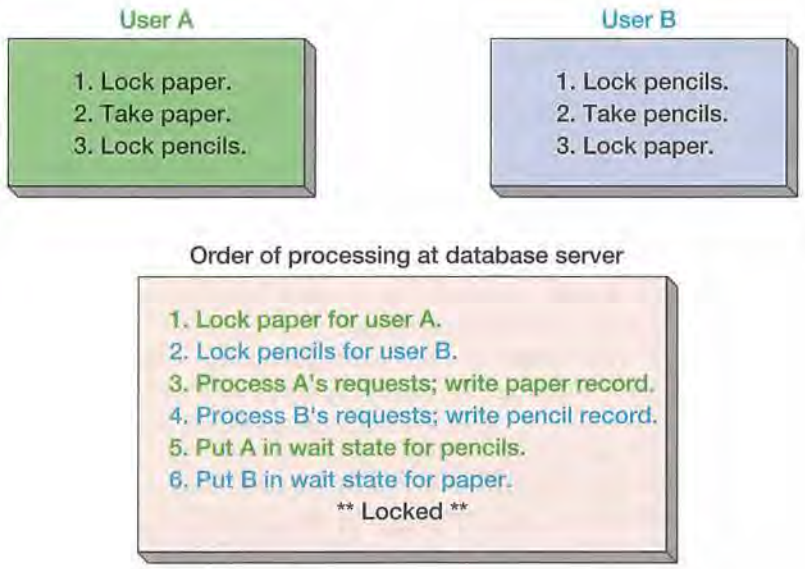


Figure 9-7
Deadlock

Amazon Ex. 1008.0356

> **BY THE WAY** Even if all the applications do not lock resources in the same order, deadlock will be prevented for those that do. Sometimes this policy is implemented with an organizational programming standard such as "Whenever processing rows from tables in a parent-child relationship, lock the parent row before the child rows." This policy will at least reduce the likelihood of deadlock and thus save the DBMS from having to recover from some deadlocked transactions.

Almost every DBMS has algorithms for breaking deadlock, when it does occur. First, the DBMS must detect that it has occurred. Then, the typical solution is to cancel one of the transactions and remove its changes from the database. You will see variants of this with Oracle, SQL Server, and MySQL in the next three chapters.

Optimistic Versus Pessimistic Locking

Locks can be invoked in two basic styles. With **optimistic locking**, the assumption is made that no conflict will occur. Data are read, the transaction is processed, updates are issued, and then a check is made to see if conflict occurred. If not, the transaction is finished. If conflict did occur, the transaction is repeated until it processes with no conflict. With **pessimistic locking**, the assumption is made that conflict will occur. Locks are issued, the transaction is processed, and then the locks are freed.

Figures 9-8 and 9-9 show examples of each style for a transaction that is reducing the quantity of the pencil row in PRODUCT by five. Figure 9-8 shows optimistic locking. First, the data are read and the current value of Quantity of pencils is saved in the variable OldQuantity. The transaction is then processed, and assuming that all is OK, a lock is obtained on PRODUCT. (In fact, the lock might be only for the pencil row or it might be at a larger level of granularity, but the principle is the same.) After obtaining the lock, an SQL statement is issued to update the pencil row with a WHERE condition that the current value of Quantity equals OldQuantity. If no other transaction has changed the Quantity of the pencil row, then this UPDATE will be successful. If another transaction has changed the Quantity of the pencil row, the UPDATE will fail. In either case, the lock is released. If the transaction failed, the process is repeated until the transaction finishes with no conflict.

Figure 9-8
Optimistic Locking

```
SELECT   PRODUCT.Name, PRODUCT.Quantity
FROM     PRODUCT
WHERE    PRODUCT.Name = 'Pencil'

Set NewQuantity = PRODUCT.Quantity - 5

{process transaction - take exception action if NewQuantity < 0, etc.

Assuming all is OK: }

LOCK PRODUCT

UPDATE   PRODUCT
SET      PRODUCT.Quantity = NewQuantity
WHERE    PRODUCT.Name = 'Pencil'
     AND PRODUCT.Quantity = OldQuantity

UNLOCK    PRODUCT

{check to see if update was successful;
if not, repeat transaction}
```

(a)

Amazon Ex. 1008.0357

```
LOCK      PRODUCT

SELECT    PRODUCT.Name, PRODUCT.Quantity
FROM      PRODUCT
WHERE     PRODUCT.Name = 'Pencil'

Set NewQuantity = PRODUCT.Quantity - 5

(process transaction - take exception action if NewQuantity < 0, etc.

Assuming all is OK: )

UPDATE    PRODUCT
SET       PRODUCT.Quantity = NewQuantity
WHERE     PRODUCT.Name = 'Pencil'

UNLOCK    PRODUCT

(no need to check if update was successful)
```

(b)

Figure 9-9
Pessimistic Locking

Figure 9-9 shows the logic for the same transaction using pessimistic locking. Here, a lock is obtained on PRODUCT before any work is begun. Then, values are read, the transaction is processed, the UPDATE occurs, and PRODUCT is unlocked.

The advantage of optimistic locking is that locks are held for much less time than with pessimistic locking, because locks are obtained only after the transaction has finished. If the transaction is complicated or if the client is slow (due to transmission delays, the client doing other work, or the user getting a cup of coffee or shutting down without exiting the browser), optimistic locking can dramatically improve throughput. This advantage will be especially true if the lock granularity is large—say, the entire PRODUCT table.

The disadvantage of optimistic locking is that if there is a lot of activity on the pencil row, the transaction may have to be repeated many times. Thus, transactions that involve a lot of activity on a given row (purchasing a popular stock, for example) are poorly suited for optimistic locking.

In general, the Internet is a wild and woolly place, and users are likely to take unexpected actions, such as abandoning transactions in the middle. So, unless Internet users have been prequalified (by enrolling in an online brokerage stock purchase plan, for example), optimistic locking is the better choice in that environment. On intranets, however, the decision is more difficult. Optimistic locking is probably still preferred unless some characteristic of the application causes substantial activity on particular rows or if application requirements make reprocessing transactions particularly undesirable.

Declaring Lock Characteristics

As you can see, concurrency control is a complicated subject; determining the level, type, and placement of the lock is difficult. Sometimes, too, the optimum locking strategy depends on which transactions are active and what they are doing. For these and other reasons, database application programs do not generally explicitly issue locks as shown in Figures 9-8 and 9-9. Instead, they mark transaction boundaries and then declare the type of locking behavior they want the DBMS to use. In this way, the DBMS can place and remove locks and even change the level and type of locks dynamically.

Figure 9-10 shows the pencil transaction with transaction boundaries marked with BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK TRANSACTION statements. These boundaries are the essential information that the DBMS needs to enforce the different locking strategies. If the developer now declares via a system parameter that he or she wants optimistic locking, the DBMS will implicitly set locks for that locking style. If, however, the developer declares pessimistic locking, the DBMS will set the locks differently.

Amazon Ex. 1008.0358

```
BEGIN TRANSACTION

SELECT     PRODUCT.Name, PRODUCT.Quantity
FROM       PRODUCT
WHERE      PRODUCT.Name = 'Pencil'

Set NewQuantity = PRODUCT.Quantity - 5

{process transaction - take exception action if NewQuantity < 0, etc.}

UPDATE     PRODUCT
SET        PRODUCT.Quantity = NewQuantity
WHERE      PRODUCT.Name = 'Pencil'

{continue processing transaction} . . .


IF {transaction has completed normally} THEN

     COMMIT TRANSACTION

ELSE

     ROLLBACK TRANSACTION

END IF

Continue processing other actions not part of this transaction . . .
```

Figure 9-10

Marking Transaction Boundaries

Consistent Transactions

Sometimes, you will see the acronym ACID applied to transactions. An **ACID transaction** is one that is ***a***tomic, ***c***onsistent, ***i***solated, and ***d***urable. Atomic and durable are easy to define. As you just learned, an **atomic** transaction is one in which either all of the database actions occur or none of them do. A **durable** transaction is one in which all committed changes are permanent. Once a durable change is committed, the DBMS takes responsibility for ensuring that the change will survive system failures.

The terms **consistent** and **isolated** are not as definitive as the terms atomic and durable. Consider the following SQL UPDATE statement:

```
UPDATE      CUSTOMER
SET         AreaCode = '425'
WHERE       ZipCode = '98050';
```

Suppose that there are 500,000 rows in the CUSTOMER table, and that 500 of them have ZipCode equal to '98050'. It will take some time for the DBMS to find those 500 rows. During that time, other transactions may attempt to update the AreaCode or ZipCode fields of CUSTOMER. If the SQL statement is consistent, such update requests will be disallowed. Hence, the update will apply to the set of rows as they existed at the time the SQL statement started. Such consistency is called **statement-level consistency**.

Now, consider a transaction that contains two SQL UPDATE statements:

```
BEGIN TRANSACTION
    UPDATE      CUSTOMER
    SET         AreaCode = '425'
    WHERE       ZipCode = '98050';
    {other transaction work}
```

Amazon Ex. 1008.0359

```
    UPDATE      CUSTOMER
    SET         Discount = 0.05
    WHERE       AreaCode = '425';
    (other transaction work)
COMMIT TRANSACTION
```

In this context, what does *consistent* mean? Statement-level consistency means that each statement independently processes rows consistently, but that changes from other users to these rows might be allowed during the interval between the two SQL statements. **Transaction-level consistency** means that all rows impacted by either of the SQL statements are protected from changes during the entire transaction.

Observe that transaction-level consistency is so strong that, for some implementations of it, a transaction will not see its own changes. In this example, the second SQL statement may not see rows changed by the first SQL statement.

Thus, when you hear the term *consistent*, look further to determine which type of consistency is meant. Be aware as well of the potential trap of transaction-level consistency.

Transaction Isolation Level

The term *isolated* has several different meanings. To understand those meanings, we need first to define several new terms.

A **dirty read** occurs when a transaction reads a row that has been changed but for which the change has not yet been committed to the database. The danger of a dirty read is that the uncommitted change can be rolled back. If so, the transaction that made the dirty read will be processing incorrect data.

Nonrepeatable reads occur when a transaction rereads data it has previously read and finds modifications or deletions caused by a committed transaction. Finally, **phantom reads** occur when a transaction rereads data and finds new rows that were inserted by a committed transaction since the prior read.

The 1992 SQL standard defines four **isolation levels** that specify which of these problems are allowed to occur. Using these levels, the application programmer can declare the type of isolation level he or she wants, and the DBMS will create and manage locks to achieve that level of isolation.

As shown in Figure 9-11, the read uncommitted isolation level allows dirty reads, nonrepeatable reads, and phantom reads to occur. With read-committed isolation, dirty reads are disallowed. The repeatable-read isolation level disallows both dirty reads and nonrepeatable reads. The serializable isolation level will not allow any of these three problems to occur.

Generally, the more restrictive the level, the less the throughput, though much depends on the workload and how the application programs are written. Moreover, not all DBMS products support all of these levels. You will learn how SQL Server, Oracle and MySQL support isolation levels in the next three chapters.

Figure 9-11

Summary of Transaction Isolation Levels

		Isolation Level			
		Read Uncommitted	Read Committed	Repeatable Read	Serializable
Problem Type	Dirty Read	Possible	Not Possible	Not Possible	Not Possible
	Nonrepeatable Read	Possible	Possible	Not Possible	Not Possible
	Phantom Read	Possible	Possible	Possible	Not Possible

Amazon Ex. 1008.0360

Cursor Type

A **cursor** is a pointer into a set of rows. Cursors are usually defined using SQL SELECT statements. For example, the following statement defines a cursor named TransCursor that operates over the set of rows indicated by the following SELECT statement:

```
DECLARE CURSOR TransCursor AS
    SELECT     *
    FROM       TRANS
    WHERE      PurchasePrice > '10000';
```

As explained in Chapter 7, after an application program opens a cursor, it can place the cursor somewhere in the result set. Most commonly, the cursor is placed on the first or last row, but other possibilities exist.

A transaction can open several cursors—either sequentially or simultaneously. Additionally, two or more cursors may be open on the same table; either directly on the table or through an SQL view on that table. Because cursors require considerable memory, having many cursors open at the same time for, say, a thousand concurrent transactions, will consume considerable memory. One way to reduce cursor burden is to define reduced-capability cursors and use them when a full-capability cursor is not needed.

Figure 9-12 lists four cursor types used in the Windows environment (cursor types for other systems are similar). The simplest cursor is forward only. With it, the application can only move forward through the records. Changes made by other cursors in this transaction and by other transactions will be visible only if they occur to rows ahead of the cursor.

The next three types of cursors are called **scrollable cursors**, because the application can scroll forward and backward through the records. A **static cursor** takes a snapshot of a relation and processes that snapshot. Changes made using this cursor are visible; changes from other sources are not visible.

A **keyset cursor** combines some of the features of static cursors with some of the features of dynamic cursors. When the cursor is opened, a primary key value is saved for each row. When the application positions the cursor on a row, the DBMS uses the key value to read the current value of the row. Inserts of new rows by other cursors (in this transaction or in other transactions) are not visible. If the application issues an update on a row that has been deleted by a different cursor, the DBMS creates a new row with the old key value and places the updated values in the new row (assuming that all required fields are present). Unless the isolation level of the transaction is a dirty read, only committed updates and deletions are visible to the cursor.

A **dynamic cursor** is a fully featured cursor. All inserts, updates, deletions, and changes in row order are visible to a dynamic cursor. As with keyset cursors, unless the isolation level of the transaction is a dirty read, only committed changes are visible.

The amount of overhead and processing required to support a cursor is different for each type of cursor. In general, the cost goes up as we move down the cursor types shown in Figure 9-12. To improve DBMS performance, the application developer should create cursors that are just powerful enough to do the job. It is also very important to understand how a particular DBMS implements cursors and whether cursors are located on the server or on the client. In some cases, it might be better to place a dynamic cursor on the client than to have a static cursor on the server. No general rule can be stated because performance depends on the implementation used by the DBMS product and the application requirements.

A word of caution: If you do not specify the isolation level of a transaction or do not specify the type of cursors you open, the DBMS will use a default level and default types. These defaults may be perfect for your application, but they also may be terrible. Thus, even though these issues can be ignored, their consequences cannot be avoided. You must learn the capabilities of your DBMS product.

Amazon Ex. 1008.0361

CursorType	Description	Comments
Forward only	Application can only move forward through the recordset.	Changes made by other cursors in this transaction or in other transactions will be visible only if they occur on rows ahead of the cursor.
Static	Application sees the data as they were at the time the cursor was opened.	Changes made by this cursor are visible. Changes from other sources are not visible. Backward and forward scrolling allowed.
Keyset	When the cursor is opened, a primary key value is saved for each row in the recordset. When the application accesses a row, the key is used to fetch the current values for the row.	Updates from any source are visible. Inserts from sources outside this cursor are not visible (there is no key for them in the keyset). Inserts from this cursor appear at the bottom of the recordset. Deletions from any source are visible. Changes in row order are not visible. If the isolation level is dirty read, then committed updates and deletions are visible; otherwise only committed updates and deletions are visible.
Dynamic	Changes of any type and from any source are visible.	All inserts, updates, deletions, and changes in recordset order are visible. If the isolation level is dirty read, then uncommitted changes are visible. Otherwise, only committed changes are visible.

Figure 9-12
Summary of Cursor Types

Database Security

The goal of database security is to ensure that only authorized users can perform authorized activities at authorized times. This goal is difficult to achieve, and to make any progress at all the database development team must determine the processing rights and responsibilities of all users during the project's requirements specification phase. These security requirements can then be enforced using the security features of the DBMS and additions to those features written into the application programs.

Processing Rights and Responsibilities

Consider, for example, the needs of View Ridge Gallery. The View Ridge database has three types of users: sales personnel, management personnel, and system administrators. View Ridge designed processing rights for each as follows: Sales personnel are allowed to enter new customer and transaction data, to change customer data, and to query any of the data. They are not allowed to enter new artist or work data. They are never allowed to delete data.

Management personnel are allowed all of the permissions of sales personnel, plus they are allowed to enter new artist and work data and to modify transaction data. Even though management personnel have the authority to delete data, they are not given that permission in this application. This restriction is made to prevent the possibility of accidental data loss.

Amazon Ex. 1008.0362

	CUSTOMER	TRANSACTION	WORK	ARTIST
Sales personnel	Insert, change, query	Insert, query	Query	Query
Management personnel	Insert, change, query	Insert, change, query	Insert, change, query	Insert, change, query
System administrator	Grant rights, modify structure	Grant rights, modify structure	Grant rights, modify structure	Grant rights, modify structure

Figure 9-13
Processing Rights at View Ridge Gallery

The system administrator can grant processing rights to other users; and he or she can change the structure of the database elements such as tables, indexes, stored procedures, and the like. The system administrator is not given rights to process the data. Figure 9-13 summarizes these processing rights.

BY THE WAY You may be wondering what good it does to say that the system administrator cannot process the data when that person has the ability to grant processing rights. He or she can just grant the right to change data to him- or herself. Although this is true, the granting of those rights will leave an audit trail in the database log. Clearly, this limitation is not foolproof, but it is better than just allowing the system administrator (or DBA) full access to all rights in the database.

The permissions in this table are not given to particular people, but rather are given to groups of people. Sometimes these groups are termed **roles**, because they describe people acting in a particular capacity. The term **user groups** is also used. Assigning permission to roles (or user groups) is typical, but not required. It would be possible to say, for example, that the user identified as "Benjamin Franklin" has certain processing rights. Note, too, that when roles are used, it is necessary to have a way to allocate users to roles. When "Mary Smith" signs on to the computer, there must be some way to determine which role or roles she has. We will discuss this further in the next section.

In this discussion, we have used the phrase **processing rights and responsibilities**. As this phrase implies, responsibilities go with processing rights. If, for example, the manager modifies transaction data, the manager has the responsibility to ensure that these modifications do not adversely impact the gallery's operation, accounting, and so forth.

Processing responsibilities cannot be enforced by the DBMS or by the database applications. Instead, they are encoded in manual procedures and explained to users during systems training. These are topics for a systems development text, and we will not consider them further here—except to reiterate that *responsibilities* go with *rights*. Such responsibilities must be documented and enforced.

According to Figure 9-1, the DBA has the task of managing processing rights and responsibilities. As this implies, these rights and responsibilities will change over time. As the database is used and as changes are made to the applications and to the structure of the DBMS, the need for new or different rights and responsibilities will arise. The DBA is a focal point for the discussion of such changes and for their implementation.

Once processing rights have been defined, they can be implemented at many levels: operating system, network, Web server, DBMS, and application. In the next two sections, we will consider DBMS and application implementation. The other levels are beyond the scope of this text.

Amazon Ex. 1008.0363

DBMS Security

The terminology, features, and functions of DBMS security depend on the DBMS product in use. Basically, all such products provide facilities that limit certain actions on certain objects to certain users. A general model of DBMS security is shown in Figure 9-14.

A USER can be assigned to one or more ROLEs (or USER GROUPs), and a ROLE can have one or more USERs. An OBJECT is an element of a database, such as a table, view, or stored procedure. PERMISSION is an association entity among USER, ROLE, and OBJECT. Hence, the relationships from USER to PERMISSION, ROLE to PERMISSION, and OBJECT to PERMISSION are all 1:N, M-O.

When a user signs on to the database, the DBMS limits the user's actions to the permissions defined for that user and to the permissions for roles to which that user has been assigned. Determining whether someone actually is who they claim to be is a difficult task, in general. All commercial DBMS products use some version of user name and password verification, even though such security is readily circumvented if users are careless with their identities.

Users can enter their name and password, or, in some applications, the name and password is entered on the user's behalf. For example, the Windows user name and password can be passed directly to the DBMS. In other cases, an application program provides the name and password. Internet applications usually define a group such as "Unknown Public" and assign anonymous users to that group when they sign on. In this way, companies, such as Dell, need not enter every potential customer into their security system by name and password.

SQL Server, Oracle, and MySQL security systems are variations of the model in Figure 9-14. You will learn about them in Chapters 10, 10A, and 10B, respectively.

DBMS Security Guidelines

Guidelines for improving security in database systems are listed in Figure 9-15. First, the DBMS must always be run behind a firewall. However, the DBA should plan security with the assumption that the firewall has been breached. The DBMS, the database, and all applications should be secure even if the firewall fails.

DBMS vendors, including IBM, Oracle, Microsoft, and Sun Microsystems are constantly adding product features to improve security and reduce vulnerability. Consequently, organizations using DBMS products should continually check the vendors' Web sites for service packs and fixes; any service packs or fixes that involve security features, functions, and processing should be installed as soon as possible.

The installation of new service packs and fixes is not quite as simple as described here. The installation of a service pack or fix can break some applications, particularly some licensed software that requires specific service packs and fixes to be installed (or not installed). It may be necessary to delay installation of DBMS service packs until vendors of licensed software have upgraded their products to work with the new versions. Sometimes just the possibility that a licensed application *might* fail after a DBMS service pack or fix is applied is sufficient reason to delay the fix. But, the DBMS is still vulnerable during this period. Pick your regret!

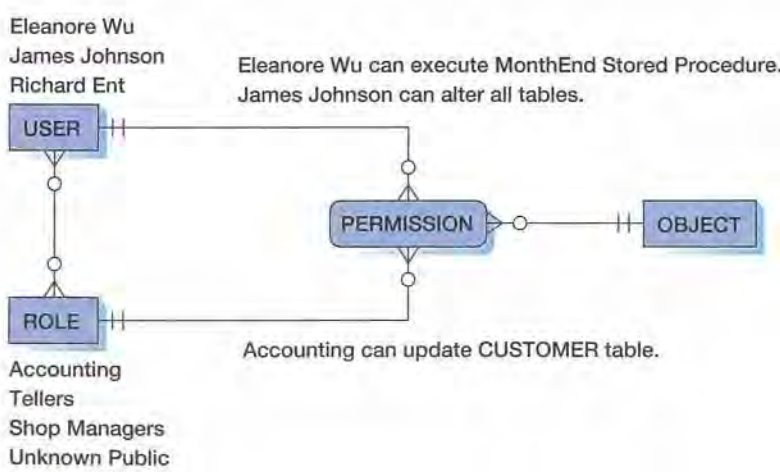


Figure 9-14
A Model of DBMS Security

Amazon Ex. 1008.0364

• Run DBMS behind a firewall, but plan as though the firewall has been breached
• Apply the latest operating system and DBMS service packs and fixes
• Use the least functionality possible   • Support the fewest network protocols possible   • Delete unnecessary or unused system stored procedures   • Disable default logins and guest users, if possible   • Unless required, never allow users to log on to the DBMS interactively
• Protect the computer that runs the DBMS   • No user allowed to work at the computer that runs the DBMS   • DBMS computer physically secured behind locked doors   • Visits to the room containing the DBMS computer should be recorded in a log
• Manage accounts and passwords   • Use a low privilege user account for the DBMS service   • Protect database accounts with strong passwords   • Monitor failed login attempts   • Frequently check group and role memberships   • Audit accounts with null passwords   • Assign accounts the lowest privileges possible   • Limit DBA account privileges
• Planning   • Develop a security plan for preventing and detecting security problems   • Create procedures for security emergencies and practice them

Figure 9-15
Summary of DBMS Security Guidelines

Additionally, database features and functions that are not required by the applications should be removed or disabled from the DBMS. For example, if TCP/IP is used to connect to the DBMS, other communications protocols should be removed. This action reduces the pathways by which unauthorized activity can reach the DBMS. Further, all DBMS products are installed with system-stored procedures that provide services such as starting a command file, modifying the system registry, initiating e-mail, and the like. Any of these stored procedures that are not needed should be removed. If all users are known to the DBMS, default logins and guest user accounts should be removed as well. Finally, unless otherwise required, users should never be allowed to log on to the DBMS in interactive mode. They should always access the database via an application.

In addition, the computer(s) that runs the DBMS must be protected. No one other than authorized DBA personnel should be allowed to work at the keyboard of the computer that runs the DBMS. The computer running the DBMS should be physically secured behind locked doors, and access to the facility housing the computer should be controlled. Visits to the DBMS computer room should be recorded in a log.

Accounts and passwords should be assigned carefully and continually managed. The DBMS itself should run on an account that has the lowest possible operating system privileges. In that way, if an intruder were to gain control of the DBMS, the intruder would have limited authority on that local computer or network. Additionally, all accounts within the DBMS should be protected by **strong passwords**. Such passwords have at least 15 characters and contain upper- and lowercase letters; numbers; special characters, such as +, @, #, ***; and unprintable key combinations (certain Alt + key combinations).

The DBA should frequently check the accounts that have been assigned to groups and roles to ensure that all accounts and roles are known, authorized, and have the correct permissions. Further, the DBA should audit accounts with null passwords. The users of such accounts should be required to protect those accounts with strong passwords. Also, as a general rule, accounts should be granted the lowest privileges possible.

As stated, the privileges for the DBA should normally not include the right to process the users' data. If the DBA grants him- or herself that privilege, the unauthorized grant operation will be visible in the database log.

Amazon Ex. 1008.0365

In the spring of 2003, the Slammer worm invaded thousands of sites running SQL Server. Microsoft had previously released a patch to SQL Server that prevented this attack. Sites that had installed the patch had no problems. The moral: Install security patches to your DBMS as promptly as possible. Create a procedure for regularly checking for such patches.

Finally, the DBA should participate in security planning. Procedures both for preventing and detecting security problems should be developed. Furthermore, procedures should be developed for actions to be taken in case of a security breach. Such procedures should be practiced. The importance of security in information systems has increased dramatically in recent years. DBA personnel should regularly search for security information on the Web in general and at the DBMS vendor's Web site.

Application Security

Although DBMS products such as Oracle, SQL Server, and MySQL do provide substantial database security capabilities, those capabilities are generic. If the application requires specific security measures such as "No user can view a row of a table or of a join of a table that has an employee name other than his or her own," the DBMS facilities will not be adequate. In these cases, the security system must be augmented by features in database applications.

For example, as you will learn in Chapter 11, application security in Internet applications is often provided on the Web server. Executing application security on this server means that sensitive security data need not be transmitted over the network.

To understand this better, suppose that an application is written so that when users click a particular button on a browser page, the following query is sent to the Web server and then to the DBMS:

```
SELECT      *
FROM        EMPLOYEE;
```

This statement will, of course, return all EMPLOYEE rows. If the application security policy only permits employees to access their own data, then a Web server could add the following WHERE clause to this query:

```
SELECT      *
FROM        EMPLOYEE
WHERE       EMPLOYEE.Name = '<% = SESSION(("EmployeeName")")%>';
```

An expression like this one will cause the Web server to fill the employee's name into the WHERE clause. For a user signed in under the name 'Benjamin Franklin', the statement that results from this expression is:

```
SELECT      *
FROM        EMPLOYEE
WHERE       EMPLOYEE.Name = 'Benjamin Franklin';
```

Because the name is inserted by a program on the Web server, the browser user does not know that it is occurring, and cannot interfere with it even if he or she did.

Such security processing can be done as shown here on a Web server, but it also can be done within the application programs themselves or written as stored procedures or triggers to be executed by the DBMS at the appropriate times.

This idea can be extended by storing additional data in a security database that is accessed by the Web server or by stored procedures and triggers. That security database could contain, for example, the identities of users paired with additional values of WHERE

Amazon Ex. 1008.0366

clauses. For example, suppose that the users in the personnel department can access more than just their own data. The predicates for appropriate WHERE clauses could be stored in the security database, read by the application program, and appended to SQL SELECT statements as necessary.

Many other possibilities exist for extending DBMS security with application processing. In general, however, you should use the DBMS security features first. Only if they are inadequate for the requirements should you add to them with application code. The closer the security enforcement is to the data, the less chance there is for infiltration. Also, using the DBMS security features is faster, cheaper, and probably results in higher-quality results than developing your own.

The SQL Injection Attack

Whenever data from the user are used to modify an SQL statement, an **SQL injection attack** is possible. For example, in the prior section, if the value of EmployeeName used in the SELECT statement is not obtained via a secure means, such as from the operating system rather from a Web form, there is the chance that the user can inject SQL into the statement.

For example, suppose that users are asked to enter their names into a Web form textbox. Suppose that a user enters the value *'Benjamin Franklin' OR TRUE* for his or her name. The SQL statement generated by the application will then be the following:

```
SELECT      *
FROM        EMPLOYEE
WHERE       EMPLOYEE.Name = 'Benjamin Franklin' OR TRUE;
```

Of course, the value TRUE is true for every row, so every row of the EMPLOYEE table will be returned! Thus, any time user input is used to modify an SQL statement, that input must be carefully edited to ensure that only valid input has been received and that no additional SQL syntax has been entered.

Database Backup and Recovery

Computer systems fail. Hardware breaks. Programs have bugs. Human procedures contain errors, and people make mistakes. All of these failures can and do occur in database applications. Because a database is shared by many people and because it often is a key element of an organization's operations, it is important to recover it as soon as possible.

Several problems must be addressed. First, from a business standpoint, business functions must continue. During the failure, customer orders, financial transactions, and packing lists must be completed somehow, even manually. Later, when the database application is operational again, the data from those activities must be entered into the database. Second, computer operations personnel must restore the system to a usable state as quickly as possible and as close as possible to what it was when the system crashed. Third, users must know what to do when the system becomes available again. Some work may need to be reentered, and users must know how far back they need to go.

When failures occur, it is impossible simply to fix the problem and resume processing. Even if no data are lost during a failure (which assumes that all types of memory are nonvolatile—an unrealistic assumption), the timing and scheduling of computer processing are too complex to be accurately re-created. Enormous amounts of overhead data and processing would be required for the operating system to be able to restart processing precisely where it was interrupted. It is simply not possible to roll back the clock and put all of the electrons in the same configuration they were in at the time of the failure. Two other approaches are possible: **recovery via reprocessing** and **recovery via rollback/rollforward**.

Amazon Ex. 1008.0367

Recovery via Reprocessing

Because processing cannot be resumed at a precise point, the next best alternative is to go back to a known point and reprocess the workload from there. The simplest form of this type of recovery is to periodically make a copy of the database (called a **database save**) and to keep a record of all transactions that have been processed since the save. Then, when there is a failure, the operations staff can restore the database from the save and then reprocess all the transactions. Unfortunately, this simple strategy is normally not feasible. First, reprocessing transactions takes the same amount of time as processing them in the first place did. If the computer is heavily scheduled, the system may never catch up.

Second, when transactions are processed concurrently, events are asynchronous. Slight variations in human activity, such as a user reading an e-mail before responding to an application prompt, can change the order of the execution of concurrent transactions. Therefore, whereas Customer A got the last seat on a flight during the original processing, Customer B may get the last seat during reprocessing. For these reasons, reprocessing is normally not a viable form of recovery from failure in concurrent processing systems.

Recovery via Rollback/Rollforward

A second approach is to periodically make a copy of the database (the database save) and to keep a log of the changes made by transactions against the database since the save. Then, when there is a failure, one of two methods can be used. Using the first method, called **rollforward**, the database is restored using the saved data, and all valid transactions since the save are reapplied. (We are not reprocessing the transactions because the application programs are not involved in the rollforward. Instead, the processed changes, as recorded in the log, are reapplied.)

The second method is **rollback**. With this method, we undo changes made by erroneous or partially processed transactions by undoing the changes they have made in the database. Then, the valid transactions that were in process at the time of the failure are restarted.

Both of these methods require that a **log** of the transaction results be kept. This log contains records of the data changes in chronological order. Transactions must be written to the log before they are applied to the database. That way, if the system crashes between the time a transaction is logged and the time it is applied, at worst there is a record of an unapplied transaction. If, however, the transactions were to be applied before they were logged, it would be possible (as well as undesirable) to change the database but have no record of the change. If this happened, an unwary user might reenter an already completed transaction. In the event of a failure, the log is used both to undo and to redo transactions, as shown in Figure 9-16.

To undo a transaction, the log must contain a copy of every database record (or page) before it was changed. Such records are called **before images**. A transaction is undone by applying before images of all of its changes to the database.

To redo a transaction, the log must contain a copy of every database record (or page) after it was changed. These records are called **after images**. A transaction is redone by applying after images of all of its changes to the database. Possible data items in a transaction log are shown in Figure 9-17.

In this example log, each transaction has a unique name for identification purposes. Furthermore, all of the images for a given transaction are linked together with pointers. One pointer points to the previous change made by this transaction (the reverse pointer), and the other points to the next change made by this transaction (the forward pointer). A zero in the pointer field means that this is the end of the list. The DBMS recovery subsystem uses these pointers to locate all of the records for a particular transaction. Figure 9-17 shows an example of the linking of log records.

Other data items in the log are the time of the action; the type of operation (START marks the beginning of a transaction and COMMIT terminates a transaction, releasing all locks that were in place); the object acted on, such as record type and identifier; and, finally, the before images and the after images.

Amazon Ex. 1008.0368

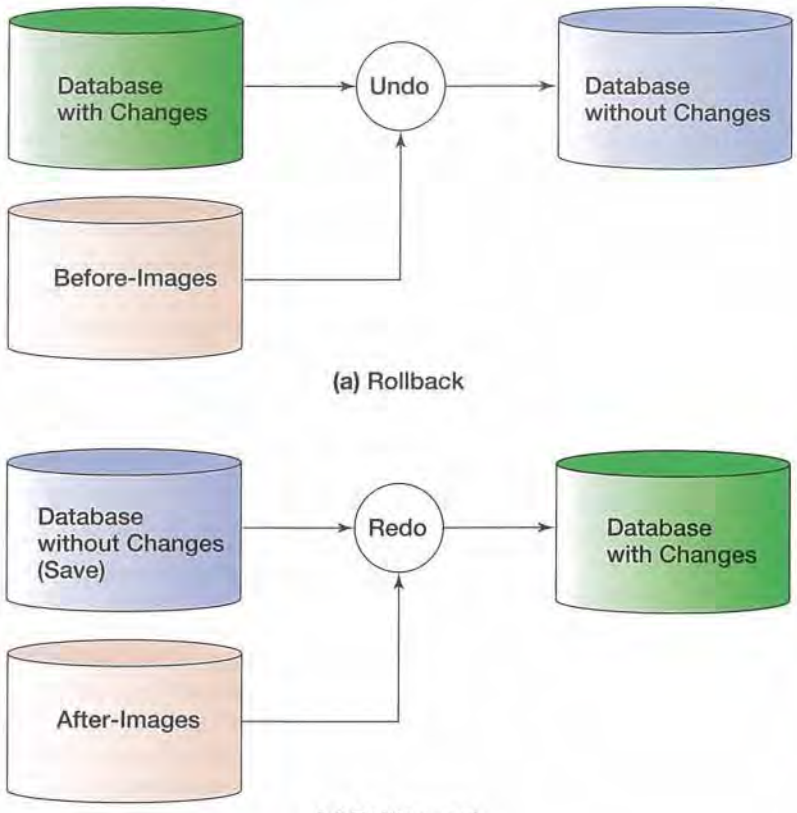


Figure 9-16
Undo and Redo Transactions

Relative Record Number	Transaction ID	Reverse Pointer	Forward Pointer	Time	Type of Operation	Object	Before-Image	After-Image
1	OT1	0	2	11:42	START			
2	OT1	1	4	11:43	MODIFY	CUST 100	(old value)	(new value)
3	OT2	0	8	11:46	START			
4	OT1	2	5	11:47	MODIFY	SP AA	(old value)	(new value)
5	OT1	4	7	11:47	INSERT	ORDER 11		(value)
6	CT1	0	9	11:48	START			
7	OT1	5	0	11:49	COMMIT			
8	OT2	3	0	11:50	COMMIT			
9	CT1	6	10	11:51	MODIFY	SP BB	(old value)	(new value)
10	CT1	9	0	11:51	COMMIT			

Figure 9-17
Example Transaction Log

Given a log with before images and after images, the undo and redo actions are straightforward. To undo the transaction in Figure 9-18, the recovery processor simply replaces each changed record with its before image.

When all of the before images have been restored, the transaction is undone. To redo a transaction, the recovery processor starts with the version of the database at the time the transaction started and applies all of the after images. As stated, this action assumes that an earlier version of the database is available from a database save.

Amazon Ex. 1008.0369

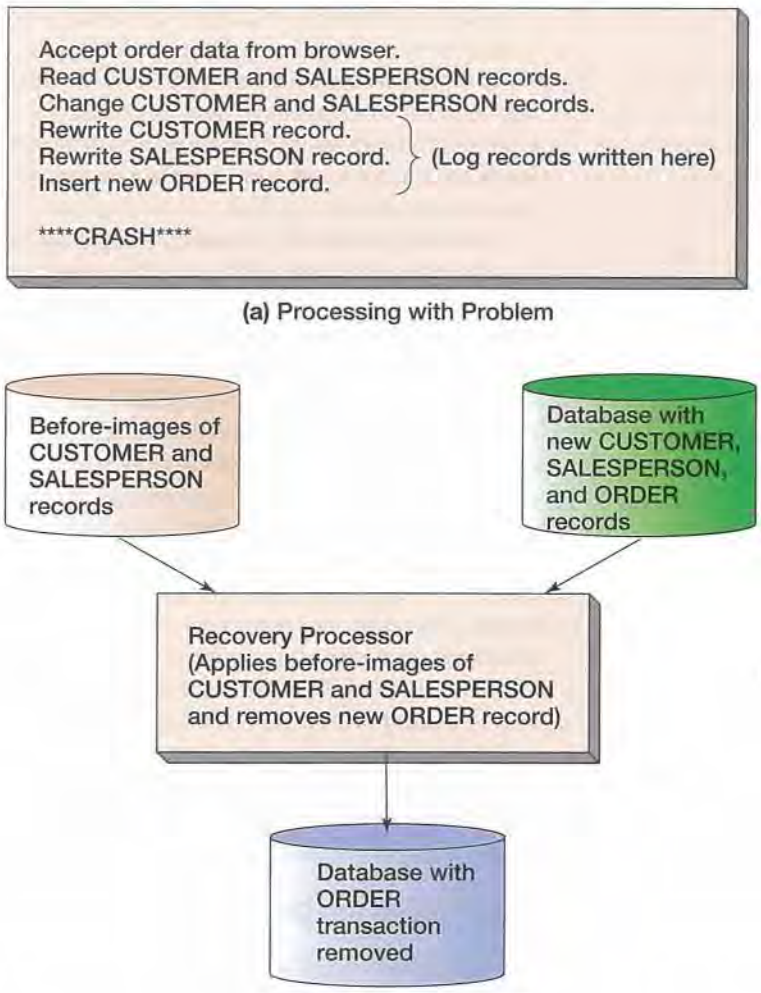


Figure 9-18
Recovery Example

Restoring a database to its most recent save and reapplying all transactions may require considerable processing. To reduce the delay, DBMS products sometimes use checkpoints. A **checkpoint** is a point of synchronization between the database and the transaction log. To perform a checkpoint, the DBMS refuses new requests, finishes processing outstanding requests, and writes its buffers to disk. The DBMS then waits until the operating system notifies it that all outstanding write requests to the database and to the log have been successfully completed. At this point, the log and the database are synchronized. A checkpoint record is then written to the log. Later, the database can be recovered from the checkpoint and only after images for transactions that started after the checkpoint need be applied.

Checkpoints are inexpensive operations, and it is feasible to take three or four (or more) per hour. In this way, no more than 15 or 20 minutes of processing need to be recovered. Most DBMS products perform automatic checkpoints, making human intervention unnecessary.

You will see specific examples of backup and recovery techniques for SQL Server, Oracle, and MySQL in the next three chapters. For now, you only need to understand the basic ideas and to realize that it is the responsibility of the DBA to ensure that adequate backup and recovery plans have been developed and that database saves and logs are generated as required.

Managing the DBMS

In addition to managing data activity and the database structure, the DBA must manage the DBMS itself. The DBA should compile and analyze statistics concerning the system's performance and identify potential problem areas. Keep in mind that the database is serving many user groups. The DBA needs to investigate all complaints about the system's response time, accuracy, ease of use, and so forth. If changes are needed, the DBA must plan and implement them.

Amazon Ex. 1008.0370

The DBA must periodically monitor the users' activity on the database. DBMS products include features that collect and report statistics. For example, some of these reports may indicate which users have been active, which files—and perhaps which data items—have been used, and which access methods have been employed. Error rates and types also can be captured and reported. The DBA analyzes these data to determine whether a change to the database design is needed to improve performance or to ease the users' tasks. If change is necessary, the DBA will ensure that it is accomplished.

The DBA should analyze run-time statistics on database activity and performance. When a performance problem is identified (by either a report or a user's complaint), the DBA must determine whether a modification of the database structure or system is appropriate. Examples of possible structural modifications are establishing new keys, purging data, deleting keys, and establishing new relationships among objects.

When the vendor of the DBMS being used announces new product features, the DBA must consider them in light of the overall needs of the user community. If the DBA decides to incorporate the new DBMS features, the developers must be notified and trained in their use. Accordingly, the DBA must manage and control changes in the DBMS as well as in the database structure.

Other changes in the system for which the DBA is responsible vary widely, depending on the DBMS product as well as on other software and hardware in use. For example, changes in other software (such as the operating system or the Web server) may mean that some DBMS features, functions, or parameters must be changed. The DBA must therefore also tune the DBMS product with other software in use.

The DBMS options (such as transaction isolation levels) are initially chosen when little is known about how the system will perform in the particular user environment. Consequently, operational experience and performance analysis over a period of time may reveal that changes are necessary. Even if the performance seems acceptable, the DBA may want to alter the options and observe the effect on performance. This process is referred to as *tuning*, or *optimizing*, the system. Figure 9-19 summarizes the DBA's responsibilities for managing the DBMS product.

Maintaining the Data Repository

Consider a large and active Internet database application, such as those used by e-commerce companies—for instance, an application that is used by a company that sells clothing over the Internet. Such a system may involve data from several different databases, dozens of different Web pages, and hundreds, or even thousands, of users.

Suppose that the company using this application decides to expand its product line to include the sale of sporting goods. Senior management of this company might ask the DBA to develop an estimate of the time and other resources required to modify the database application to support this new product line.

To respond to this request, the DBA needs accurate metadata about the database, about the database applications and application components, about the users and their rights and privileges, and about other system elements. The database does carry some of this metadata in system tables, but this metadata is inadequate to answer the questions posed by senior management. The DBA needs additional metadata about COM and ActiveX objects, script procedures and functions, Active Server Pages (ASPs), style sheets, document type definitions, and the like. Furthermore, although

Figure 9-19
Summary of the DBA's Responsibilities for Managing the DBMS

• Generate database application performance reports
• Investigate user performance complaints
• Assess need for changes in database structure or application design
• Modify database structure
• Evaluate and implement new DBMS features
• Tune the DBMS

Amazon Ex. 1008.0371

DBMS security mechanisms document users, groups, and privileges, they do so in a highly structured, and often inconvenient, form.

For all of these reasons, many organizations develop and maintain **data repositories**, which are collections of metadata about databases, database applications, Web pages, users, and other application components. The repository may be virtual in that it is composed of metadata from many different sources: the DBMS, version-control software, code libraries, Web page generation and editing tools, and so forth. Or, the data repository may be an integrated product from a CASE tool vendor or from a company such as Microsoft or Oracle.

Either way, the time for the DBA to think about constructing such a facility is long before senior management asks questions. In fact, the repository should be constructed as the system is developed and should be considered an important part of the system deliverables. If such a facility is not constructed, the DBA will always be playing catch-up—trying to maintain the existing applications, adapting them to new needs, and somehow gathering together the metadata to form a repository.

The best repositories are **active repositories**—they are part of the systems development process in that metadata is created automatically as the system components are created. Less desirable, but still effective, are **passive repositories**, which are filled only when someone takes the time to generate the needed metadata and place it in the repository.

The Internet has created enormous opportunities for businesses to expand their customer bases and increase their sales and profitability. The databases and database applications that support these companies are an essential element of that success. Unfortunately, the growth of some organizations will be stymied by their inability to grow their applications or adapt them to changing needs. Often, building a new system is easier than adapting an existing one. Building a new system that integrates with an old one while it replaces that old one can be very difficult.

Summary

Multiuser databases pose difficult problems for the organizations that create and use them, and most organizations have created an office of database administration to ensure that such problems are solved. In this text, the term *database administrator* refers to the person or office that is concerned with a single database. The term *data administrator* is used to describe a management function that is concerned with the organization's data policy and security. Major functions of the database administrator are listed in Figure 9-1.

The database administrator (DBA) participates in the initial development of database structures and in providing configuration control when requests for changes arise. Keeping accurate documentation of the structure and changes to it is an important DBA function.

The goal of concurrency control is to ensure that one user's work does not inappropriately influence another user's work. No single concurrency control technique is ideal for all circumstances. Trade-offs need to be made between the level of protection and throughput. A transaction, or logical unit of work (LUW), is a series of actions taken against the database that occurs as an atomic unit; either all of them occur or none of them do. The activity of concurrent transactions is interleaved on the database server. In some cases, updates can be lost if concurrent transactions are not controlled. Another concurrency problem concerns inconsistent reads.

To avoid concurrency problems, database elements are locked. Implicit locks are placed by the DBMS; explicit locks are issued by the application program. The size of the locked resource is called lock granularity. An exclusive lock prohibits other users from reading the locked resource; a shared lock allows other users to read the locked resource, but they cannot update it. Two transactions that run concurrently and generate results that are consistent with the results that would have occurred if they had run separately are referred to as serializable transactions. Two-phased locking, in which locks are acquired in a growing phase and released in a shrinking phase, is one scheme for serializability. A special case of two-phase locking is to acquire locks throughout the transaction, but not to free any lock until the transaction is finished.

Deadlock, or the deadly embrace, occurs when two transactions are each waiting on a resource that the other transaction holds. Deadlock can be prevented by requiring transactions to acquire all locks at the same time. Once deadlock occurs, the only way to cure it is to abort one of the transactions (and back out of partially completed work). Optimistic locking assumes that no transaction conflict will occur and deals with the consequences if it does. Pessimistic locking assumes that conflict will occur and so prevents it ahead of time with locks. In general, optimistic locking is preferred for the Internet and for many intranet applications.

Most application programs do not explicitly declare locks. Instead, they mark transaction boundaries with BEGIN, COMMIT, and ROLLBACK transaction statements and

Amazon Ex. 1008.0372

declare the concurrent behavior they want. The DBMS then places locks for the application that will result in the desired behavior.

An ACID transaction is one that is atomic, consistent, isolated, and durable. *Durable* means that database changes are permanent. *Consistency* can mean either statement-level or transaction-level consistency. With transaction-level consistency, a transaction may not see its own changes. The 1992 SQL standard defines four transaction isolation levels: read uncommitted, read committed, repeatable read, and serializable. The characteristics of each are summarized in Figure 9-11.

A cursor is a pointer into a set of records. Four cursor types are prevalent: forward only, static, keyset, and dynamic. Developers should select isolation levels and cursor types that are appropriate for their application workload and for the DBMS product in use.

The goal of database security is to ensure that only authorized users can perform authorized activities at authorized times. To develop effective database security, the processing rights and responsibilities of all users must be determined.

DBMS products provide security facilities. Most involve the declaration of users, groups, objects to be protected, and permissions or privileges on those objects. Almost all DBMS products use some form of user name and password security. Security guidelines are listed in Figure 9-15. DBMS security can be augmented by application security.

In the event of system failure, the database must be restored to a usable state as soon as possible. Transactions in process at the time of the failure must be reapplied or restarted. Although in some cases recovery can be done by reprocessing, the use of logs and rollback and rollforward is almost always preferred. Checkpoints can be taken to reduce the amount of work that needs to be done after a failure.

In addition to these tasks, the DBA manages the DBMS product itself, measuring database application performance and assessing the need for changes in database structure or DBMS performance tuning. The DBA also ensures that new DBMS features are evaluated and used as appropriate. Finally, the DBA is responsible for maintaining the data repository.

Key Terms

ACID transaction
active repository
after image
atomic
before image
checkpoint
concurrent transaction
concurrent update problem
consistent
cursor
data administration
data repository
database administration
database administrator
database save
DBA
deadlock
deadly embrace
dirty read
durable
dynamic cursor
exclusive lock
explicit lock
growing phase
implicit lock
inconsistent read problem
isolated
isolation levels
keyset cursor
lock
lock granularity
log
logical unit of work (LUW)
lost update problem
nonrepeatable reads
optimistic locking
passive repository
pessimistic locking
phantom reads
processing rights and responsibilities
recovery via reprocessing
recovery via rollback/rollforward
resource locking
role
rollback
rollforward
scrollable cursor
serializable
shared lock
shrinking phase
SQL injection attack
statement-level consistency
static cursor
strong password
transaction
transaction-level consistency
two-phased locking
user group

Amazon Ex. 1008.0373

Review Questions

9.1 Briefly describe five difficult problems for organizations that create and use multiuser databases.

9.2 Explain the difference between a database administrator and a data administrator.

9.3 List seven important DBA tasks.

9.4 Summarize the DBA's responsibilities for managing database structure.

9.5 What is configuration control? Why is it necessary?

9.6 Explain the meaning of the word *inappropriately* in the phrase "one user's work does not inappropriately influence another user's work."

9.7 Explain the trade-off that exists in concurrency control.

9.8 Define an atomic transaction and explain why atomicity is important.

9.9 Explain the difference between concurrent transactions and simultaneous transactions. How many CPUs are required for simultaneous transactions?

9.10 Give an example, other than the one in this text, of the lost update problem.

9.11 Explain the difference between an explicit and an implicit lock.

9.12 What is lock granularity?

9.13 Explain the difference between an exclusive lock and a shared lock.

9.14 Explain two-phased locking.

9.15 How does releasing all locks at the end of the transaction relate to two-phase locking?

9.16 In general, how should the boundaries of a transaction be defined?

9.17 What is deadlock? How can it be avoided? How can it be resolved once it occurs?

9.18 Explain the difference between optimistic and pessimistic locking.

9.19 Explain the benefits of marking transaction boundaries, declaring lock characteristics, and letting the DBMS place locks.

9.20 Explain the use of BEGIN, COMMIT, and ROLLBACK TRANSACTION statements.

9.21 Explain the meaning of the expression ACID transaction.

9.22 Describe statement-level consistency.

9.23 Describe transaction-level consistency. What disadvantage can exist with it?

9.24 What is the purpose of transaction isolation levels?

9.25 Explain the read uncommitted isolation level. Give an example of its use.

9.26 Explain the read committed isolation level. Give an example of its use.

9.27 Explain the repeatable read isolation level. Give an example of its use.

9.28 Explain the serializable isolation level. Give an example of its use.

9.29 Explain the term *cursor*.

9.30 Explain why a transaction may have many cursors. Also, how is it possible that a transaction may have more than one cursor on a given table?

9.31 What is the advantage of using different types of cursors?

9.32 Explain forward-only cursors. Give an example of their use.

Amazon Ex. 1008.0374

9.33 Explain static cursors. Give an example of their use.

9.34 Explain keyset cursors. Give an example of their use.

9.35 Explain dynamic cursors. Give an example of their use.

9.36 What happens if you do not declare the transaction isolation level and the cursor type to the DBMS? Is this good or bad?

9.37 Explain the necessity of defining processing rights and responsibilities. How are such responsibilities enforced?

9.38 Explain the relationships among USER, ROLE, PERMISSION, and OBJECT for a generic database security system.

9.39 Should the DBA assume a firewall when planning security?

9.40 What should be done with unused DBMS features and functions?

9.41 Explain how to protect the computer that runs the DBMS.

9.42 With regard to security, what actions should the DBA take on user accounts and passwords?

9.43 List two elements of a database security plan.

9.44 Describe the advantages and disadvantages of DBMS-provided and application-provided security.

9.45 What is an SQL injection attack and how can it be prevented?

9.46 Explain how a database could be recovered via reprocessing. Why is this generally not feasible?

9.47 Define *rollback* and *rollforward*.

9.48 Why is it important to write to the log before changing the database values?

9.49 Describe the rollback process. Under what conditions should it be used?

9.50 Describe the rollforward process. Under what conditions should it be used?

9.51 What is the advantage of taking frequent checkpoints of a database?

9.52 Summarize the DBA's responsibilities for managing the DBMS.

9.53 What is a data repository? A passive data repository? An active data repository?

9.54 Explain why a data repository is important. What is likely to happen if one is not available?

Project Questions

9.55 Visit *www.oracle.com* and search for "Oracle Security Guidelines." Read articles at three of the links that you find and summarize them. How does the information you find compare with that in Figure 9-15?

9.56 Visit *www.msdn.microsoft.com* and search for "SQL Server Security Guidelines." Read articles at three of the links that you find and summarize them. How does the information you find compare with that in Figure 9-15?

9.57 Visit *www.mysql.com* and search for "MySQL Security Guidelines." Read articles at three of the links that you find and summarize them. How does the information you find compare with that in Figure 9-15?

Amazon Ex. 1008.0375

9.58 Use Google *(www.google.com)* or another search engine and search the Web for "Database Security Guidelines." Read articles at three of the links that you find and summarize them. How does the information you find compare with that in Figure 9-15?

9.59 Answer the following questions for the View Ridge Gallery database discussed in Chapter 7 with the tables shown in Figure 7-15.

A. Suppose that you are developing a stored procedure to record an artist who has never been in the gallery before, a work for that artist, and a row in the TRANS table to record the date acquired and the acquisition price. How will you declare the boundaries of the transaction? What transaction isolation level will you use?

B. Suppose that you are writing a stored procedure to change values in the CUSTOMER table. What transaction isolation level will you use?

C. Suppose that you are writing a stored procedure to record a customer's purchase. Assume that the customer's data are new. How will you declare the boundaries of the transaction? What isolation level will you use?

D. Suppose that you are writing a stored procedure to check the validity of the intersection table. Specifically, for each customer, your procedure should read the customer's transaction and determine the artist of that work. Given the artist, your procedure should then check to ensure that an interest has been declared for that artist in the intersection table. If there is no such intersection row, your procedure should create one. How will you set the boundaries of your transaction? What isolation level will you use? What cursor types (if any) will you use?

Marcia's Dry Cleaning



Assume that Marcia has hired you as a database consultant to develop an operational database having the following four tables (the same tables described at the end of Chapter 7):

CUSTOMER (*CustomerID*, FirstName, LastName, Phone, Email)

ORDER (InvoiceNumber, *CustomerID*, DateIn, DateOut, Subtotal, Tax, TotalAmount)

ORDER_ITEM (*InvoiceNumber*, ItemNumber, *Service*, Quantity, UnitPrice, ExtendedPrice)

SERVICE (Service, Description, UnitPrice)

A. Assume that Marcia's has the following personnel: two owners, a shift manager, a part-time seamstress, and two salesclerks. Prepare a two-to-three-page memo that addresses the following points:

1. The need for database administration.
2. Your recommendation as to who should serve as database administrator. Assume that Marcia's is not sufficiently large to need or afford a full-time database administrator.
3. Using Figure 9-1 as a guide, describe the nature of database administration activities at Marcia's. As an aggressive consultant, keep in mind that you can recommend yourself for performing some of the DBA functions.

B. For the employees described in part A, define users, groups, and permissions on data in these four tables. Use the security scheme shown in Figure 9-13 as an example. Create a table like that in Figure 9-12. Don't forget to include yourself.

C. Suppose that you are writing a stored procedure to create new records in SERVICE for new services that Marcia's will perform. Suppose that you know that while your procedure is running another stored procedure that records new or modifies existing customer orders and order line items can also be running. Additionally, suppose that a third stored procedure that records new customer data also can be running.

Amazon Ex. 1008.0376

1. Give an example of a dirty read, a nonrepeatable read, and a phantom read among this group of stored procedures.
2. What concurrency control measures are appropriate for the stored procedure that you are creating?
3. What concurrency control measures are appropriate for the two other stored procedures?



Assume that Morgan has hired you as a database consultant to develop an operational database having the same tables described at the end of Chapter 7 (Note that STORE uses the surrogate key StoreID):

STORE (StoreID, StoreName, City, Country, Phone, Fax, Email, Contact)

PURCHASE (PurchaseID, *StoreName*, Date, Description, Category, PriceUSD)

SHIPMENT (ShipmentID, ShipDate, ShipperName, ShipperInvoiceNumber, Origin, Destination)

SHIPMENT_ITEM (*ShipmentID*, *PurchaseID*, InsuredValue)

SHIPPER (ShipperName, Phone, Fax, Email, Contact)

A. Assume that Morgan personnel are the owner (Morgan), an office administrator, one full-time salesperson, and two part-time salespeople. Morgan and the office administrator want to process data in all tables. Additionally, the full-time salesperson can enter purchase and shipment data. The part-time employees can only read shipment data; they are not allowed to see InsuredValue, however. Prepare a three-to-five-page memo for the owner that addresses the following issues:

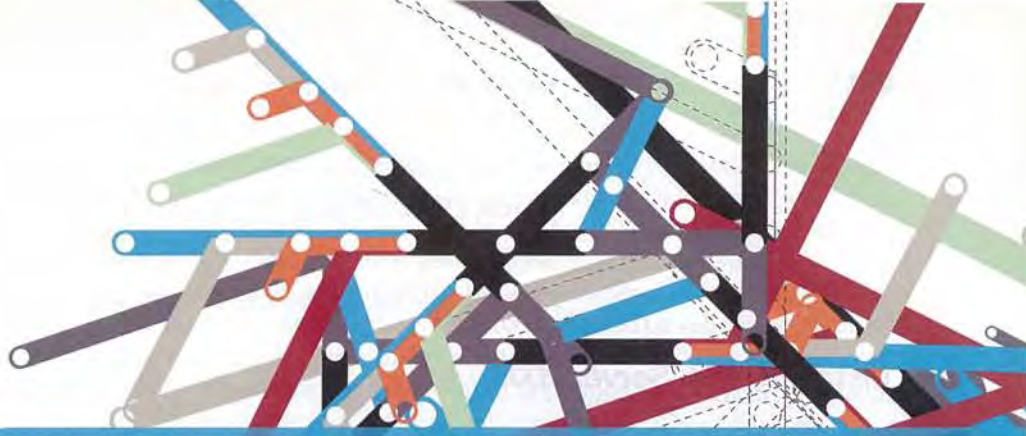
1. The need for database administration at Morgan.
2. Your recommendation as to who should serve as database administrator. Assume that Morgan is not sufficiently large that it needs or can afford a full-time database administrator.
3. Using Figure 9-1 as a guide, describe the nature of database administration activities at Morgan. As an aggressive consultant, keep in mind that you can recommend yourself for performing some of the DBA functions.

B. For the employees described in part A, define users, groups, and permissions on data in these five tables. Use the security scheme shown in Figure 9-13 as an example. Create a table like that in Figure 9-12. Don't forget to include yourself.

C. Suppose that you are writing a stored procedure to record new purchases. Suppose that you know that while your procedure is running, another stored procedure that records shipment data can be running, and a third stored procedure that updates shipper data can also be running.

1. Give an example of a dirty read, a nonrepeatable read, and a phantom read among this group of stored procedures.
2. What concurrency control measures are appropriate for the stored procedure that you are creating?
3. What concurrency control measures are appropriate for the two other stored procedures?

Amazon Ex. 1008.0377



10 Managing Databases with SQL Server 2008

Chapter Objectives

- To be able to install SQL Server and create a database.
- To be able to use SQL Server's graphical utilities.
- To be able to submit both SQL DDL and DML via the Microsoft SQL Server Management Studio.
- To understand the purpose and role of stored procedures and know how to create simple stored procedures.
- To understand the purpose and role of triggers and know how to create simple stored procedures.
- To understand how SQL Server implements concurrency control.
- To understand the fundamental features of SQL Server backup and recovery facilities.

This chapter describes the basic features and functions of Microsoft SQL Server 2008. The discussion uses the View Ridge Gallery database from Chapter 7, and it parallels the discussion of the database administration tasks in Chapter 9. The presentation is similar in scope and orientation to that of Oracle Database 11g in Chapter 10A and MySQL 5.1 in Chapter 10B.

SQL Server 2008 is a large and complicated product. In this one chapter, we will only be able to scratch the surface. Your goal should be to learn sufficient basics so that you can continue learning on your own or in other classes.

Amazon Ex. 1008.0378

The topics and techniques discussed here also apply to the earlier SQL Server 2005, although the exact functions of the SQL Server 2005 Management Studio vary a bit from the SQL Server 2008 version. The material you learn in this chapter will be applicable to that older version.

Installing SQL Server 2008

Microsoft SQL Server is an enterprise-class DBMS that has been around for many years. In 2005, SQL Server 2005 was released, followed by SQL Server 2008 in 2008. SQL Server 2008 is available in several versions, which can be reviewed at the Microsoft SQL Server 2008 Web site (*www.microsoft.com/sqlserver/2008/en/us/editions.aspx*). For our purposes, there are three editions you need to be aware of:

- **Enterprise Edition.** This is the most powerful and feature-laden version. It handles as many CPUs as the computer's operating system will support and includes data warehouse capabilities.
- **Standard Edition.** This is the basic commercial version. It does not have the complete feature set of the Enterprise Edition. It handles up to four CPUs and does not include data warehouse capabilities.
- **Express Edition.** This free, feature-limited version is available for download. It supports one CPU, 1 GByte of memory, and a maximum database size of 4 GBytes. Despite its limitations, it is a great learning tool when the Express Advanced version, which includes the SQL Server 2008 Management Studio and a basic reporting services package, is used (and did we mention that it's free?).

The SQL Server Express Edition was introduced with SQL Server 2005, and SQL Server 2008 includes the **SQL Server 2008 Express** version. The SQL Server Express Editions seem to be designed to compete with Sun Microsystems MySQL (see Chapter 10B). MySQL, while not having as many features as SQL Server, is an open-source database that has had the advantage of being available for free via download over the Internet. It has become widely used and very popular as a DBMS, supporting Web sites running the Apache Web server.

SQL Server 2008 Express Advanced is the version of SQL Server 2008 Express that contains an advanced feature package. The advanced features package includes Microsoft's GUI SQL Server management tool and support for SQL Server Reporting Services. Both of these features are well worth having (otherwise you will need to download the management tool separately), so download and install the version with the advanced features package.

For SQL Server 2008 Express, you can start at the SQL Server 2008 homepage at *www.microsoft.com/sqlserver/2008/en/us/default.aspx*. To install SQL Server 2008 Express Edition with Advanced Services, you need to download four programs:

1. **Microsoft .NET Framework version 3.5 Service Pack 1**. Before installing any of the SQL Server products, you must install the Microsoft .NET Framework 3.5 SP1.
2. **Microsoft Windows Installer 4.5**. Before installing any of the SQL Server products, you must install Windows Installer 4.5.
3. **Microsoft Windows PowerShell 1.0**. Before installing any of the SQL Server products, you must install Windows PowerShell 1.0.
4. **Microsoft SQL Server 2008 Express Advanced**. Note that **Microsoft SQL Server 2008 Management Studio** is included in the download of Microsoft SQL Server 2008 Express Advanced. If you choose to download the basic version of SQL Server 2008 Express, you must download and install this program separately. Microsoft SQL Server 2008 Management Studio is the graphical management utility for the SQL Server 2008 Express edition. While SQL Server 2008 was originally a command-line-oriented program, SQL Server 2008 Management Studio makes it much easier to work with SQL Server.

SQL Server 2008 Express is one of several Express Editions available from Microsoft. For more information on the Microsoft Express series of products, a good place to start is *www.microsoft.com/express*.

Amazon Ex. 1008.0379

Although SQL Server 2008 Express Advanced can handle most of the topics in this book, it cannot handle advanced business intelligence (BI) systems topics, which are discussed in Chapter 13. For that you will need the Enterprise Edition. An 180-day trial version can be downloaded from Microsoft (*www.microsoft.com/sqlserver/2008/en/us/trial-software.aspx*).

You may also want to download and install the following software from the SQL Server 2008 Feature Pack depending on whether SQL Server 2008 is installed on your workstation or a separate server (start at *www.microsoft.com/sqlserver/2008/en/us/trial-software.aspx* and click the **Download it now** button to ensure that you are taken to the most current set of software):

- Microsoft Core XML Services (MSXML) 6.0
- Microsoft SQL Server 2008 Command Line Utilities
- Microsoft SQL Server 2008 Native Client
- Microsoft SQL Server Driver for PHP (Listed as Microsoft SQL Server 2005 Driver for PHP
- Microsoft SQL Server 2008 Data Mining Add-ins for Microsoft Office 2007

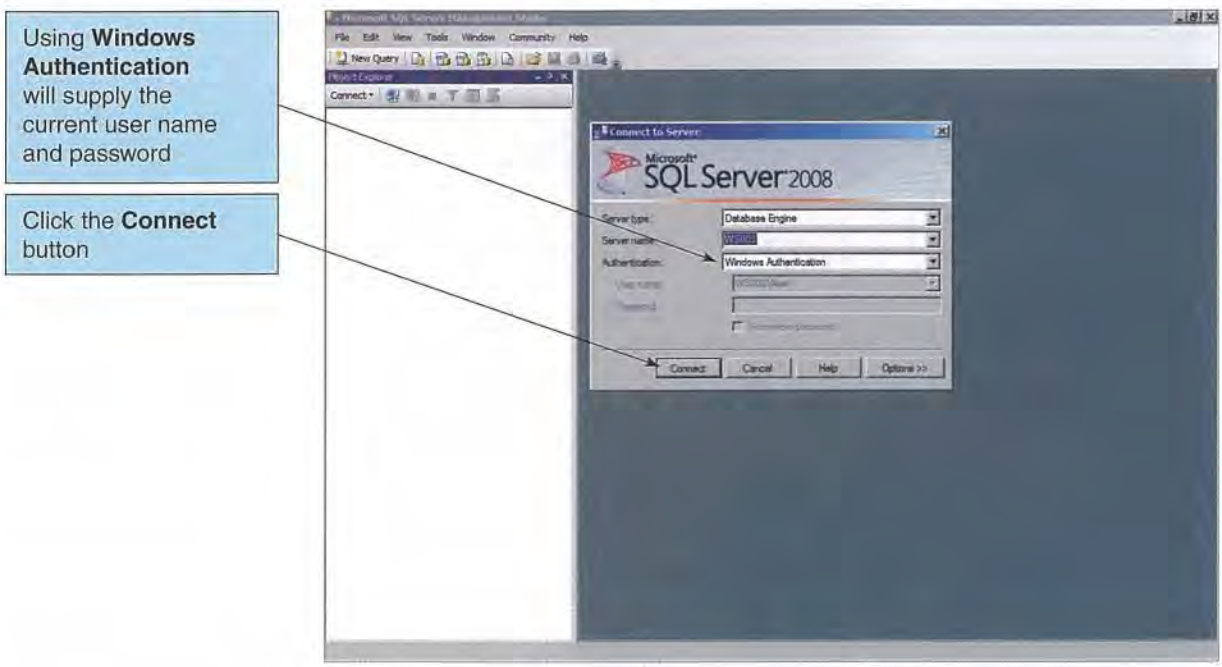
Be aware that SQL Server 2008 is an enterprise-class DBMS and, as such, is much more complex than Microsoft Access. Further, it does not include application development tools, such as form and report generators.

Regardless of which version of SQL Server 2008 you are going to use, you should install it now. You should then check for the latest service packs and patches to make sure your installation is as secure as possible.

The Microsoft SQL Server 2008 Management Studio

After you install SQL Server 2008, you can start working with SQL Server by opening the Microsoft SQL Server Management Studio. In this book, we are running SQL Server 2008 Enterprise Edition in Microsoft Server 2008. To open the Microsoft SQL Server Management Studio in Windows Server 2008 or in the Windows Vista operating system, select **Start | All Programs | Microsoft SQL Server 2008 | SQL Server Management Studio Express**.[1] The Microsoft SQL Server Management Studio **Connect to Server dialog box** appears, as shown in Figure 10-1.

Figure 10-1
The Connect to Server Dialog Box



[1] If you're using the Windows XP operating system, select **Start | All Programs | Microsoft SQL Server 2008 | SQL Server Management Studio**.

Amazon Ex. 1008.0380

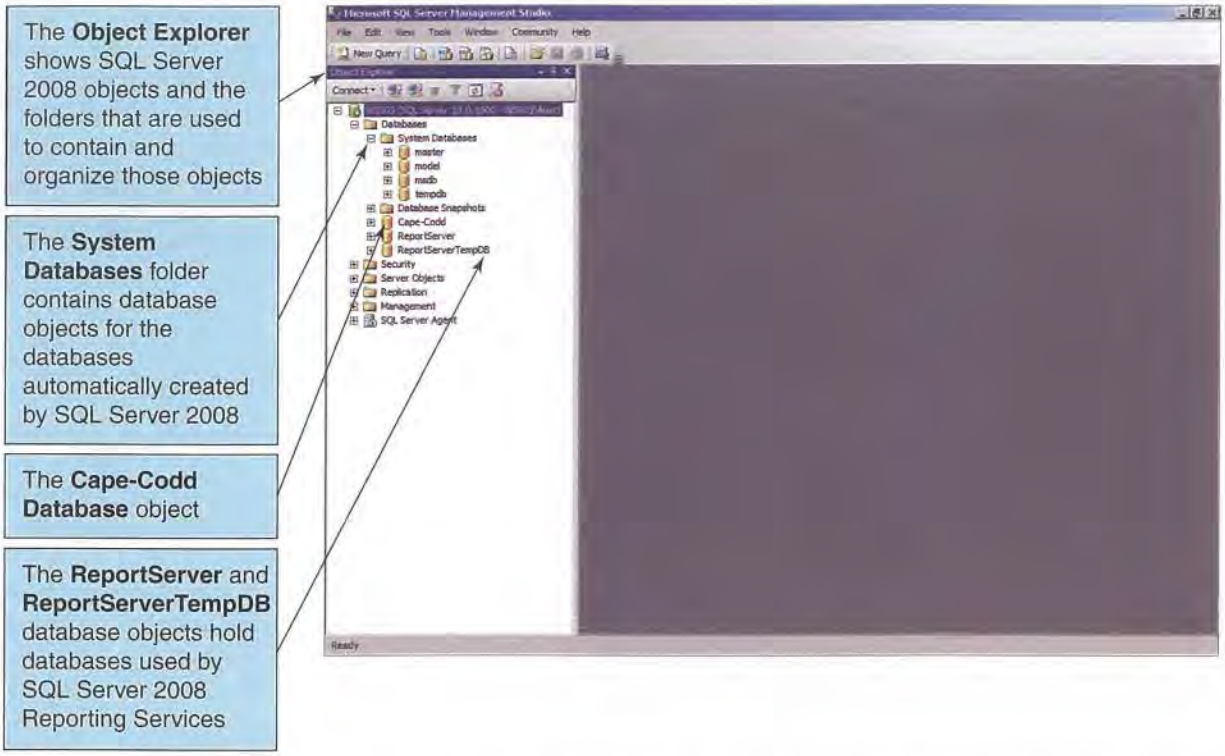


Figure 10-2
The Microsoft SQL Server Management Studio Window

After you have done this, find the folder icon labeled Databases in the Object Explorer, as shown in Figure 10-2. Click the plus sign to open it, and then open the System Database folder the same way. As shown in Figure 10-2, objects representing the databases managed by the SQL Server 2008 DBMS are displayed in these folders. For example, you can see the Cape Codd database that we used in our discussion of SQL queries in Chapter 2. We normally keep the System Databases folder closed because we usually do not work with these databases.

Creating an SQL Server 2008 Database

Now that the SQL Server 2008 is installed and the Microsoft SQL Server Management Studio is open, we can create a new database. We will create a database named **VRG** for the View Ridge Gallery database we designed in Chapter 6, and for which we wrote the SQL statements in Chapter 7.

Creating an SQL Server 2008 Database

1. Right-click the **Databases** folder in the Object Explorer to display a shortcut menu, as shown in Figure 10-3.
2. Click the **New Database** command to display the New Database dialog box, as shown in Figure 10-4.
3. Type the database name **VRG** in the Database Name text box, and then click the **OK** button. The database is created and the VRG database object is displayed in the Object Explorer, as shown in Figure 10-5. Click the plus (+) button to display the VRG folders, also shown in Figure 10-5.
4. Right-click the **VRG** database object to display a shortcut menu, and then click the **Properties** command. The Database Properties – VRG dialog box is displayed, as shown in Figure 10-5.
5. In the **Database Properties – VRG** dialog box, click the **Files** page object, as shown in Figure 10-5. The database files associated with the VRG database are displayed.
6. Click the **OK** button in the Database Properties – VRG dialog box to close the dialog box.

Amazon Ex. 1008.0381

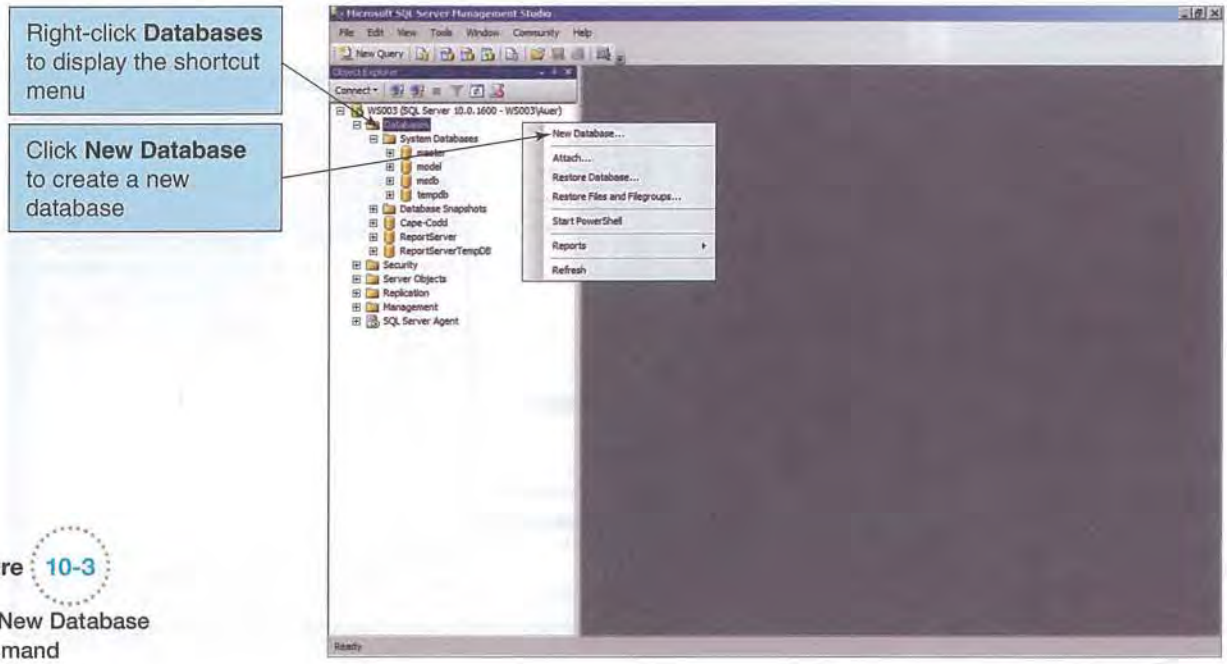


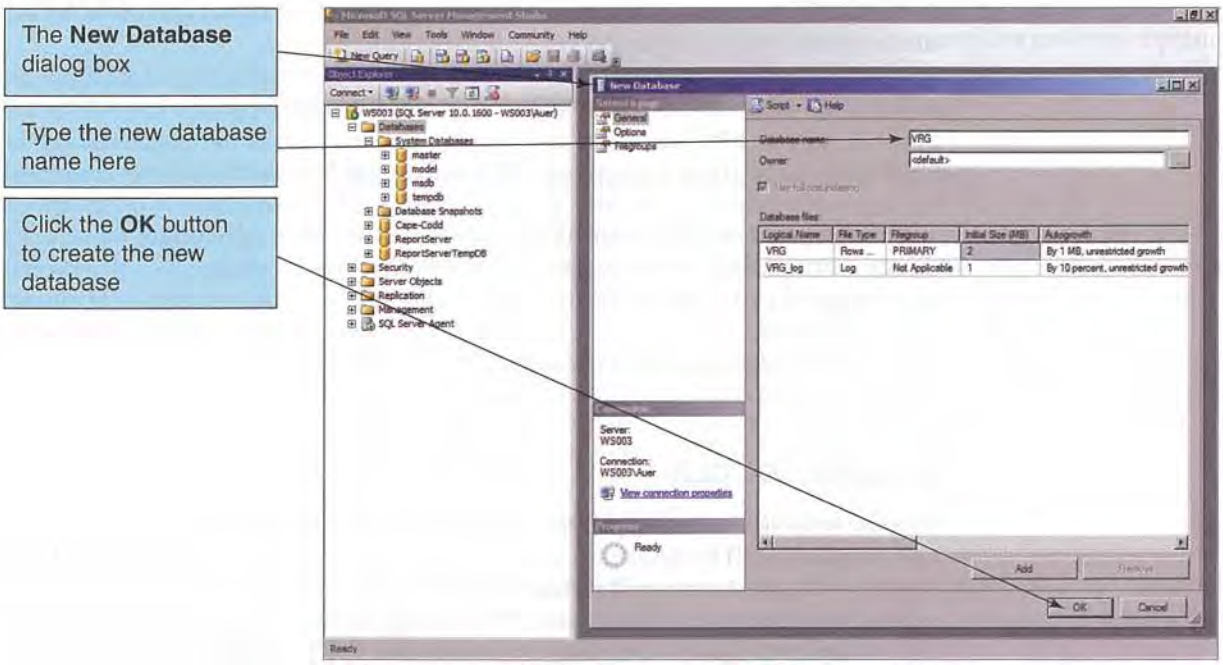
Figure 10-3

The New Database Command

If you look at the database files listed for the VRG database in the New Database dialog box in Figure 10-4, and displayed again on the Files page of the Database Properties – VRG dialog box, as shown in Figure 10-5, you will see that by default SQL Server creates one data file (logical name VRG) and one log file (logical name VRG_log) for each database. You can create multiple files for both data and logs and assign particular tables and logs to particular files and file groups. However, all of this is beyond the scope of our current discussion, however. To learn more about it on your own, use the SQL Server 2008 help system.

Figure 10-4

Naming the New Database



Amazon Ex. 1008.0382

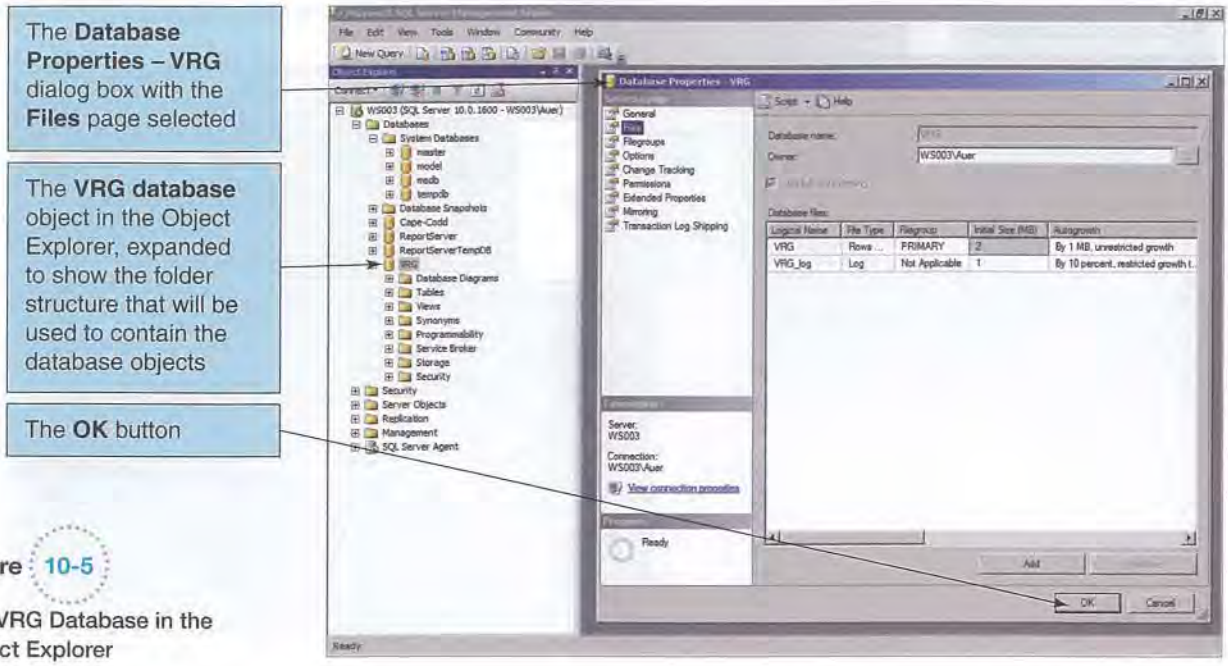


Figure 10-5

The VRG Database in the Object Explorer

SQL Server 2008 Utilities

Now that we have created the VRG database itself, we need to create the table and relationship structure of the database, and then populate the tables with data. What SQL Server 2008 utilities or tools shall we use to do this? It turns out that we have several choices.

SQL CMD and Microsoft PowerShell

In the beginning, there were the command-line utilities. A **command-line utility** is strictly text based. You are presented with a symbolic prompt to show you where to enter your commands. You type in a command (only one at a time) and press the [Enter] key to execute it. The results are displayed as plaintext (with some rudimentary character-based line- and box-drawing capabilities) in response. All major computer operating systems have their version of a command-line utility. For personal computer users using a Microsoft operating system, the classic example is the MS-DOS command line, which still exists in Windows as the CMD program.

For SQL Server, the classic command-line tool is the **SQL CMD utility**, which is still available as part of the Microsoft SQL Server 2008 Command Line Utilities package in the Microsoft SQL Server 2008 Feature Pack discussed earlier in this chapter. However, the functionality of the SQL CMD utility has been now been made available in the newer **Microsoft Windows PowerShell** utility. Microsoft PowerShell is a very powerful command-line and scripting utility, which runs command equivalents called **cmdlets**. By loading the **PowerShell sqlps utility**, the capabilities of the SQL CMD utility are made available in PowerShell, as shown in Figure 10-6. You can easily see that PowerShell carries on the command-line tradition (for more information on Microsoft Windows PowerShell, start at *www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx*, and then search the Microsoft Web site for specific information on SQL Server PowerShell.

Microsoft SQL CLR

If you are an applications developer using Microsoft Visual Studio as your **Integrated Development Environment (IDE)** for developing applications that use SQL Server, you will probably be using the **SQL Common Language Runtime (CLR)** technology that is built into SQL Server. SQL CLR enables SQL Server to provide Microsoft .NET CLR support, which allows application components written in programming languages such as Visual Basic .NET (VB.NET) to be stored and run in

Amazon Ex. 1008.0383

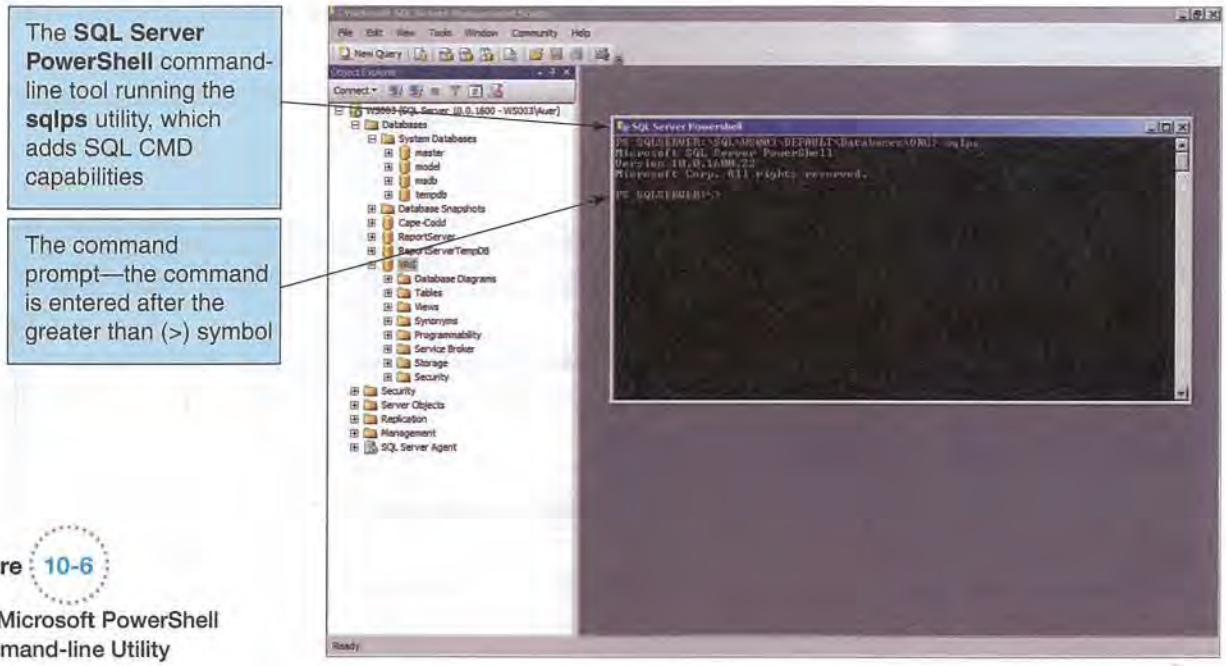


Figure 10-6
The Microsoft PowerShell Command-line Utility

SQL Server. Now database triggers and stored procedures (discussed in Chapter 7 and later in this chapter) can be written in VB.NET or C#.NET instead of standard SQL. SQL CLR is beyond the scope of this book, but you may use it in a class on applications development.

SQL Server 2008 GUI Displays

Although command-line utilities can be powerful, they can also be tedious and ugly. That's why GUI applications such as Windows were created in the first place. And popular personal databases such as Microsoft Access have certainly put GUI features to good use. Therefore, we have to ask "Is there a GUI display for SQL Server 2008?" The answer, of course, is "yes, there is." In fact, we are already using it. The Microsoft SQL Server Management Studio is the GUI display for SQL Server 2008.

Can we use the Microsoft SQL Server Management Studio the same way we use Microsoft Access to build tables using the GUI? Again, the answer is "yes." Figure 10-7 shows the start of a new table currently named Table_1. The column data shown are for the first column (ArtistID) in the VRG database ARTIST table described in Chapter 7. The design interface shown is very similar to Microsoft Access—basic column specifications are entered in a row in the top pane and specific column properties are set in the bottom pane. And, like Microsoft Access, the GUI will actually create an SQL statement and run that statement in the DBMS.

SQL Server 2008 SQL Statements and SQL Scripts

Because we have already argued that you need to know how to write and use SQL statements instead of relying on GUI tools, we come back to simply using SQL as the basis of our work. But we do not want to use a command-line utility, and we are not going to use the GUI tool in GUI mode, so what's left?

The answer is that the Microsoft SQL Server Management Studio provides us with an excellent SQL editing environment. This lets us take advantage of GUI capabilities while still working with text-based SQL statements. We do this by opening an SQL Query window, but using it more generally as an "SQL statement editor" window.

Opening an SQL Server 2008 SQL Statement Editor Window

1. Click the **VRG Database** object in the Object Browser to select it.
2. Click the **New Query** button. A new tabbed SQL Query window is displayed, and the SQL Editor toolbar appears, as shown in Figure 10-8.

Amazon Ex. 1008.0384

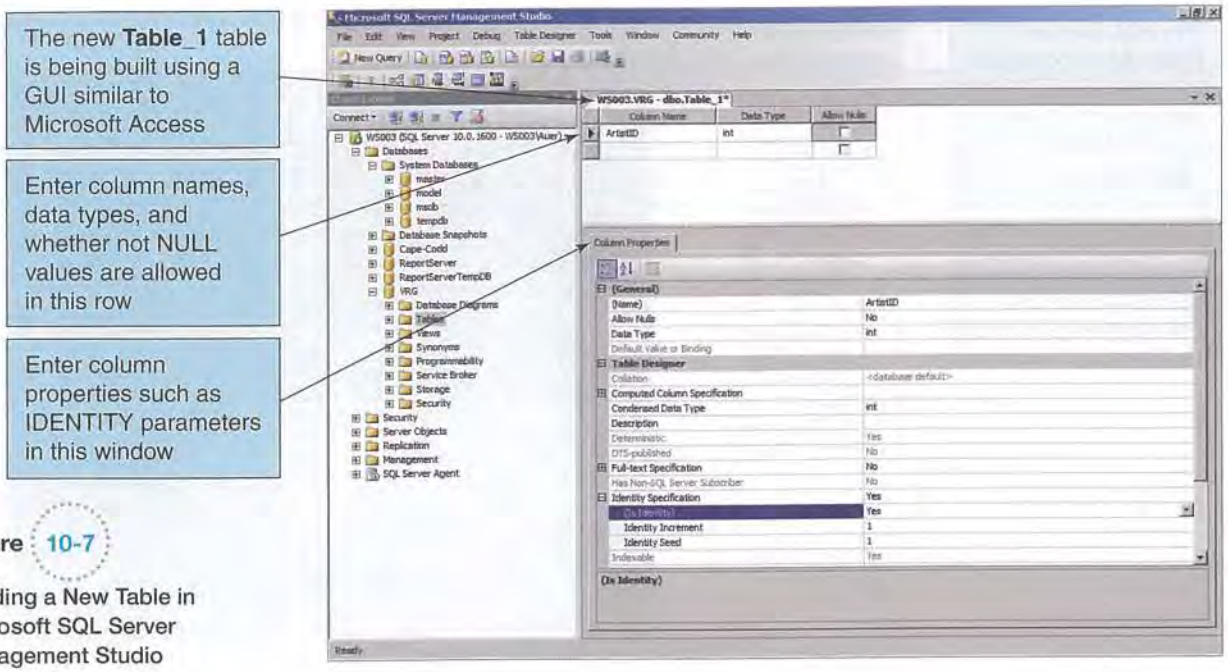


Figure 10-7

Building a New Table in Microsoft SQL Server Management Studio

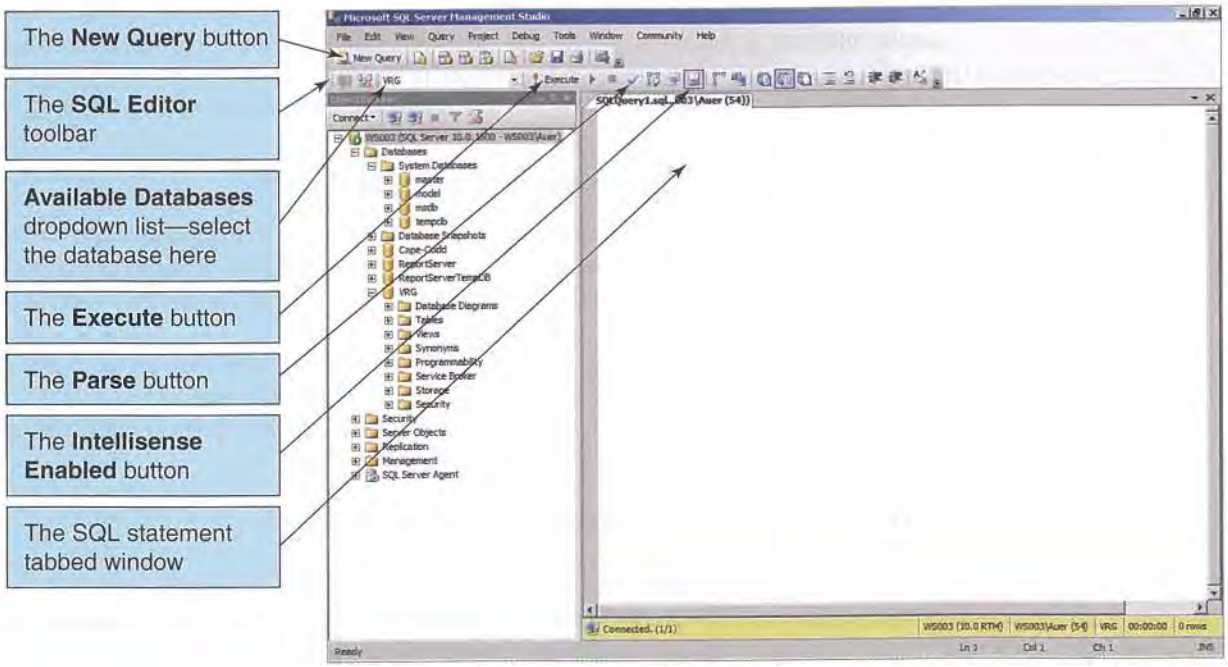
3. Click the **Intellisense Enabled** button to *disable* Intellisense. Intellisense is an object-search feature that, while useful for experienced developers, can be more confusing than helpful if you do not know how to use it. In this book, we will not use Intellisense, but if you want to learn more about it, you can read about it in SQL Server 2008 Books Online.

If the tabbed document window looks familiar, it is because it is the same window we used for SQL Server 2008 SQL queries in Chapter 2. We are simply using it for a slightly different purpose. This SQL editing environment will be our tool of choice.

One advantage of using this SQL editor is the ability to save and reuse SQL scripts. For SQL Server, **SQL scripts** are plaintext files labeled with the **.sql* file extension. We can save, open, and

Figure 10-8

The Microsoft SQL Server Management Studio SQL Editor



Amazon Ex. 1008.0385

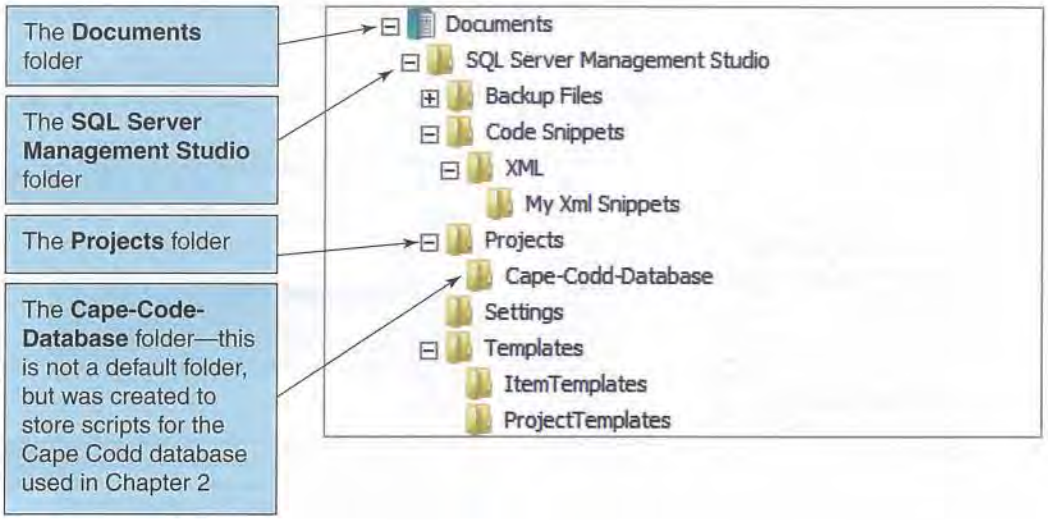


Figure 10-9
The Microsoft SQL Server Management Studio Folder Structure

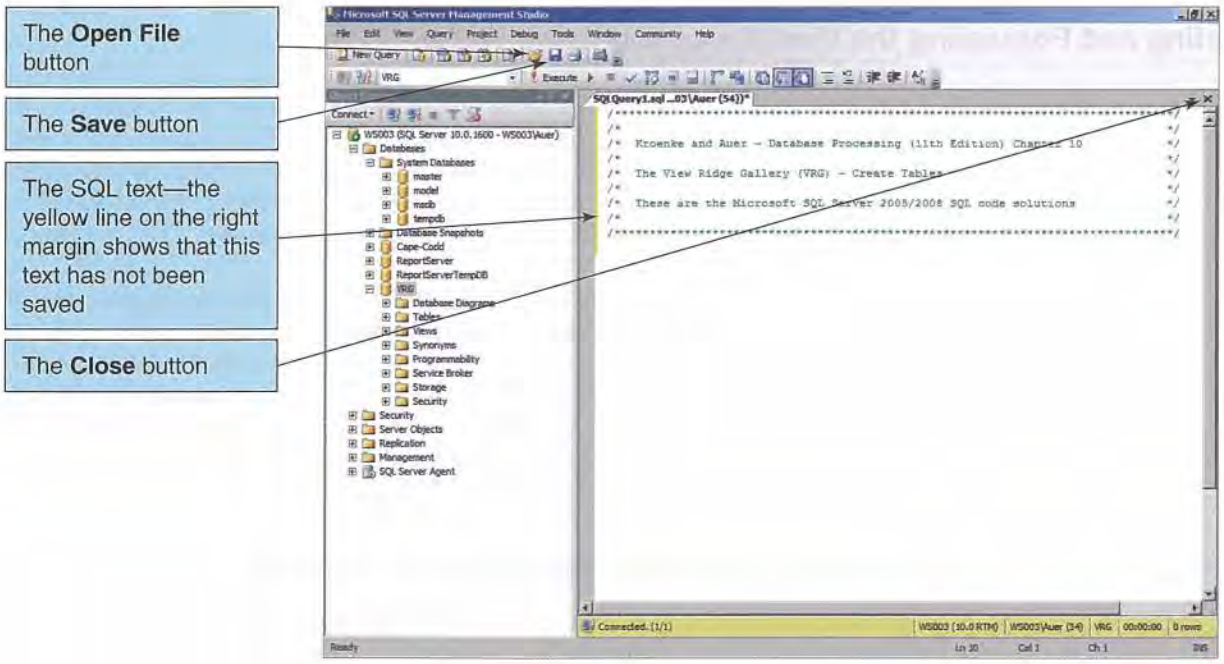
run (and rerun) SQL scripts. As shown in Figure 10-9, when the SQL Server Management Studio is installed, the installation process creates a set of folders in the user's Documents folder. By default, scripts are stored in the Projects folder. However, in order to organize our work we can create subfolders for various database projects. This is illustrated by the Cape-Codd-Database folder that was created to store the scripts (including scripts containing SQL queries) that we used with the Cape Codd Outdoor Sports database in Chapter 2.

An SQL script is composed of one or more SQL statements, which can include SQL script comments. **SQL script comments** are lines of text that do not run when the script is executed, but are used to document the purpose and contents of the script. Each comment line begins with the characters /* and ends with the characters */.

Creating and Saving an SQL Script

1. In the open tabbed SQL Query window, type the SQL comments shown in Figure 10-10.
2. Click the **Save** button. The Save File As dialog box is displayed, as shown in Figure 10-11.

Figure 10-10
Entering SQL statements in the SQL Editor



Amazon Ex. 1008.0386

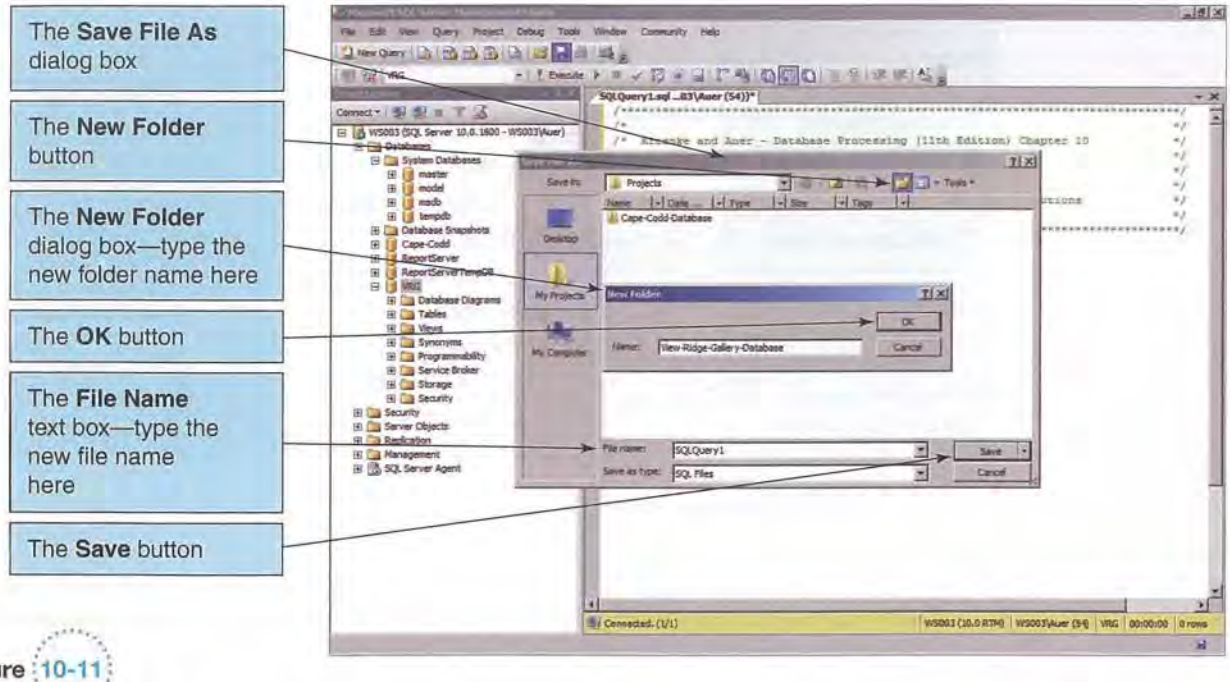


Figure 10-11
Saving the SQL Script in a New Folder

3. Click the **New Folder** button in the Save File As dialog box. The New Folder dialog box is displayed, as shown in Figure 10-11.
4. Type the folder name **View-Ridge-Gallery-Database** into the Name text box of the New Folder dialog box.
5. Click the **OK** Button on the New Folder dialog box.
6. Type the file name **VRG-Create-Tables** in the File Name text box of the Save File As dialog box.
7. Click the **OK** Button on the Save File As dialog box. The script is saved, the colored line on the right-hand margin of the text is changed to green, and the tab is renamed with the new file name.
8. Click the document **Close** button shown in Figure 10-10 to close the script window.

Creating and Populating the View Ridge Database Tables

As we have seen, tables and other SQL Server structures can be created and modified in two ways. The first is to write SQL code using either the CREATE or ALTER SQL statements we discussed in Chapter 7. The second is to use the SQL Server 2008 GUI display tools discussed earlier in this chapter. Although either method will work, CREATE statements are preferred for the reasons described in Chapter 7. Some professionals choose to create structures via SQL but then modify them with the GUI tools.

As discussed in Chapter 7, each DBMS product has its own variant or extension of SQL, where the extensions are usually **procedural language** extensions, which are additions that allow SQL to function similarly to a procedural programming language (for example, IF . . . THEN . . . ELSE structures). Microsoft's SQL Server version is called **Transact-SQL (T-SQL)**. We will point out specific Transact-SQL syntax as we encounter it in our discussion. For more on Transact-SQL, see the SQL Server 2008 Books Online article "Transact-SQL Reference" at *http://msdn.microsoft.com/en-us/library/bb510741.aspx*.

Creating the View Ridge Database Table Structure

The SQL Server 2008 version of the SQL CREATE TABLE statements for the View Ridge database in Chapter 7 are shown in Figure 10-12.

Amazon Ex. 1008.0387

Note that we are using the table name TRANS rather than TRANSACTION in Figure 10-12. This was done because TRANSACTION is a **reserved word** in SQL Server 2008.[2] Even if you make TRANSACTION a **delimited identifier** by placing the name in square brackets, as in [TRANSACTION], SQL Server still becomes confused when executing the logic of stored procedures and triggers. Life became much simpler for applications using this database when the table TRANSACTION was renamed to TRANS (which is not a Transact-SQL keyword, although TRAN is). WORK is not currently a Transact-SQL reserved word, but it is an ODBC reserved word (OBDC will be discussed in Chapter 11). Still, SQL Server is less sensitive to it, and therefore we can use it in delimited identifier form, enclosed in square brackets, as [WORK].

SQL Server supports surrogate keys, and the surrogate key columns are created using the **Transact-SQL IDENTITY property** with the primary key of the ARTIST, [WORK], CUSTOMER, and TRANS tables. The IDENTITY property has the syntax IDENTITY (seed, increment), where *seed* is the starting value for the surrogate key and *increment* is the value added to the previous surrogate key value each time a new value is created.

Creating the VRG Table Structure Using SQL Statements

1. In the SQL Server Management Studio, click the **Open File** button shown in Figure 10-10 to open the Open File dialog box.
2. Click the file name **VRG-Create-Tables.sql** to select the file.
3. Click the **Open** button in the Open File dialog box. The script is opened for use in a tabbed document window in the SQL editor.

Figure 10-12

The SQL Statements to Create the VRG Table Structure

```
CREATE TABLE ARTIST (
      ArtistID            Int              NOT NULL IDENTITY(1,1),
      LastName            Char(25)         NOT NULL,
      FirstName           Char(25)         NOT NULL,
      Nationality         Char(30)         NULL,
      DateOfBirth         Numeric(4)       NULL,
      DateDeceased        Numeric(4)       NULL,
      CONSTRAINT   ArtistPK            PRIMARY KEY(ArtistID),
      CONSTRAINT   ArtistAK1           UNIQUE(LastName, FirstName),
      CONSTRAINT   NationalityValues   CHECK
                   (Nationality IN ('Canadian', 'English', 'French',
                    'German', 'Mexican', 'Russian', 'Spanish',
                    'United States')),
      CONSTRAINT   BirthValuesCheck    CHECK (DateOfBirth < DateDeceased),
      CONSTRAINT   ValidBirthYear      CHECK
                   (DateOfBirth LIKE '[1-2][0-9][0-9][0-9]'),
      CONSTRAINT   ValidDeathYear      CHECK
                   (DateDeceased LIKE '[1-2][0-9][0-9][0-9]')
      );

CREATE TABLE WORK (
      WorkID              Int              NOT NULL IDENTITY(500,1),
      Title               Char(35)         NOT NULL,
      Copy                Char(12)         NOT NULL,
      Medium              Char(35)         NULL,
      [Description]       Varchar(1000)    NULL DEFAULT 'Unknown provenance',
      ArtistID            Int              NOT NULL,
      CONSTRAINT   WorkPK              PRIMARY KEY(WorkID),
      CONSTRAINT   WorkAK1             UNIQUE(Title, Copy),
      CONSTRAINT   ArtistFK            FOREIGN KEY(ArtistID)
                        REFERENCES ARTIST(ArtistID)
                           ON UPDATE NO ACTION
                           ON DELETE NO ACTION

      );
```

[2] For a complete list of SQL Server 2008 reserved keywords, OBDC reserved keywords (ODBC is discussed in Chapter 13) and potential future SQL Server keywords, see the SQL Server 2008 Books Online article on *Reserved Keywords (Transact-SQL)* at *http://msdn.microsoft.com/en-us/library/ms189822.aspx*.

Amazon Ex. 1008.0388

```
CREATE TABLE CUSTOMER (
      CustomerID          Int              NOT NULL IDENTITY(1000,1),
      LastName            Char(25)         NOT NULL,
      FirstName           Char(25)         NOT NULL,
      Street              Char(30)         NULL,
      City                Char(35)         NULL,
      [State]             Char(2)          NULL,
      ZipPostalCode       Char(9)          NULL,
      Country             Char(50)         NULL,
      AreaCode            Char(3)          NULL,
      PhoneNumber         Char(8)          NULL,
      Email               Varchar(100)     Null,
      CONSTRAINT    CustomerPK          PRIMARY KEY(CustomerID),
      CONSTRAINT    EmailAK1            UNIQUE(Email)
      );

CREATE TABLE TRANS (
      TransactionID       Int              NOT NULL IDENTITY(100,1),
      DateAcquired        Datetime         NOT NULL,
      AcquisitionPrice    Numeric(8,2)     NOT NULL,
      DateSold            Datetime         NULL,
      AskingPrice         Numeric(8,2)     NULL,
      SalesPrice          Numeric(8,2)     NULL,
      CustomerID          Int              NULL,
      WorkID              Int              NOT NULL,
      CONSTRAINT    TransPK             PRIMARY KEY(TransactionID),
      CONSTRAINT    TransWorkFK         FOREIGN KEY(WorkID)
                        REFERENCES WORK(WorkID)
                           ON UPDATE NO ACTION
                           ON DELETE NO ACTION,
      CONSTRAINT    TransCustomerFK     FOREIGN KEY(CustomerID)
                        REFERENCES CUSTOMER(CustomerID)
                           ON UPDATE NO ACTION
                           ON DELETE NO ACTION,
      CONSTRAINT    SalesPriceRange  CHECK
                        ((SalesPrice > 0) AND (SalesPrice <=500000)),
      CONSTRAINT    ValidTransDate      CHECK (DateAcquired <= DateSold),
      );

CREATE TABLE CUSTOMER_ARTIST_INT(
      ArtistID            Int              NOT NULL,
      CustomerID          Int              NOT NULL,
      CONSTRAINT    CAIntPK             PRIMARY KEY(ArtistID, CustomerID),
      CONSTRAINT    CAInt_ArtistFK      FOREIGN KEY(ArtistID)
                        REFERENCES ARTIST(ArtistID)
                           ON UPDATE NO ACTION
                           ON DELETE CASCADE,
      CONSTRAINT    CAInt_CustomerFK    FOREIGN KEY(CustomerID)
                        REFERENCES CUSTOMER(CustomerID)
                           ON UPDATE NO ACTION
                           ON DELETE CASCADE
      );
```

Figure 10-12
(Continued)

4. Click the **Intellisense Enabled** button to disable Intellisense.
5. Type in the SQL statements shown in Figure 10-12. Be sure to save the script often, and save the script a final time after you have finished entering all the SQL statements.
6. Scroll to the top of the script. The completed SQL script to create the VRG table structure appears as shown in Figure 10-13.
7. Make sure the database name **VRG** is shown in the Available Databases drop-down list as shown in Figure 10-13. If it isn't, click the Available Databases drop-down list arrow and select VRG.
8. Click the **Parse** button shown in Figure 10-13. A message window appears below the tabbed document window. If this window contains the message "The command(s)

Amazon Ex. 1008.0389

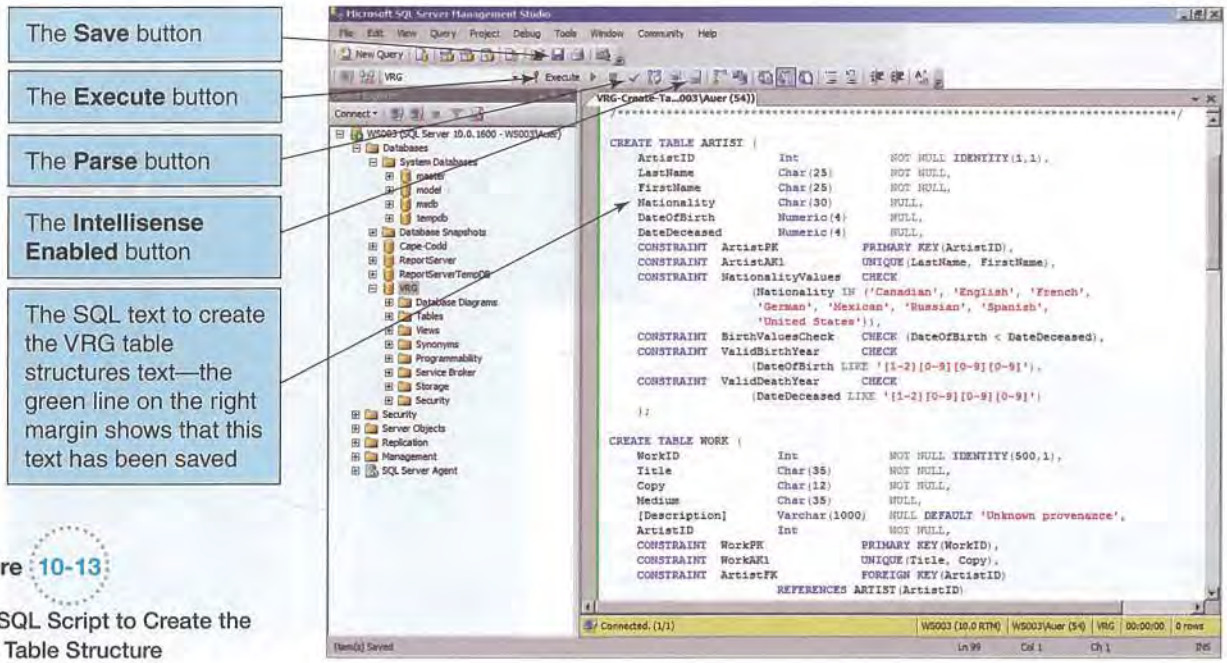


Figure 10-13

The SQL Script to Create the VRG Table Structure

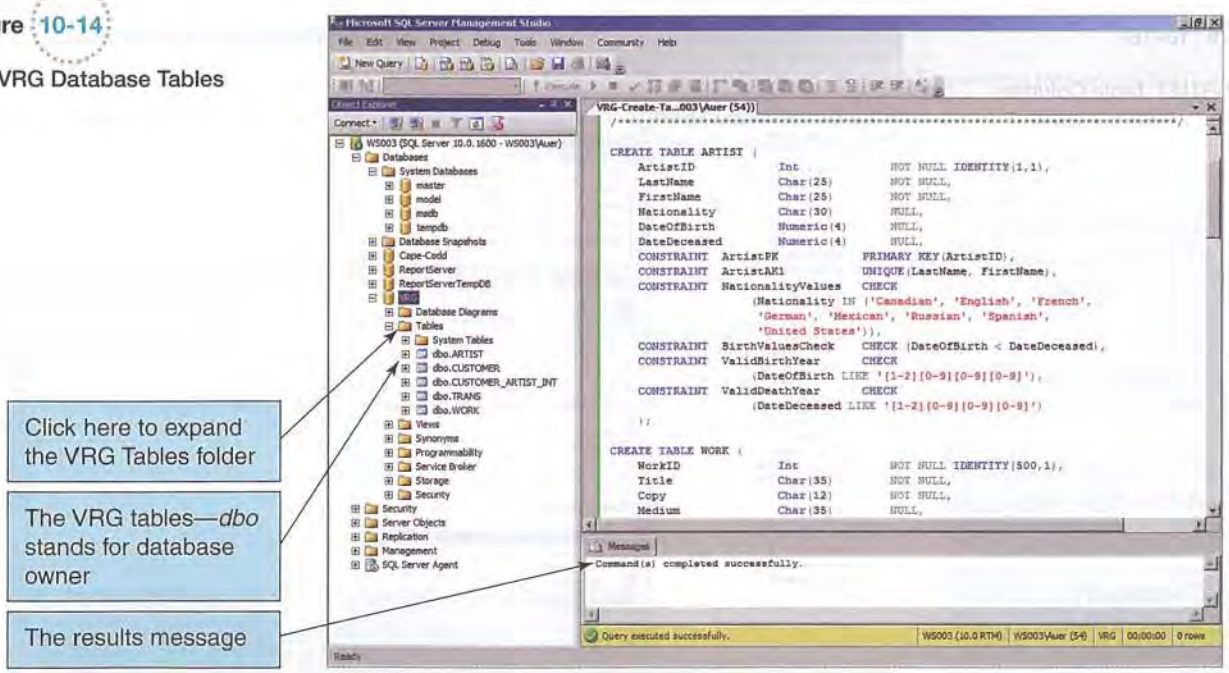
completed successfully." there are probably no errors in the script. However, if any error messages are displayed, then you have errors in your SQL statements. Correct any errors. Repeat this step until you see the message "The command(s) completed successfully."

9. Click the **Save** button to save your debugged SQL script.
10. Click the **Execute** button shown in Figure 10-14. The tables are created, and the Results message "The command(s) completed successfully." appears.
11. Expand the VRG Tables folder to see the VRG tables, as shown in Figure 10-14.
12. Click the document window **Close** button to close the SQL script.

Now we have created the VRG table and relationship structure. By the way, in Figure 10-14 *dbo* stands for database owner. That will be you if you installed SQL Server and created this database.

Figure 10-14

The VRG Database Tables



Amazon Ex. 1008.0390

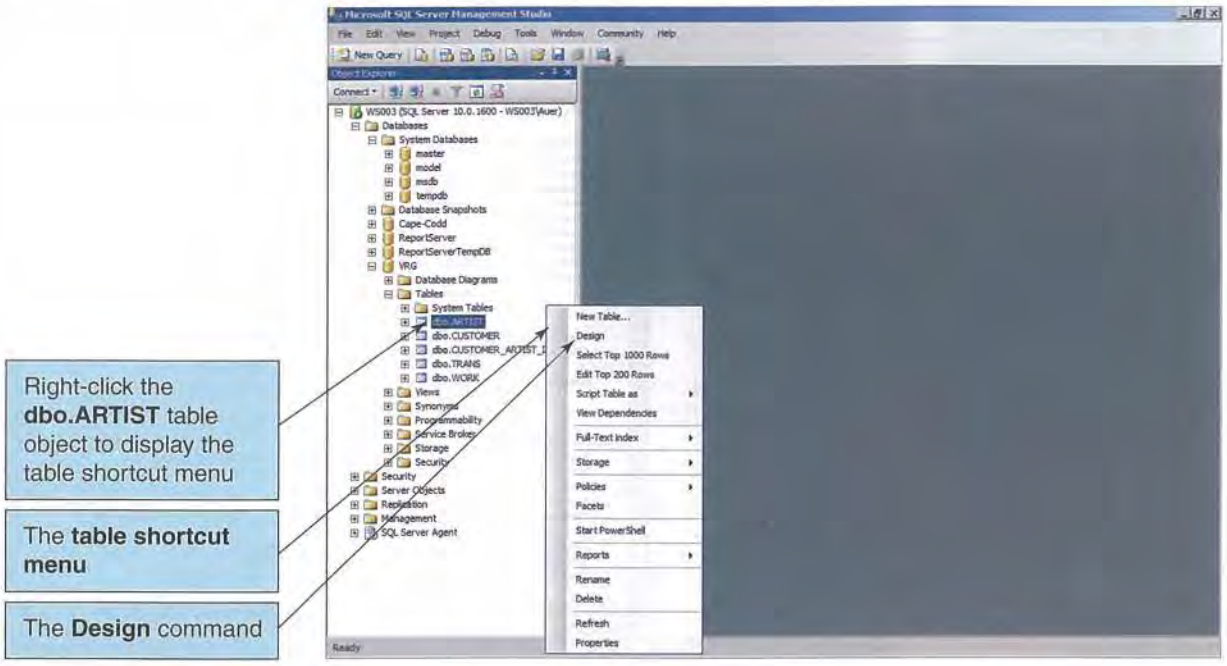


Figure 10-15

The Table Shortcut Menu

Reviewing Database Structures in the SQL Server GUI Display

After building the table structure using SQL statements, we can inspect the results using the SQL Server GUI tools. Let's take a look at the ARTIST table, particularly at the properties of the ArtistID primary key.

Viewing the ARTIST Table Structure in the GUI Display

1. In the SQL Server Management Studio Object Browser, right-click the **dbo.ARTIST** table object to open the table shortcut menu, as shown in Figure 10-15.
2. In the table shortcut menu, click the **Design** command. The ARTIST table design is displayed in a tabbed document window, as shown in Figure 10-16, with the Identity Specification properties expanded. Compare this figure to Figure 10-7.
3. Click the document window **Close** button to close the ARTIST table's columns and column properties tabbed window.

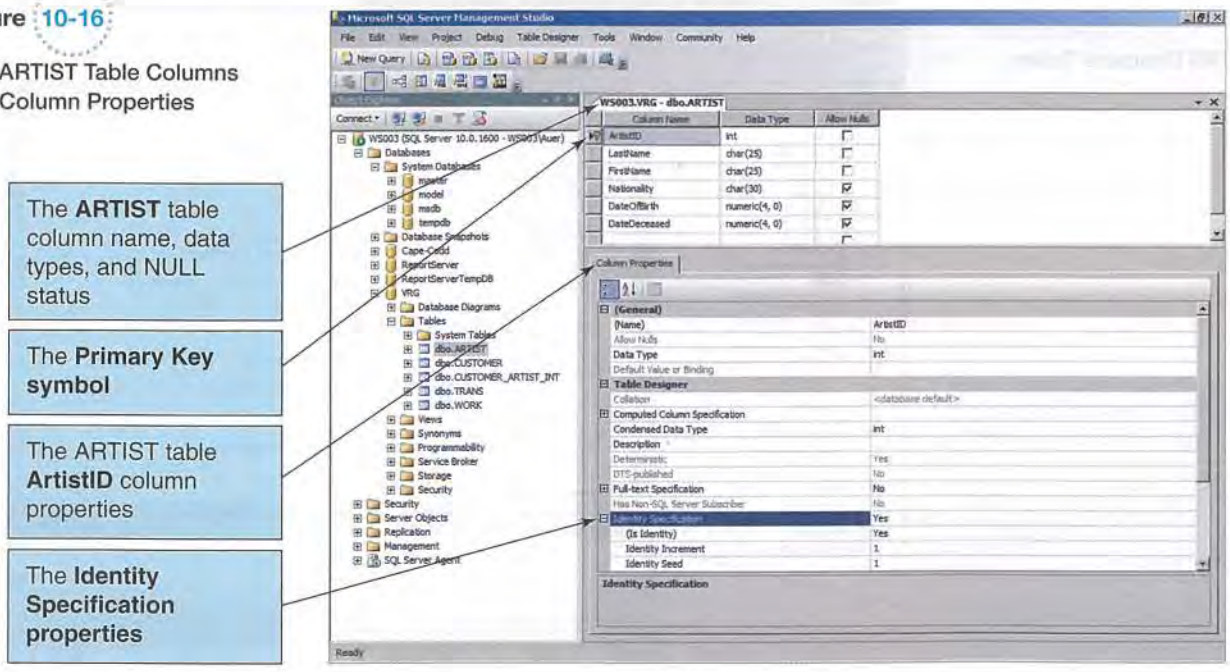


Figure 10-16

The ARTIST Table Columns and Column Properties

Amazon Ex. 1008.0391

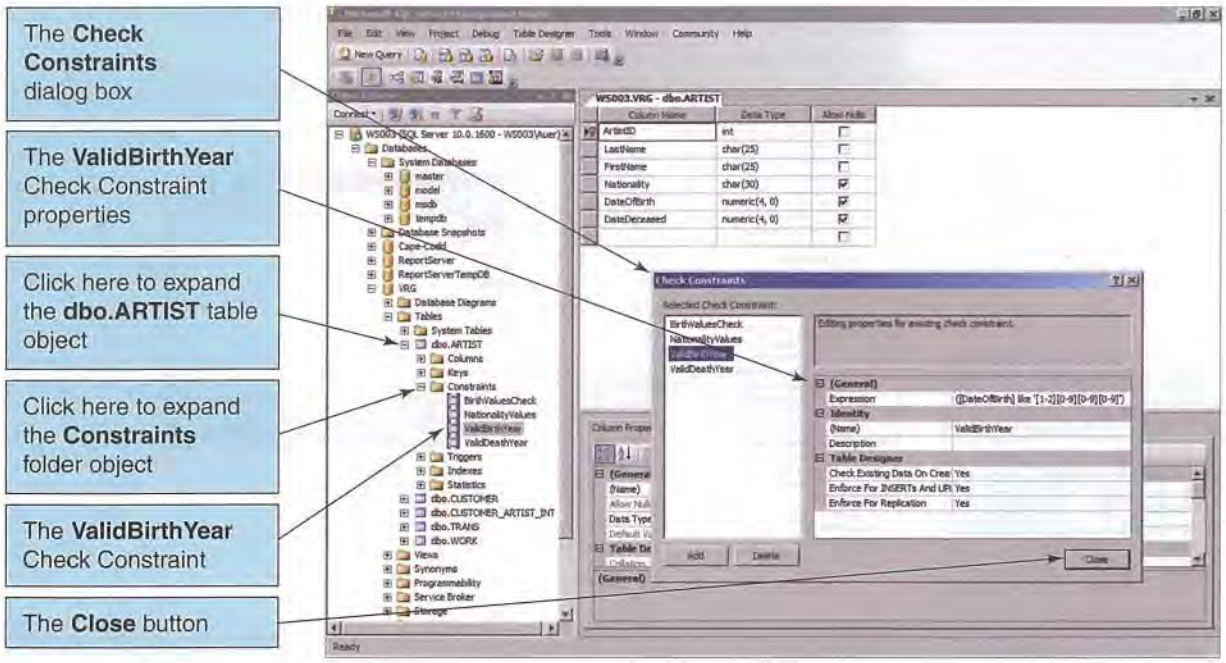


Figure 10-17

The ARTIST Table Check Constraints

We can also inspect the constraints on the ARTIST table. We'll take a look at the ValidBirthYear constraint we coded into our SQL CREATE TABLE statements.

Viewing the ARTIST Table Constraints in the GUI Display

1. In the SQL Server Management Studio Object Browser, expand the **dbo.ARTIST** table object.
2. Expand the **Constraints** folder. The ARTIST table constraint objects are displayed in the Object Browser.
3. Right-click the **ValidBirthYear** constraint object to display the shortcut menu.
4. Click the **Modify** command in the shortcut menu. The Check Constraints dialog box appears, as shown in Figure 10-17.
5. Note that the check constraint itself is located in the Expression text box of the (General) properties group, and it can be edited there if needed. At this point, however, there is no need to change the constraint.
6. Click the **Close** button to close the Check Constraints dialog box.
7. Click the document window **Close** button to close the ARTIST table's columns and column properties tabbed window.
8. Contract (minimize) the **dbo.ARTIST** table object display.

Clearly, it is easier to key the constraint into SQL statements than it would be to type it as shown in this window!

Throughout this chapter, you may see dialog boxes similar to the one shown in Figure 10-17 that have properties that reference something about replication. All such references refer to distributed SQL Server databases in which data are placed in two or more databases and updates to them are coordinated in some fashion. We will not consider that topic here, so ignore any references to replication. You can learn more by searching for the replication topic in the SQL Server 2008 documentation.

To ensure that the VRG database relationships were created correctly, let's create a database diagram for the VRG database. We can then use that diagram as the basis for checking the relationships among the various tables.

Amazon Ex. 1008.0392

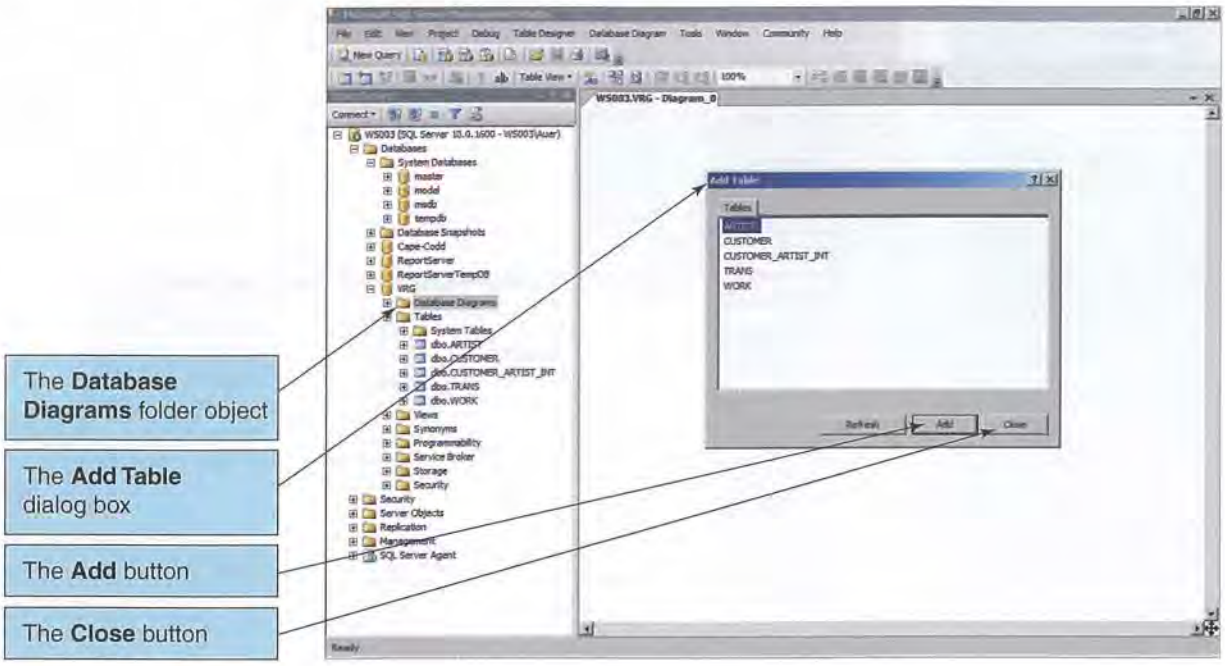


Figure 10-18
The Add Table Dialog Box

Creating an SQL Server Database Diagram

1. In the SQL Server Management Studio Object Browser, right-click the **VRG Database Diagrams** folder object to display the shortcut menu.
2. Click the **Install Diagram Support** command on the shortcut menu.
3. A Microsoft SQL Server Management Studio dialog box appears asking if you want to create some objects that support database diagrams. Of course you do! Isn't that the command you just gave? Click the **Yes** button!
4. Right-click the **VRG Database Diagrams** folder object to display the shortcut menu.
5. Click the **New Database Diagram** command on the shortcut menu. The Add Table dialog box appears, as shown in Figure 10-18.
6. Click the **Add** button to add the highlighted ARTIST table to the database diagram.
7. Repeat step 6 for each of the other four tables in the VRG database
8. Click the **Close** button to close the Add Table dialog box.
9. Rearrange the VRG tables in the database diagram until it appears as shown in Figure 10-19.
10. Click the **Save** button, and name the diagram **VRG-Database-Diagram**.
11. Expand the **VRG Database Diagrams** folder object to display the VRG-Database-Diagram object, which is shown in Figure 10-20.

Now we can use the VRG database diagram to view the properties of a relationship. We will take a look at the relationship between ARTIST and WORK, checking the referential integrity constraint and the cascading update and deletion behavior between ARTIST and WORK.

Viewing Relationship Properties

1. Right-click the **ARTIST** table in the VRG Database Diagram to display the short-cut menu as shown in Figure 10-20. Note how many actions are available from this short-cut menu.
2. Click the **Relationships** command to display the Foreign Key Relationships dialog box, as shown in Figure 10-21.
3. Expand the **Tables and Columns Specifications** property section, as shown in Figure 10-21.

Amazon Ex. 1008.0393

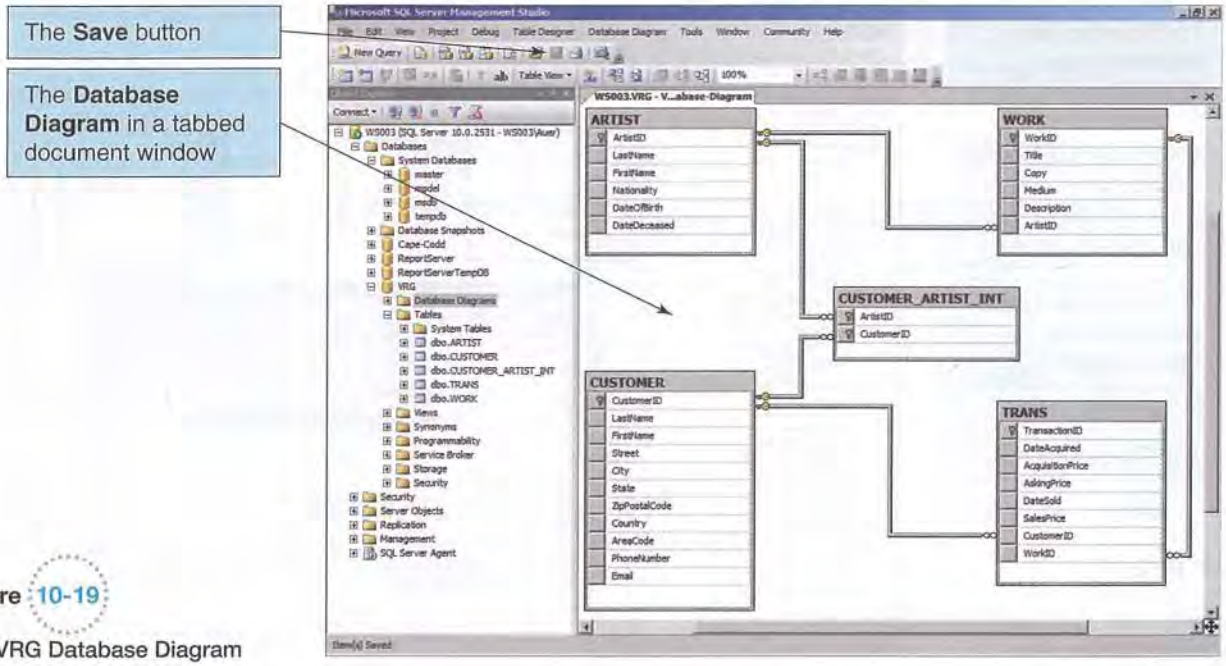


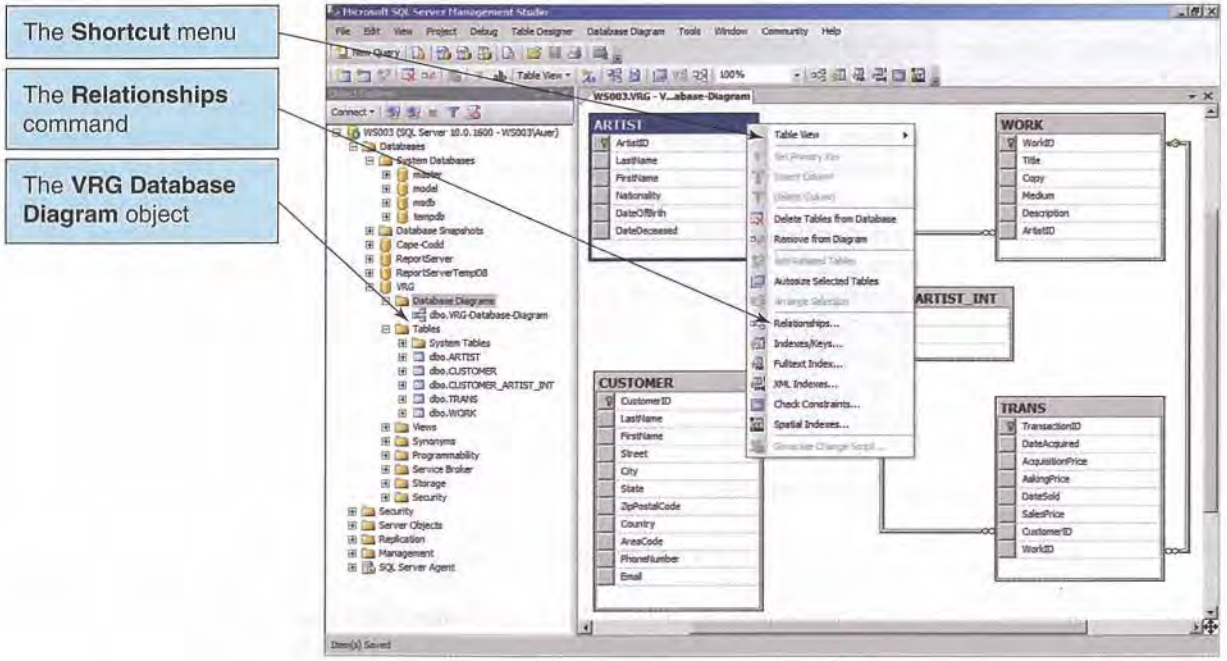
Figure 10-19

The VRG Database Diagram

4. In Figure 10-21, note that the Tables and Columns Specifications property section details the *primary and foreign keys* in the relationship between the two tables.
5. In Figure 10-21, note that the Check Existing Data On Creation Or Re-Enabling property is enabled. This is the *referential integrity constraint* between the ArtistID column in WORK and the ArtistID column in ARTIST.
6. In the Foreign Key Relationships dialog box, expand the **INSERT and UPDATE Specification**, and then scroll down until it is visible. The property settings indicate that neither updates to the primary key of ARTIST nor deletions of rows in ARTIST will cause any cascading actions in the WORK table.
7. Click the **Close** button to close the Foreign Key Relationships dialog box.
8. Click the **Close** button to close the VRG Database Diagram.
9. Contract (minimize) the **System Databases** folder.

Figure 10-20

The Database Diagram Table Shortcut Menu



Amazon Ex. 1008.0394

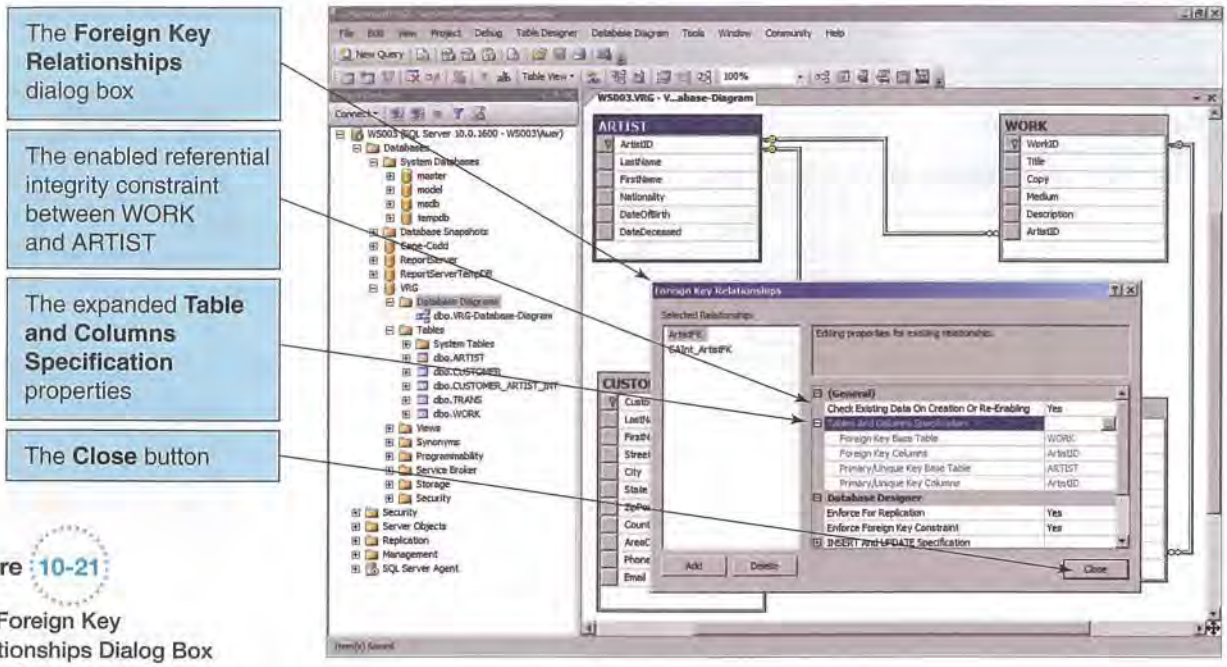


Figure 10-21

The Foreign Key Relationships Dialog Box

Indexes

As discussed in Appendix D, an **index** is a special data structure that is created to improve database performance. SQL Server automatically creates an index on all primary and foreign keys. A developer can also direct SQL Server to create an index on other columns that are frequently used in WHERE clauses or on columns that are used for sorting data when sequentially processing a table for queries and reports.

SQL Server supports two kinds of indexes for tables: clustered and nonclustered. With a **clustered index**, the data are stored in the bottom level of the index and in the same order as that index. With a **nonclustered index**, the bottom level of an index does not contain data; it contains pointers to the data. Because rows can be sorted only in one physical order at a time, only one clustered index is allowed per table. Retrieval is faster with clustered indexes than nonclustered indexes. Updating is normally faster as well with clustered indexes, but not if there are many updates in the same spot in the middle of the relation. For more information on clustered and nonclustered indexes, see the SQL Server 2008 Books Online article "Tables and Index Data Structures Architecture," which is available at *http://msdn.microsoft.com/en-us/library/ms180978.aspx*.

SQL Server 2008 also supports indexes on XML data (see the discussion of XML in Chapter 12) and geometry or geographic spatial data types. For each column of XML data, four types of indexes can be created: a primary XML index, a PATH secondary XML index, a VALUE secondary XML index, and a PROPERTY secondary XML index. For each column of spatial data, we can create a spatial index. For more information on XML indexes, see the SQL Server 2008 Books Online article "Tables and Index Data Structures Architecture" (*http://msdn.microsoft.com/en-us/library/ms180978.aspx*). For more information on spatial indexes, see the SQL Server 2008 Books Online article "Spatial Indexing Overview" (*http://msdn.microsoft.com/en-us/library/bb964712.aspx*).

To illustrate how to create indexes, we will create a new index on the ZipPostalCode column in the CUSTOMER table.

Creating a New Index

1. In the SQL Server Management Studio Object Browser, expand the **dbo.CUSTOMER** table object.
2. Expand the dbo.CUSTOMER table **Index** folder object to display the existing indexes.
3. Right-click the **Index** folder object to display the shortcut menu.

Amazon Ex. 1008.0395

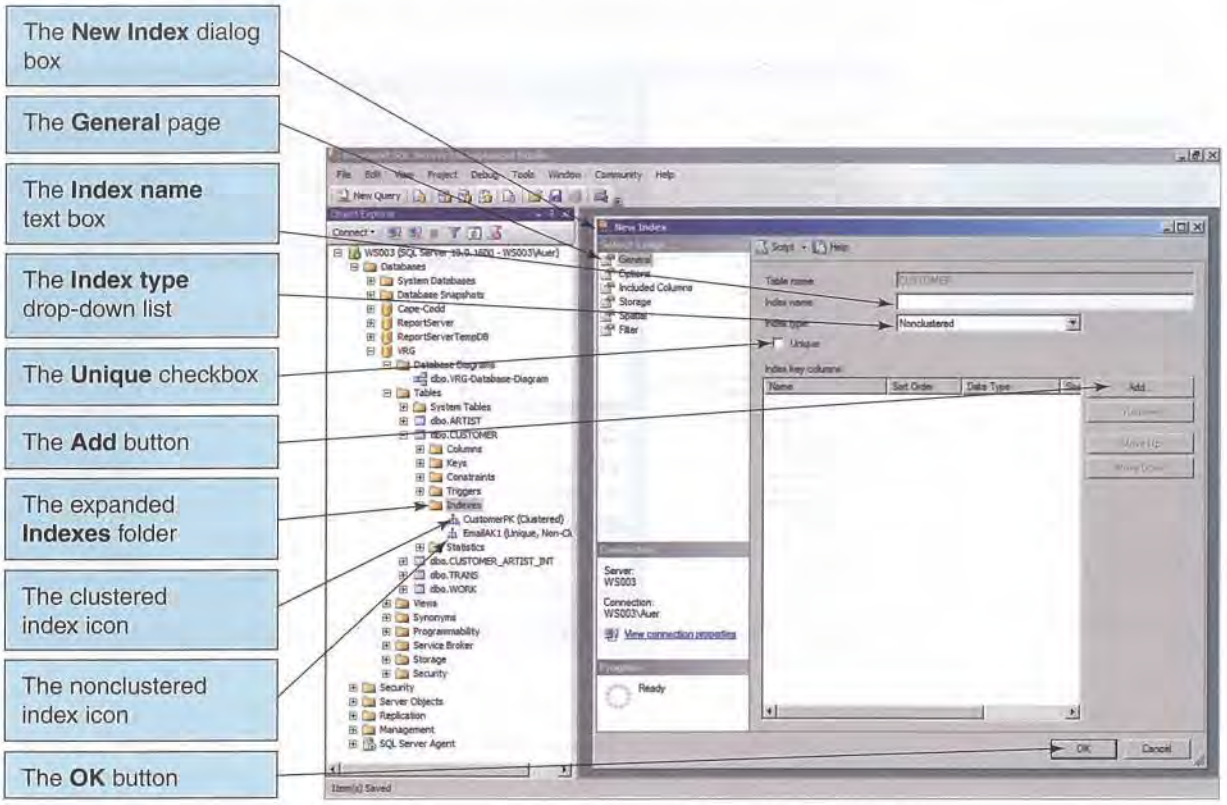


Figure 10-22

The New Index Dialog Box

4. Click the **New Index** command on the shortcut menu. The New Index dialog box appears, as shown in Figure 10-22.
5. Type the name **ZipPostalCodeIndex** in the Index Name text box of the New Index dialog box.
6. Click the **Add** button on the New Index dialog box to display the Select Columns from 'dbo.CUSTOMER' dialog box.
7. Click the **check box** for the ZipPostalCode column in the Select Columns from 'dbo.CUSTOMER' dialog box.
8. Click the **OK** button on the Select Columns from 'dbo.CUSTOMER' dialog box. The New Index dialog box now appears as shown in Figure 10-23.
9. Click the **OK** button in the New Index dialog box to create the ZipPostalCodeIndex. The ZipPostalCodeIndex nonclustered index object appears in the Indexes folder of the dbo.CUSTOMER table in the Object Browser.
10. Collapse the **dbo.CUSTOMER** table structure in the Object Browser.

Populating the VRG Tables with Data

You can enter data into SQL Server either by entering data into a table grid in the Microsoft SQL Server Management Studio GUI display or by using SQL INSERT statements. The Microsoft SQL Server Management Studio GUI display is more useful for occasional data edits than for populating all the tables of a new database. You can open a table grid for data entry by a right-clicking the table name to display a shortcut menu, then click the Edit Top 200 Rows command.

However, we will use the same method for populating the VRG database tables that we used to create the table structure: an SQL script. But before we do that, we need to address the surrogate key values issue raised in Chapter 7. The data shown in Figure 7-16 is sample data, and the primary key values of CustomerID, ArtistID, WorkID, and TransactionID shown in that figure are nonsequential. Yet, the IDENTITY(*m,n*) property that we use to populate SQL Server surrogate primary keys creates sequential numbering.

Amazon Ex. 1008.0396

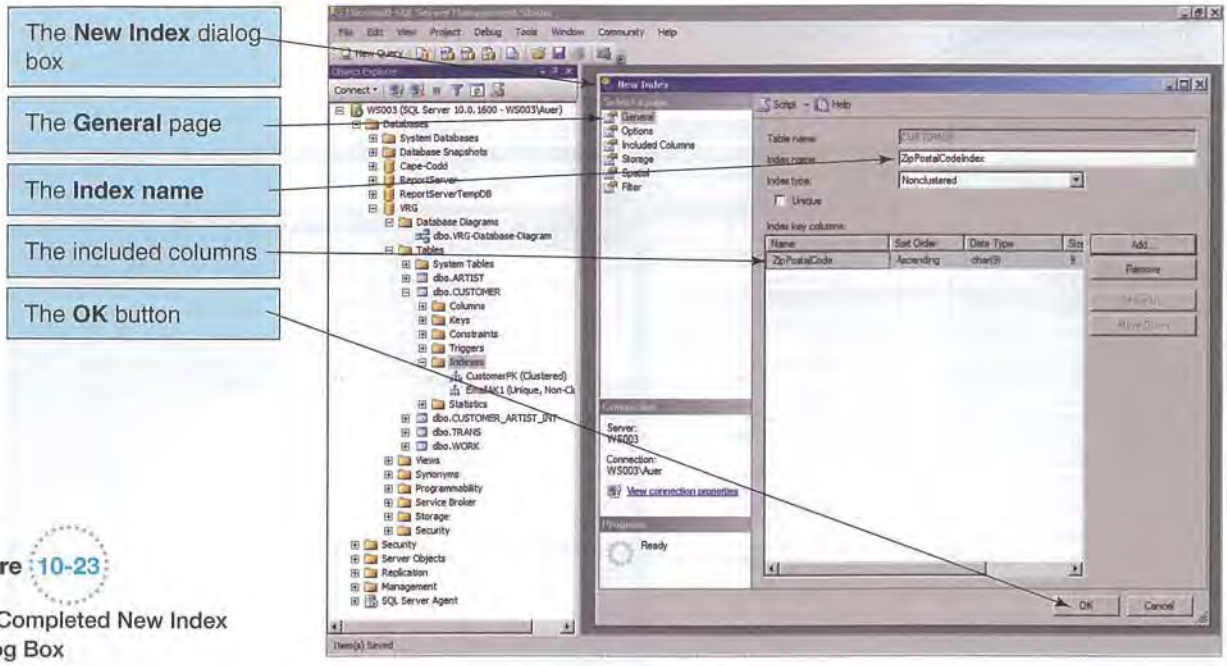


Figure 10-23
The Completed New Index Dialog Box

This means that if we write and execute SQL INSERT statements to put the artist data shown in Figure 7-12(b) into the ARTIST table, the values of ArtistID that will be added to the table will be (1, 2, 3, 4, 5, 6, 7, 8, 9) instead of the values of (1, 2, 3, 4, 5, 11, 17, 18, 19) listed in the figure. How can we enter the needed nonsequential values?

The answer is the **Transact-SQL IDENTITY_INSERT property**. When IDENTITY_INSERT is set to OFF (the default setting), only the SQL Server DBMS can enter data into the controlled ID column in the table. When IDENTITY_INSERT is set to ON, values can be input into the controlled column in the table. However, IDENTITY_INSERT can only be set to ON for only one table at a time. Futher, IDENTITY_INSERT requires the use of a column list containing the name of the surrogate key in each INSERT command.

Thus, instead of using an SQL INSERT statement that automatically enters the surrogate value, such as:

```
INSERT INTO ARTIST VALUES('Miro', 'Joan', 'Spanish', 1893, 1983);
```

we have to use a set of SQL statements similar to the following:

```
SET IDENTITY_INSERT dbo.ARTIST ON
INSERT INTO ARTIST
   (ArtistID, LastName, FirstName, Nationality,
   DateOfBirth, DateDeceased)
   VALUES (1, 'Miro', 'Joan', 'Spanish', 1893, 1983);
SET IDENTITY_INSERT dbo.ARTIST OFF
```

Note how we set INDENTITY_INSERT to ON, insert the date, and then set IDENTITY_INSERT to OFF. Of course this is a lot of work if we are inserting one row of data at a time, but when used in an SQL script that inserts a lot of data into a table it makes sense. So, we will use an SQL script.

The set of SQL INSERT statements needed to populate the VRG database with the View Ridge Gallery data shown in Figure 7-16 is shown in Figure 10-24. Create and save a new SQL script named **VRG-Table-Data.sql** based on Figure 10-24, testing it with the **Parse** command (use the Parse button) until you have corrected any errors. Save the corrected script, and then run the script (use the **Execute** button) to populate the tables. Close the script window after the script has been successfully run.

Amazon Ex. 1008.0397

```
/*****************************************************************************/
/*                                                                           */
/*     Kroenke and Auer - Database Processing (11th Edition) Chapter 10      */
/*                                                                           */
/*     The View Ridge Gallery (VRG) Database - Insert Data                   */
/*                                                                           */
/*     These are the Microsoft SQL Server 2005/2008 SQL code solutions       */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*     This file contains the initial data for each table.                   */
/*                                                                           */
/*     NOTE:  We will have a problem entering the surrogate key values shown in  */
/*     the text.                                                             */
/*                                                                           */
/*     The database table structure was set up using the SQL Server T-SQL    */
/*     keyword IDENTITY to create surrogate keys. This means that by default */
/*     we cannot enter values into any column defined with IDENTITY.         */
/*                                                                           */
/*     But that means we cannot enter the primary key values as shown.       */
/*     To work around this, we will use the T-SQL command IDENTITY_INSERT.   */
/*                                                                           */
/*     When IDENTITY_INSERT is set to OFF (the default), only the DBMS       */
/*     can enter data into the controlled column in the table.               */
/*                                                                           */
/*     When IDENTITY_INSERT is set to ON, values can be input into the       */
/*     controlled column in the table. However, IDENTITY_INSERT can only be set  */
/*     to ON for only one table at a time. Futher, IDENTITY_INSERT requires  */
/*     the use of a column list containing the name of the surrogate key in each */
/*     INSERT command.                                                       */
/*                                                                           */
/*****************************************************************************/

/*     Be sure IDENTITY_INSERT is OFF for all tables.                        */

SET IDENTITY_INSERT dbo.CUSTOMER OFF
SET IDENTITY_INSERT dbo.ARTIST OFF
SET IDENTITY_INSERT dbo.WORK OFF
SET IDENTITY_INSERT dbo.TRANS OFF

/*****************************************************************************/

/*     INSERT data for CUSTOMER                                              */

/*     Set IDENTITY_INSERT to ON for CUSTOMER;                               */
/*     reset it to OFF after CUSTOMER data is inserted.                      */

SET IDENTITY_INSERT dbo.CUSTOMER ON

INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1000, 'Janes', 'Jeffrey', '123 W. Elm St', 'Renton', 'WA', '98055', 'USA',
       '425', '543-2345', 'Jeffrey.Janes@somewhere.com');
```

Figure 10-24

The SQL Statements to Populate the VRG Tables

Amazon Ex. 1008.0398

```
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1001, 'Smith', 'David', '813 Tumbleweed Lane', 'Loveland', 'CO', '81201', 'USA',
       '970', '654-9876', 'David.Smith@somewhere.com');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1015, 'Twilight', 'Tiffany', '88 1st Avenue', 'Langley', 'WA', '98260', 'USA',
        '360', '765-5566', 'Tiffany.Twilight@somewhere.com');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1033, 'Smathers', 'Fred', '10899 88th Ave', 'Bainbridge Island', 'WA', '98110', 'USA'
       '206', '876-9911', 'Fred.Smathers@somewhere.com');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1034,'Frederickson','Mary Beth','25 South Lafayette', 'Denver', 'CO', '80201', 'USA',
       '303', '513-8822', 'MaryBeth.Frederickson@somewhere.com');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1036, 'Warning', 'Selma', '205 Burnaby', 'Vancouver', 'BC', 'V6Z 1W2', 'Canada',
       '604', '988-0512', 'Selma.Warning@somewhere.com');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1037, 'Wu', 'Susan', '105 Locust Ave', 'Atlanta', 'GA', '30322', 'USA',
       '404', '653-3465', 'Susan.Wu@somewhere.com');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1040, 'Gray', 'Donald','55 Bodega Ave', 'Bodega Bay', 'CA', '94923', 'USA',
       '707', '568-4839', 'Donald.Gray@somewhere.com');
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber)
       VALUES (
       1041, 'Johnson', 'Lynda', '117 C Street', 'Washington', 'DC', '20003', 'USA',
INSERT INTO CUSTOMER
       (CustomerID, LastName, FirstName, Street, City, State, ZipPostalCode, Country,
        AreaCode, PhoneNumber, Email)
       VALUES (
       1051, 'Wilkens', 'Chris', '87 Highland Drive', 'Olympia', 'WA', '98508', 'USA',
       '360', '765-7766', 'Chris.Wilkens@somewhere.com');

SET IDENTITY_INSERT dbo.CUSTOMER OFF
```

Figure 10-24

(Continued)

Amazon Ex. 1008.0399

```
/*****************************************************************************/

/*     INSERT data for ARTIST                                                */

/*     Set IDENTITY_INSERT to ON for ARTIST;                                 */
/*     reset it to OFF after ARTIST data is inserted.                        */

SET IDENTITY_INSERT dbo.ARTIST ON

INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       1, 'Miro', 'Joan', 'Spanish', 1893, 1983);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       2, 'Kandinsky', 'Wassily', 'Russian', 1866, 1944);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       3, 'Klee', 'Paul', 'German', 1879, 1940);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       4, 'Matisse', 'Henri', 'French', 1869, 1954);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       5, 'Chagall', 'Marc', 'French', 1887, 1985);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       11, 'Sargent', 'John Singer', 'United States', 1856, 1925);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       17, 'Tobey', 'Mark', 'United States', 1890, 1976);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       18, 'Horiuchi', 'Paul', 'United States', 1906, 1999);
INSERT INTO ARTIST
       (ArtistID, LastName, FirstName, Nationality, DateOfBirth, DateDeceased)
       VALUES (
       19, 'Graves', 'Morris', 'United States', 1920, 2001);

SET IDENTITY_INSERT dbo.ARTIST OFF
```

Figure 10-24

(Continued)

Amazon Ex. 1008.0400

```
/*****************************************************************************/
/*     INSERT data for CUSTOMER_ARTIST_INT                                   */
/*     IDENTITY_INSERT OFF is NOT needed for CUSTOMER_ARTIST_INT -           */
/*     There are NO surrogate keys in CUSTOMER_ARTIST_INT.                   */
INSERT INTO CUSTOMER_ARTIST_INT VALUES (1, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (1, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (2, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (2, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (4, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (4, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1034);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (5, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1001);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (11, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (17, 1051);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (18, 1051);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1000);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1015);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1033);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1036);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1040);
INSERT INTO CUSTOMER_ARTIST_INT VALUES (19, 1051);

/*****************************************************************************/
/*     INSERT data for WORK                                                  */
/*     Set IDENTITY_INSERT to ON for WORK;                                   */
/*     reset it to OFF after WORK data is inserted.                          */
SET IDENTITY_INSERT dbo.WORK ON

INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
      VALUES (
      500, 'Memories IV', 'Unique', 'Casein rice paper collage', '31 x 24.8 in.', 18);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
      VALUES (
      511, 'Surf and Bird', '142/500', 'High Quality Limited Print',
      'Northwest School Expressionist style', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
      VALUES (
      521, 'The Tilled Field', '788/1000', 'High Quality Limited Print',
      'Early Surrealist style', 1);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
      VALUES (
      522, 'La Lecon de Ski', '353/500', 'High Quality Limited Print',
      'Surrealist style', 1);
```

Figure 10-24

(Continued)

Amazon Ex. 1008.0401

```
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       523, 'On White II', '435/500', 'High Quality Limited Print',
       'Bauhaus style of Kandinsky', 2);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       524, 'Woman with a Hat', '596/750', 'High Quality Limited Print',
       'A very colorful Impressionist piece', 4);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       537, 'The Woven World', '17/750', 'Color lithograph', 'Signed', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       548, 'Night Bird', 'Unique', 'Watercolor on Paper',
       '50 x 72.5 cm. - Signed', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       551, 'Der Blaue Reiter', '236/1000', 'High Quality Limited Print',
       'The Blue Rider-Early Pointilism influence', 2);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       552, 'Angelus Novus', '659/750', 'High Quality Limited Print',
       'Bauhaus style of Klee', 3);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       553, 'The Dance', '734/1000', 'High Quality Limited Print',
       'An Impressionist masterpiece', 4);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       554, 'I and the Village', '834/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       555, 'Claude Monet Painting', '684/1000', 'High Quality Limited Print',
       'Shows French Impressionist influence of Monet', 11);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       561, 'Sunflower', 'Unique', 'Watercolor and ink',
       '33.3 x 16.1 cm. - Signed', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       562, 'The Fiddler', '251/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       563, 'Spanish Dancer', '583/750', 'High Quality Limited Print',
       'American realist style - From work in Spain', 11);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       564, 'Farmer''s Market #2',   '267/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       565, 'Farmer''s Market #2',   '268/500', 'High Quality Limited Print',
        'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       566, 'Into Time', '323/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 18);
```

Figure 10-24

(Continued)

Amazon Ex. 1008.0402

```
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       570, 'Untitled Number 1', 'Unique', 'Monotype with tempera',
       '4.3 x 6.1 in. Signed', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       571, 'Yellow Covers Blue', 'Unique', 'Oil and collage',
       '71 x 78 in. - Signed', 18);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       578, 'Mid-Century Hibernation', '362/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       580, 'Forms in Progress I', 'Unique', 'Color aquatint',
       '19.3 x 24.4 in. - Signed', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       581, 'Forms in Progress II', 'Unique', 'Color aquatint',
       '19.3 x 24.4 in. - Signed', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       585, 'The Fiddler', '252/1000', 'High Quality Limited Print',
       'Shows Belarusian folk-life themes and symbology', 5);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       586, 'Spanish Dancer', '588/750', 'High Quality Limited Print',
       'American Realist style - From work in Spain', 11);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       587, 'Broadway Boggie', '433/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       588, 'Universal Field', '114/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       589, 'Color Floating in Time', '487/500', 'High Quality Limited Print',
       'Northwest School Abstract Expressionist style', 18);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       590, 'Blue Interior', 'Unique', 'Tempera on card', '43.9 x 28 in.', 17);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       593, 'Surf and Bird', 'Unique', 'Gouache', '26.5 x 29.75 in. - Signed', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       594, 'Surf and Bird', '362/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
```

Figure 10-24
(Continued)

Amazon Ex. 1008.0403

```
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       595, 'Surf and Bird', '365/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);
INSERT INTO WORK (WorkID, Title, Copy, Medium, Description, ArtistID)
       VALUES (
       596, 'Surf and Bird', '366/500', 'High Quality Limited Print',
       'Northwest School Expressionist style', 19);

SET IDENTITY_INSERT dbo.WORK OFF

/*****************************************************************************/

/*     INSERT data for TRANS                                                 */

/*     Set IDENTITY_INSERT to ON for TRANS;                                  */
/*     reset it to OFF after TRANS data is inserted.                         */

SET IDENTITY_INSERT dbo.TRANS ON

INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       100, '11/4/2005', 30000.00, 45000.00, '12/14/2005', 42500.00, 1000, 500);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       101, '11/7/2005', 250.00, 500.00, '12/19/2005', 500.00, 1015, 511);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       102, '11/17/2005', 125.00, 250.00, '01/18/2006', 200.00, 1001, 521);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       103, '11/17/2005', 250.00, 500.00, '12/12/2006', 400.00, 1034, 522);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       104, '11/17/2005', 250.00, 250.00, '01/18/2006', 200.00, 1001, 523);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       105, '11/17/2005', 200.00, 500.00, '12/12/2006', 400.00, 1034, 524);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       115, '03/03/2006', 1500.00, 3000.00, '06/07/2006', 2750.00, 1033, 537);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       121, '09/21/2006', 15000.00, 30000.00, '11/28/2006', 27500.00, 1015, 548);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       125, '11/21/2006', 125.00, 250.00, '12/18/2006', 200.00, 1001, 551);
```

Figure 10-24

(Continued)

Amazon Ex. 1008.0404

```
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, WorkID)
      VALUES (
      126, '11/21/2006', 200.00, 400.00, 552)
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      127, '11/21/2006', 125.00, 500.00, '12/22/2006', 400.00, 1034, 553);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      128, '11/21/2006', 125.00, 250.00, '03/16/2007', 225.00, 1036, 554);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      129, '11/21/2006', 125.00, 250.00, '03/16/2007', 225.00, 1036, 555);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      151, '05/7/2007', 10000.00, 20000.00, '06/28/2007', 17500.00, 1036, 561);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      152, '05/18/2007', 125.00, 250.00, '08/15/2007', 225.00, 1001, 562);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      153, '05/18/2007', 200.00, 400.00, '08/15/2007', 350.00, 1001, 563);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      154, '05/18/2007', 250.00, 500.00, '09/28/2007', 400.00, 1040, 564);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, WorkID)
      VALUES (
      155, '05/18/2007', 250.00, 500.00, 565);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      156, '05/18/2007', 250.00, 500.00, '09/27/2007', 400.00, 1040, 566);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      161, '06/28/2007', 7500.00, 15000.00, '09/29/2007', 13750.00, 1033, 570);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      171, '08/23/2007', 35000.00, 60000.00, '09/29/2007', 55000.00, 1000, 571);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
      VALUES (
      175, '09/29/2007', 40000.00, 75000.00, '12/18/2007', 72500.00, 1036, 500);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
      AskingPrice, WorkID)
      VALUES (
      181, '10/11/2007', 250.00, 500.00, 578);
```

Figure 10-24

(Continued)

Amazon Ex. 1008.0405

```
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       201, '02/28/2008', 2000.00, 3500.00, '04/26/2008', 3250.00, 1040, 580);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       202, '02/28/2008', 2000.00, 3500.00, '04/26/2008', 3250.00, 1040, 581);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       225, '06/8/2008', 125.00, 250.00, '09/27/2008', 225.00, 1051, 585);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       226, '06/8/2008', 200.00, 400.00, 586);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       227, '06/8/2008', 250.00, 500.00, '09/27/2008', 475.00, 1051, 587);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       228, '06/8/2008', 250.00, 500.00, 588)
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       229, '06/8/2008', 250.00, 500.00, 589)
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, DateSold, SalesPrice, CustomerID, WorkID)
       VALUES (
       241, '08/29/2008', 2500.00, 5000.00, '09/27/2008', 4750.00, 1015, 590)
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       251, '10/25/2008', 25000.00, 50000.00, 593);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       252, '10/27/2008', 250.00, 500.00, 594);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       253, '10/27/2008', 250.00, 500.00, 595);
INSERT INTO TRANS (TransactionID, DateAcquired, AcquisitionPrice,
       AskingPrice, WorkID)
       VALUES (
       254, '10/27/2008', 250.00, 500.00, 596);

SET IDENTITY_INSERT dbo.TRANS OFF

/*****************************************************************************
```

Figure 10-24

(Continued)

Creating Views

SQL views were discussed in Chapter 7. One view we created there was *CustomerInterestsView*. In SQL Server 2008, views can be created in the Microsoft SQL Server Management Studio by using either an SQL statement (as we have done to create and populate the VRG tables) or by using the GUI Display (by right-clicking the Views folder to display a shortcut menu and

Amazon Ex. 1008.0406

then clicking the New View command.) *CustomerInterestsView* can be created with the SQL statement:

```
CREATE VIEW CustomerInterestsView AS
   SELECT   C.LastName AS CustomerLastName,
            C.FirstName AS CustomerFirstName,
            A.LastName AS ArtistName
   FROM     CUSTOMER AS C JOIN CUSTOMER_ARTIST_INT AS CI
      ON    C.CustomerID = CI.CustomerID
            JOIN   ARTIST AS A
               ON  CI.ArtistID = A.ArtistID;
```

Figure 10-25 shows this SQL CREATE VIEW statement in an SQL script named VRG-Create-Views.sql in the Microsoft SQL Server Management Studio. We'll step through creating and testing this view.

Creating a New View

1. In the Microsoft SQL Server Management Studio Object Explorer, click the **New Query** button to open a new tabbed SQL document window.
2. Click the **Intellisense Enabled** button to disable the Intellisense feature.
3. Type the SQL statements, but not the comments, shown in Figure 10-25.
4. Click the **Parse** button. If there are any SQL coding errors detected, fix them.
5. Click the **Execute** button.
6. Expand the **Views** folder in the VRG database object. Note that the dbo.CustomerInterestsView object has been added to the VRG database, as shown in Figure 11-25.
7. To save this CREATE VIEW statement as part of an SQL script, add the comments shown in Figure 11-25 and then save the script as **VRG-Create-Views.sql**.
8. Click the document window **Close** button to close the window.

We can now look at the view in a GUI display.

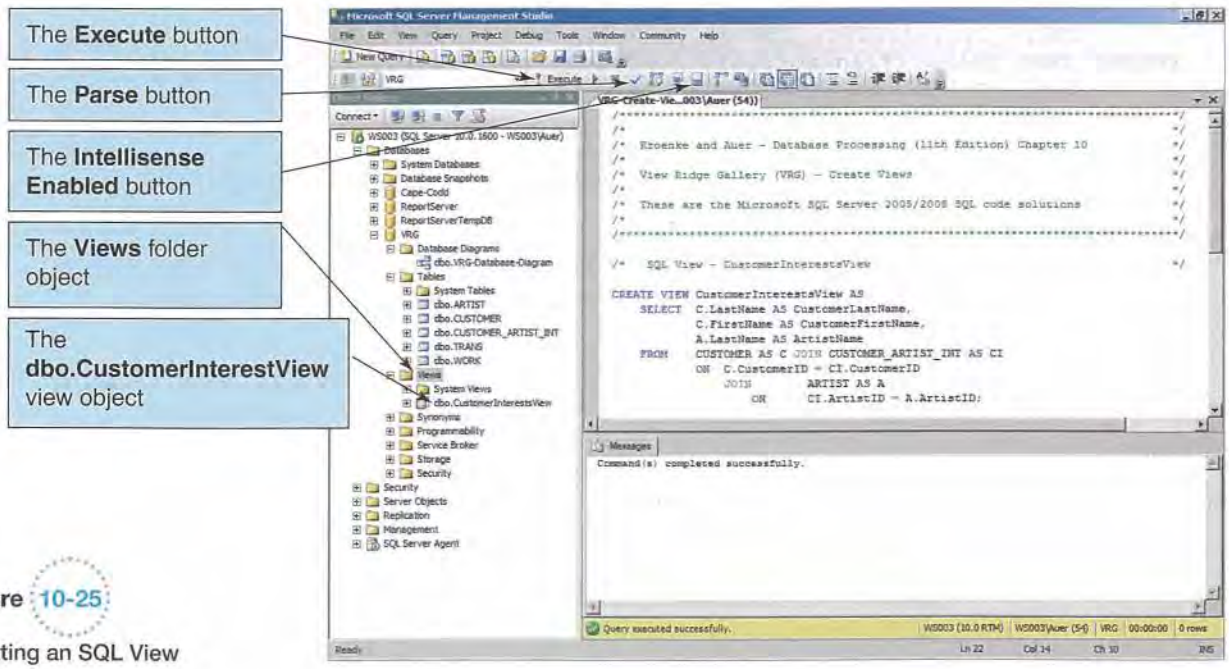
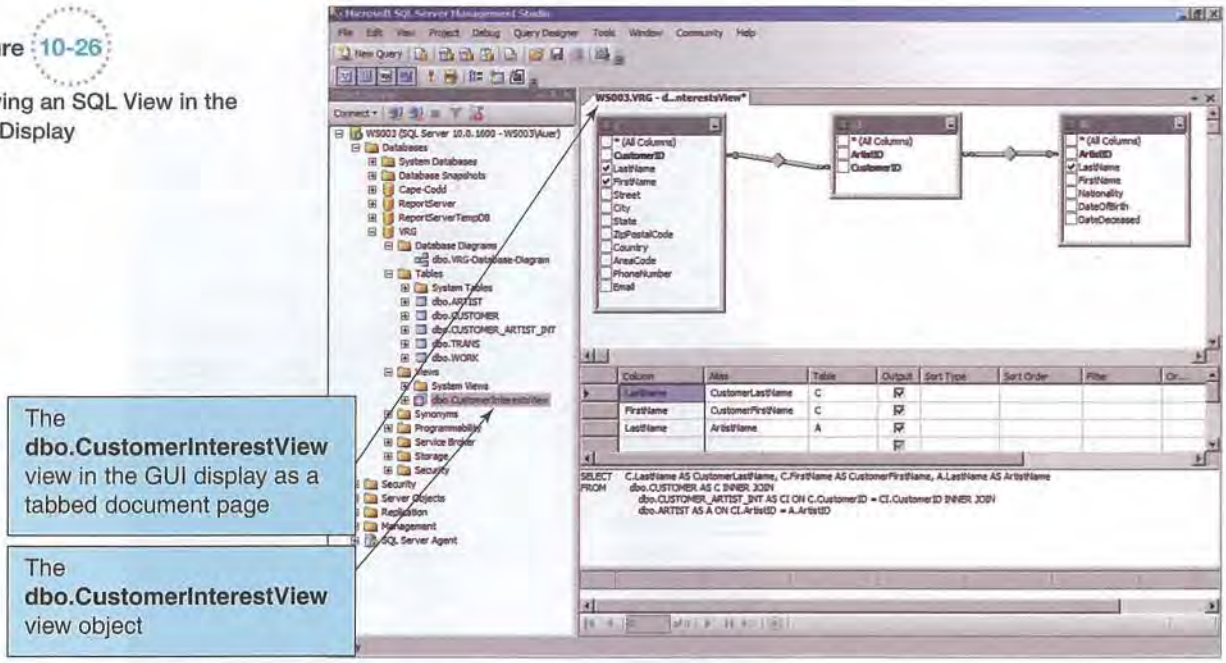


Figure 10-25
Creating an SQL View

Amazon Ex. 1008.0407

Figure 10-26

Viewing an SQL View in the GUI Display



Viewing an Existing View in the SQL Server GUI Display

1. Right-click the **dbo.CustomerInterests** view object in the Object Browser to display a shortcut menu.
2. Click the **Design** command on the shortcut menu. The dbo.CustomerInterestView is displayed in a new tabbed document window in the GUI display format.
3. Rearrange the components of the view in the GUI display so that it appears as shown in Figure 10-26.
4. Click the **Save** button to save the reconfigured view GUI display.
5. Click the document window **Close** button to close the GUI display window.
6. Collapse the **Views** folder in the Object Browser.

As explained in Chapter 7, SQL views are used like tables in other SQL statements. For example, to see all the data in the view, we use the SQL SELECT statement:

```
SELECT        *
FROM          CustomerInterestsView
ORDER BY      CustomerLastName, CustomerFirstName;
```

The result is shown in Figure 10-27.

SQL Server Application Logic

An SQL Server database can be processed from an application in a number of different ways. The first is one that we are already familiar with—SQL scripts. For security, however, such files should only be used during application development and testing and never on an operational database.

Another way is to create application code using a Microsoft .NET language such as C#.NET, C++.NET, VB.NET, or some other programming language and then invoke SQL Server DBMS commands from those programs. The modern way to do this is to use a library of object classes, create objects that accomplish database work, and then process those objects by setting object properties and invoking object methods.

Amazon Ex. 1008.0408

	CustomerLastName	CustomerFirstName	ArtistName
1	Frederickson	Mary Beth	Chagall
2	Frederickson	Mary Beth	Kandinsky
3	Frederickson	Mary Beth	Miro
4	Frederickson	Mary Beth	Matisse
5	Gray	Donald	Tobey
6	Gray	Donald	Horiuchi
7	Gray	Donald	Graves
8	Janes	Jeffrey	Graves
9	Janes	Jeffrey	Horiuchi
10	Janes	Jeffrey	Tobey
11	Smathers	Fred	Tobey
12	Smathers	Fred	Horiuchi
13	Smathers	Fred	Graves
14	Smith	David	Chagall
15	Smith	David	Matisse
16	Smith	David	Kandinsky
17	Smith	David	Miro
18	Smith	David	Sargent
19	Twilight	Tiffany	Sargent
20	Twilight	Tiffany	Tobey
21	Twilight	Tiffany	Horiuchi
22	Twilight	Tiffany	Graves
23	Warning	Selma	Chagall
24	Warning	Selma	Graves
25	Warning	Selma	Sargent
26	Wilkens	Chris	Tobey
27	Wilkens	Chris	Graves
28	Wilkens	Chris	Horiuchi

Figure 10-27

Result of Using the View CustomerInterestView

Application logic can be embedded in triggers. As you learned in Chapter 7, triggers can be used to set default values, to enforce data constraints, to update views, and to enforce referential integrity constraints. In this chapter, we will describe four triggers, one for each of the four trigger uses. These triggers will be invoked by SQL Server when the specified actions occur.

Finally, another way of processing an SQL Server database is to create stored procedures, as described in Chapter 7. These stored procedures can then be invoked from application programs or from Web pages using languages such as VBScript or PHP. Stored procedures can also be executed from the SQL Server Management Studio. This should be done only when the procedures are being developed and tested, however. As described in Chapter 9, for security reasons, no one other than authorized members of the DBA staff should be allowed to interactively process an operational database.

In this chapter, we will describe and illustrate two stored procedures. Here, we will test those procedures by invoking them from the SQL Server Management Studio, and some of our output will be designed specifically for this environment. Again, this should be done only during development and testing. You will learn how to invoke these stored procedures from application code in Chapter 11.

Transact-SQL

Transact-SQL (T-SQL) is Microsoft's variant of SQL with additional procedural language added for use in stored procedures and triggers. We will use certain elements of Transact-SQL in such code,

Amazon Ex. 1008.0409

and therefore we need to discuss them at this point. Information on these and other Transact-SQL language components can be found at the Microsoft SQL Server Language Reference at *http://msdn.microsoft.com/en-us/library/ms166026(SQL.90).aspx*.

Transact-SQL identifies variables and parameters with an @ symbol. Thus, WorkID is a column name, but @WorkID is a Transact-SQL variable or parameter. A **parameter** is a value that is passed to a stored procedure when it is called. A **variable** is a value used within a stored procedure or trigger itself. Comments in Transact-SQL are either enclosed in /* and */ signs or follow **two dashes (--)** if they are restricted to one line.



Transact-SQL statements in procedures and triggers can be written with or without the ending semicolon (;) required in nonprocedural SQL statements. For consistency, we will continue to use semicolons for SQL statements, but we will not use them for some of the other procedural elements discussed in the following sections.

Transact-SQL Control-of-flow Statements

The **Transact-SQL control-of-flow language** contains procedural language components that let you control exactly which parts of your code are used and the conditions required for their use. These components include IF . . . ELSE, BEGIN . . . END, WHILE, RETURN, and other keywords that can be used to direct the operations of a block of code. The **IF . . . ELSE keywords** are used to test for a condition, and then direct which blocks of code are used based on the result of that test. Note that the END keyword is *not* used as part of this construct in Transact-SQL, although it is commonly used here in many other programming languages. The **BEGIN . . END keywords** define a block of Transact-SQL statements so that more than one statement can be executed. Without this grouping, only one SQL statement can be used in either the IF or ELSE section of an IF . . . ELSE conditional branching. The **WHILE keyword** is used to create loops in Transact-SQL, where a section of code is repeated as long as ("while") some condition is true. The **RETURN keyword** is used to exit a block of code and terminate whatever code structure (stored procedure or trigger) is running. In this case, control is "returned" to the DBMS.

As an example, let's consider a new customer at the View Ridge Gallery who needs to have customer data entered into the CUSTOMER table and artist interest data entered into the CUSTOMER_ARTIST_INT table. The new customer is Michael Bench, with phone number 206-876-8822, e-mail address Michael.Bench@somewhere.com, and an interest in French artists.

Before we enter Michael's data, we need to check to see whether he is already in the database. To do this, we can use the following Transact-SQL code:

```
DECLARE    @RowCount    AS    Int
-- Check to see if Customer already exists in database
SELECT     @RowCount = COUNT(*)
FROM       CUSTOMER
WHERE      LastName = 'Bench'
     AND   FirstName = 'Michael'
     AND   AreaCode = '206'
     AND   PhoneNumber = '876-8822'
     AND   Email = 'Michael.Bench@somewhere.com';
-- IF @RowCount > 0 THEN Customer already exists.
IF (@RowCount > 0)
     BEGIN
        PRINT ' The Customer is already in the database. '
        RETURN
     END
```

Amazon Ex. 1008.0410

```
-- IF (@RowCount = 0) THEN Customer does not exist in database.
ELSE
    BEGIN
    -- Insert new Customer data.
       INSERT INTO dbo.CUSTOMER
          (LastName, FirstName, AreaCode, PhoneNumber,
          Email)
          VALUES('Bench', 'Michael', '206', '876-8822',
          'Michael.Bench@somewhere.com');
          PRINT ' The new Customer is now in the database. '
    END
```

This block of SQL code illustrates the use of all of the control-of-flow keywords we have discussed except WHILE. The WHILE keyword is used in code loops, and one use of a code loop is in an SQL cursor.

Transact-SQL Cursor Statements

As we discussed in Chapter 7, a cursor is used so that SQL results stored in a table can be processed one row at a time. The **Transact-SQL cursor** is Microsoft's implementation of a cursor. Related cursor keywords include DECLARE, OPEN, FETCH, CLOSE, and DEALLOCATE. The **DECLARE CURSOR keywords** are used to create a cursor, whereas the **OPEN CURSOR keywords** start the use of the cursor. The **FETCH keyword** is used to retrieve row data. The **CLOSE CURSOR keywords** are used to exit the use of a cursor, and the **DEALLOCATE CURSOR keywords** remove the cursor from the DBMS. When using a cursor, the WHILE keyword is used to control how long the cursor is active.

Let's consider Michael Bench's interest in French artists. The ARTIST table has two French artists—Henri Matisse and Marc Chagall. Therefore, we need to add new rows to CUSTOMER_ARTIST_INT, both of which will contain Michael's CustomerID number (now that he has one) and each of which contain the ArtistID for one of these artists. To do this, we can use the following Transact-SQL code:

```
DECLARE       @ArtistID     AS     Int
DECLARE       @CustomerID   AS     Int
-- GET the CustomerID surrogate key value.
SELECT        @CustomerID = CustomerID
FROM          CUSTOMER
WHERE         LastName = 'Bench'
        AND   FirstName = 'Michael'
        AND   AreaCode = '206'
        AND   PhoneNumber = '876-8822'
        AND   Email = 'Michael.Bench@somewhere.com';
-- Create intersection record for each appropriate Artist.
-- Create cursor ArtistCursor
DECLARE ArtistCursor CURSOR FOR
          SELECT     ArtistID
          FROM       ARTIST
          WHERE      Nationality = 'French';
-- Process each appropriate Artist
OPEN ArtistCursor
    FETCH NEXT FROM ArtistCursor INTO @ArtistID
       WHILE @@FETCH_STATUS = 0
```

Amazon Ex. 1008.0411

```
        BEGIN
          INSERT INTO CUSTOMER_ARTIST_INT
          (ArtistID, CustomerID)
          VALUES(@ArtistID, @CustomerID)
        PRINT ' New CUSTOMER_ARTIST_INT row added. '
        PRINT ' ArtistID = '+CONVERT(Char(6), @ArtistID)
        PRINT ' CustomerID = '+CONVERT(Char(6), @CustomerID)
          FETCH NEXT FROM ArtistCursor INTO @ArtistID
        END
CLOSE ArtistCursor
-- Remove cursor ArtistCursor
DEALLOCATE ArtistCursor
```

In this code, the ArtistCursor loops through the set of ArtistID values for French artists as long as ("while") the value of @@FETCH_STATUS is equal to 0. The **Transact-SQL @@FETCH_STATUS function** returns a value based on whether the FETCH NEXT statement actually returned a value from the next row. If there is no next row, then @@FETCH_STATUS returns 0.

Transact-SQL Output Statements

The two previous code segments also illustrate the use of the **Transact-SQL PRINT command**. This command will send a message to the SQL Server Management Console Messages window. We will use this output as a proxy for the real output that would be returned to an application, as it would be in most situations. Note that the actual output to be printed is marked by a single quote at the beginning and the end. Also note the use of the CONVERT function. The **Transact-SQL CONVERT function** is used to change one data type to another. In this case, we can only print character strings, so numbers such as ArtistID must be converted to character strings using the CONVERT function. Character strings can be combined by using the plus sign (+), and this is also shown in the code segments above.

Stored Procedures

With SQL Server 2000, stored procedures must be written in Transact-SQL. Starting with the release of SQL Server 2005, stored procedures and triggers can be written in any of the .NET or Common Language Runtime languages such as Visual Basic .NET, Visual C# .NET, and Visual C++ .NET. Transact-SQL continues to be supported and will have improved features and functions for error handling as well as other language upgrades. The Transact-SQL shown here will operate in SQL Server 2005 and SQL 2008, but in the long run, however, Visual Basic .NET, Visual C# .NET, or Visual C++ .NET will become better choices for stored procedures and triggers.

As with other database structures, you can write a stored procedure in an SQL script text file and process the commands using the SQL Server Management Studio. However, there is one little gotcha. The first time you create a stored procedure in an SQL script, you use the **SQL CREATE PROCEDURE statement**. Subsequently, if you change the procedure, use the **SQL ALTER PROCEDURE statement**. Otherwise, you will get an error message saying that the procedure already exists when you execute the modified procedure code.

You can also create a stored procedure within the SQL Server Management Studio by Enterprise Manager by expanding a database's Programmability folder object, right-clicking Stored Procedures, and selecting New Stored Procedure. If you do this, however, SQL Server simply opens an editable draft template in a tabbed document window.

The Stored Procedure InsertCustomerAndInterests

In our preceding discussion of Transact-SQL, we used as our example the need to enter data for a new customer and the artists of interest to that customer. The code segments we wrote were very specifically tied to the data we used, and thus of limited use. Is there a way to write a general block of code that could be used for more than one customer? Yes, and that block of code is a stored procedure.

Figure 10-28 shows the SQL code for the InsertCustomerAndInterests stored procedure. This stored procedure generalizes our previous code and can be used to insert data for any new

Amazon Ex. 1008.0412

customer into CUSTOMER, and then store data for that customer in CUSTOMER_ARTIST_INT, linking the customer to all artists having a particular nationality.

Five parameters are input to the procedure: @NewLastName, @NewFirstName, @NewAreaCode, @NewPhoneNumber, and @Nationality. The first four parameters are the new customer data, and the fourth one is the nationality of the artists for which the new customer has an interest. The stored procedure also uses three variables: @RowCount, @ArtistID, and @CustomerID.

Figure 10-28

The SQL Statements for the InsertCustomerAndInterests Stored Procedure

```
CREATE PROCEDURE InsertCustomerAndInterests
          @NewLastName      Char(25),
          @NewFirstName     Char(25),
          @NewAreaCode      Char(3),
          @NewPhoneNumber   Char(8),
          @NewEmail         Varchar(100),
          @Nationality      Char(30)
AS
   DECLARE   @RowCount     AS Int
   DECLARE   @ArtistID     AS Int
   DECLARE   @CustomerID   AS Int

   -- Check to see if Customer already exists in database

   SELECT @RowCount = COUNT(*)
   FROM   dbo.CUSTOMER
   WHERE  LastName = @NewLastName
      AND FirstName = @NewFirstName
      AND AreaCode = @NewAreaCode
      AND PhoneNumber = @NewPhoneNumber
      AND Email =   @NewEmail;

   -- IF @RowCount > 0 THEN Customer already exists.
   IF (@RowCount > 0)
      BEGIN
         PRINT '*******************************************************'
         PRINT ''
         PRINT '   The Customer is already in the database. '
         PRINT ''
         PRINT '   Customer Last Name         =  '+@NewLastName
         PRINT '   Customer First Name        =  '+@NewFirstName
         PRINT ''
         PRINT '*******************************************************'
         RETURN
      END
   -- IF @RowCount = 0 THEN Customer does not exist in database.
   ELSE
      BEGIN
      -- Insert new Customer data.
         INSERT INTO dbo.CUSTOMER
            (LastName, FirstName, AreaCode, PhoneNumber, Email)
            VALUES(
            @NewLastName, @NewFirstName, @NewAreaCode,
            @NewPhoneNumber, @NewEmail);
         -- Get new CustomerID surrogate key value.
         SELECT @CustomerID = CustomerID
         FROM   dbo.CUSTOMER
         WHERE  LastName = @NewLastName
            AND FirstName = @NewFirstName
            AND AreaCode = @NewAreaCode
            AND PhoneNumber = @NewPhoneNumber
            AND Email =   @NewEmail;
```

Amazon Ex. 1008.0413

```
                PRINT '*****************************************************'
                PRINT ''
                PRINT '   The new Customer is now in the database. '
                PRINT ''
                PRINT '   Customer Last Name          =  '+@NewLastName
                PRINT '   Customer First Name         =  '+@NewFirstName
                PRINT ''
                PRINT '*****************************************************'
    -- Create intersection record for each appropriate Artist.
    DECLARE ArtistCursor CURSOR FOR
          SELECT ArtistID
          FROM   dbo.ARTIST
          WHERE  Nationality=@Nationality
    --Process each appropriate Artist
    OPEN   ArtistCursor
       FETCH NEXT FROM ArtistCursor INTO @ArtistID
             WHILE @@FETCH_STATUS = 0
             BEGIN
                INSERT INTO dbo.CUSTOMER_ARTIST_INT
                   (ArtistID, CustomerID)
                   VALUES(@ArtistID, @CustomerID);
                   PRINT '************************************************'
                   PRINT ''
                   PRINT '   New CUSTOMER_ARTIST_INT row added. '
                   PRINT ''
                   PRINT '   ArtistID    =  '+CONVERT(Char(6), @ArtistID)
                   PRINT '   CustomerID  =  '+CONVERT(Char(6), @CustomerID)
                   PRINT ''
                   PRINT '************************************************'
                FETCH NEXT FROM ArtistCursor INTO @ArtistID
             END
    CLOSE  ArtistCursor
    DEALLOCATE   ArtistCursor
END
```

Figure 10-28 (Continued)

These variables are used to store values of the number of the row, the value of the ArtistID primary key, and the value of the CustomerID primary key, respectively.

The first task performed by this stored procedure is to determine whether the customer already exists. If the value of @RowCount in the first SELECT statement is greater than zero, then a row for that customer already exists. In this case, nothing is done, and the stored procedure prints an error message and exits (using the RETURN command). As stated earlier, the error message is visible in the Microsoft SQL Server Management Studio, but it generally would not be visible to application programs that invoked this procedure. Instead, a parameter or other facility needs to be used to return the error message back to the user via the application program. Discussion of that topic is beyond the scope of the present discussion, but we will send message back to the Microsoft SQL Server Management Studio to mimic such actions and provide a means to make sure our stored procedures are working correctly.

If the customer does not already exist, the procedure inserts the new data into the table dbo.CUSTOMER and then a new value of CustomerID is read into the variable @CustomerID. Internally, SQL Server adds a prefix to the table name that shows the name of the user who created it. Here, the prefix dbo is used to ensure that the CUSTOMER table created by the database owner (dbo) is processed. Without the dbo prefix, if the user invoking the stored procedure had created a table named CUSTOMER, then the user's table and not the dbo's table would be used.

The purpose of the second SELECT in Figure 10-28 is to obtain the value of the surrogate key CustomerID that was created by the INSERT statement. Another option is to use the **SQL Server function @@Identity**, which provides the value of the most recently created surrogate key value. Using this function, you could replace the second SELECT statement with the expression:

```
SET @CustomerID = @@Identity
```

Amazon Ex. 1008.0414

To create the appropriate intersection table rows, an SQL cursor named ArtistCursor is created on an SQL statement that obtains all ARTIST rows where Nationality equals the parameter @Nationality. The cursor is opened and positioned on the first row by calling FETCH NEXT, and then the cursor is processed in a WHILE loop. In this loop, statements between BEGIN and END are iterated until SQL Server signals the end of the data by setting the value of the SQL Server function @@FETCH_STATUS to zero. At each iteration of the WHILE loop, a new row is inserted into the intersection table CUSTOMER_ ARTIST_INT. The FETCH NEXT statement at the end of the block moves the cursor to the next row.

To create the InsertCustomerAndInterests stored procedure in the VRG database, create a new SQL script named VRG-Create-Stored-Procedures.sql containing the SQL code in Figure 10-28. Include a beginning comments section similar to the one shown in Figure 10-10. Use the Parse button to check your SQL code, and after it parses correctly save the completed code before running it. Finally, check to make sure that the VRG database is selected in the Available Databases list, and then use the Execute button to create the stored procedure.

To invoke the InsertCustomerAndInterests stored procedure for Michael Bench, we use the following SQL statement:

```
EXEC InsertCustomerAndInterests
     @NewLastName = 'Bench', @NewFirstName = 'Michael',
     @NewAreaCode = '206', @NewPhoneNumber = '876-8822',
     @NewEmail = 'Michael.Bench@somewhere.com',
     @Nationality = 'French';
```

BY THE WAY Before we test any new functionality of a database such as a stored procedure or a trigger, it is always a good idea to refresh content in database in the Microsoft SQL Server Management Studio. This can be done by using the Refresh command in the object shortcut menu. For example, after creating the InsertCustomerAndInterests stored procedure, right-click the VRG Stored Procedures folder in the Programmability folder and then click the Refresh command. Then expand the Stored Procedures folder and make sure the stored procedure object (shown as dbo.InsertCustomerWithInterests) is visible before running any test data.

Figure 10-29 shows the execution of the stored procedure in the SQL Server Management Studio. Notice how our sections of PRINT commands have produced the necessary output so that we can see what actions were taken. If we now wanted to check the tables themselves, we could do so, but that is not necessary at this point. In the output in Figure 10-29, we see that customer Michael Bench has been added to the CUSTOMER table and that new rows (a total of two, although we would have to scroll through the output to see that there are no more than two) have been inserted into the CUSTOMER_ARTIST_INT table.

BY THE WAY You can include the preceding EXEC InsertCustomerAndInterests statement in your VRG-Create-Stored-Procedures.sql script and run it from there. When there are multiple SQL statements in an open SQL script file, you can still run them one at a time by highlighting the SQL statement that you want to use. The Microsoft SQL Server Management Studio will then only apply actions such as parsing (using the Parse button) and executing (using the Execute button) to the highlighted SQL statement.

This is a good trick to know, because it allows you to store multiple SQL statements (for example, a set of queries) in one SQL script for convenience but still control which statements are actually executed. Even better, note that if you highlight more than one SQL statement then the highlighted *set* of commands can be controlled the same way.

Amazon Ex. 1008.0415

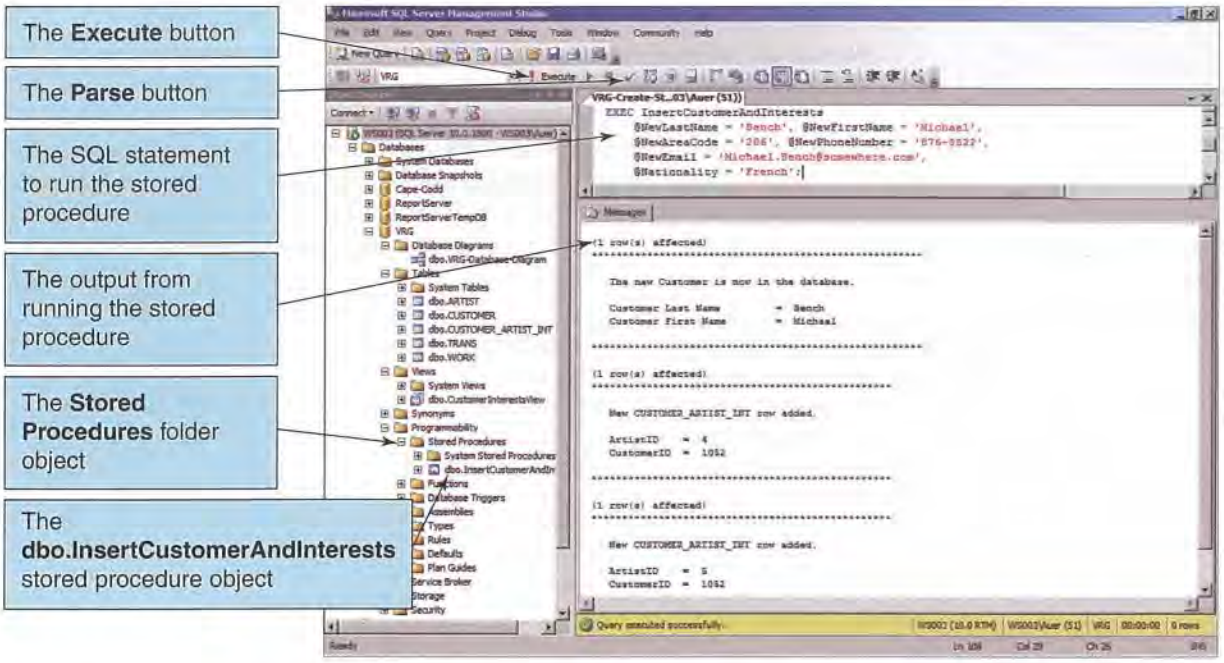


Figure 10-29

Running InsertCustomerAndInterests Stored Procedure

The Stored Procedure InsertCustomerWithTransaction

Now we will write a stored procedure that inserts data for a new customer, records a purchase, and creates an entry in the CUSTOMER_ARTIST_INT table. We will name this stored procedure *InsertCustomerWithTransaction,* and the necessary SQL code is shown in Figure 10-30. This procedure receives seven parameters having data about the new customer and about the customer's purchase. We will use this procedure to discuss transaction processing in SQL Server 2008.

The first action is to see whether the customer already exists. If so, the procedure exits with an error message. If the customer does not exist, this procedure then starts a transaction with the **Transact-SQL BEGIN TRANSACTION command**. Recall from Chapter 9 that transactions ensure that all of the database activity is committed atomically; either all of the updates occur or none of them do. The transaction begins, and the new customer row is inserted. The new value of CustomerID is obtained, as shown in the InsertCustomerWithInterests stored procedure. Next, the procedure checks to determine whether ArtistID, WorkID, and TransactionID are valid. If any are invalid, the transaction is rolled back using the **Transact-SQL ROLLBACK TRANSACTION command**.

If all the surrogate key values are valid, two actions in the transaction are completed. First, an UPDATE statement updates DateSold, SalesPrice, and CustomerID in the appropriate TRANS row. DateSold is set to system date via the **Transact-SQL Getdate( ) function**, SalesPrice is set to the value of @TransSalesPrice, and CustomerID is set to the value of @CustomerID. Second, a row is added to CUSTOMER_ARTIST_INT to record the customer's interest in this artist.

If everything proceeds normally to this point, the transaction is committed using the **Transact-SQL COMMIT TRANSACTION command**. After, and only after, the transaction is committed, we print the results messages.

To create the InsertCustomerWithTransaction stored procedure in the VRG database, add the SQL code in Figure 10-30 to your VRG-Create-Stored-Procedures.sql script. Include a comment to separate the code sections in this script file. Use the highlighting technique described in the preceding by the Way section to parse and execute the SQL code.

To use the InsertCustomerWithTransaction stored procedure, we will record the following purchase by our next new customer, Melinda Gliddens, who just bought a print of John Singer Sargent's *Spanish Dancer* for $350.00. The SQL statement is:

```
EXEC InsertCustomerWithTransaction
     @NewCustomerLastName = 'Gliddens',
     @NewCustomerFirstName = 'Melinda',
     @NewCustomerAreaCode = '360',
     @NewCustomerPhoneNumber = '765-8877',
```

Amazon Ex. 1008.0416

```
    @NewCustomerEmail = 'Melinda.Gliddens@somewhere.com',
    @ArtistLastName = 'Sargent', @WorkTitle = 'Spanish Dancer',
    @WorkCopy = '588/750', @TransSalesPrice = 350.00;
```

Figure 10-30

The SQL Statements for the InsertCustomer WithTransaction Stored Procedure

To execute this EXEC InsertCustomerWithTransaction statement, add the SQL statement to your VRG-Create-Stored-Procedures.sql script. Include a comment to separate the new SQL statement. Use the highlighting technique described in the preceding By the Way section to parse and execute the SQL statement. Figure 10-31 shows the invocation of this procedure using sample data.

```
CREATE PROCEDURE InsertCustomerWithTransaction
            @NewCustomerLastName        Char(25),
            @NewCustomerFirstName       Char(25),
            @NewCustomerAreaCode        Char(3),
            @NewCustomerPhoneNumber     Char(8),
            @NewCustomerEmail           Varchar(100),
            @ArtistLastName             Char(25),
            @WorkTitle                  Char(35),
            @WorkCopy                   Char(12),
            @TransSalesPrice            Numeric(8,2)

AS
   DECLARE  @RowCount          AS Int,
            @ArtistID          AS Int,
            @CustomerID        AS Int,
            @WorkID            AS Int,
            @TransactionID     AS Int

   -- Check to see if Customer already exists in database

   SELECT   @RowCount = COUNT(*)
   FROM     dbo.CUSTOMER
   WHERE    LastName = @NewCustomerLastName
      AND   FirstName = @NewCustomerFirstName
      AND   AreaCode = @NewCustomerAreaCode
      AND   PhoneNumber = @NewCustomerPhoneNumber
      AND   Email =   @NewCustomerEmail;

   -- IF @RowCount > 0 THEN Customer already exists.
   IF (@RowCount > 0)
      BEGIN
         PRINT '*******************************************************'
         PRINT ''
         PRINT '   The Customer is already in the database. '
         PRINT ''
         PRINT '   Customer Last Name     =  '+@NewCustomerLastName
         PRINT '   Customer First Name    =  '+@NewCustomerFirstName
         PRINT ''
         PRINT '*******************************************************'
         RETURN
      END
   -- IF @RowCount = 0 THEN Customer does not exist in database.
   ELSE
      -- Start transaction - Rollback everything if unable to complete it.
      BEGIN TRANSACTION
         -- Insert new Customer data.
         INSERT INTO dbo.CUSTOMER
            (LastName, FirstName, AreaCode, PhoneNumber, Email)
            VALUES(
            @NewCustomerLastName, @NewCustomerFirstName,
            @NewCustomerAreaCode, @NewCustomerPhoneNumber, @NewCustomerEmail);
         -- Get new CustomerID surrogate key value.
```

Amazon Ex. 1008.0417

```
SELECT    @CustomerID = CustomerID
FROM      dbo.CUSTOMER
WHERE     LastName = @NewCustomerLastName
   AND    FirstName = @NewCustomerFirstName
   AND    AreaCode = @NewCustomerAreaCode
   AND    PhoneNumber = @NewCustomerPhoneNumber
   AND    Email =   @NewCustomerEmail;

-- Get ArtistID surrogate key value, check for validity.
SELECT    @ArtistID = ArtistID
FROM      dbo.ARTIST
WHERE     LastName = @ArtistLastName;

IF @ArtistID IS NULL
   BEGIN
      PRINT '*****************************************************'
      PRINT ''
      PRINT '   Invalid ArtistID '
      PRINT ''
      PRINT '*****************************************************'
      ROLLBACK TRANSACTION
      RETURN
   END

-- Get WorkID surrogate key value, check for validity.
SELECT    @WorkID = WorkID
FROM      dbo.WORK
WHERE     ArtistID = @ArtistID
   AND    Title = @WorkTitle
   AND    Copy = @WorkCopy;

IF @WorkID IS NULL
   BEGIN
      PRINT '*****************************************************'
      PRINT ''
      PRINT '   Invalid WorkID '
      PRINT ''
      PRINT '*****************************************************'
      ROLLBACK TRANSACTION
      RETURN
   END

-- Get TransID surrogate key value, check for validity.
SELECT    @TransactionID = TransactionID
FROM      dbo.TRANS
WHERE     WorkID = @WorkID
   AND    SalesPrice = NULL;

IF @TransactionID IS NULL
   BEGIN
      PRINT '*****************************************************'
      PRINT ''
      PRINT '   Invalid TransactionID '
      PRINT ''
      PRINT '*****************************************************'
      ROLLBACK TRANSACTION
      RETURN
   END
```

Figure 10-30
(Continued)

Amazon Ex. 1008.0418

```
    -- All surrogate key values of OK, complete the transaction
    BEGIN
        -- Update TRANS row
        UPDATE dbo.TRANS
        SET    DateSold = GETDATE(),
               SalesPrice = @TransSalesPrice,
               CustomerID = @CustomerID
        WHERE  TransactionID = @TransactionID;
        -- Create CUSTOMER_ARTIST_INT row
        INSERT INTO dbo.CUSTOMER_ARTIST_INT (CustomerID, ArtistID)
               VALUES(@CustomerID, @ArtistID);
    END
-- Commit the Transaction
COMMIT TRANSACTION
    -- The transaction is completed. Print output
    BEGIN
        -- Print Customer results.
        PRINT '*****************************************************'
        PRINT ''
        PRINT '   The new Customer is now in the database. '
        PRINT ''
        PRINT '   Customer Last Name  =  '+@NewCustomerLastName
        PRINT '   Customer First Name =  '+@NewCustomerFirstName
        PRINT ''
        PRINT '*****************************************************'
        -- Print Transaction result
        PRINT ''
        PRINT '   Transaction complete. '
        PRINT ''
        PRINT '   TransactionID       =  '+CONVERT(Char(6), @TransactionID)
        PRINT '   ArtistID            =  '+CONVERT(Char(6), @ArtistID)
        PRINT '   WorkID              =  '+CONVERT(Char(6), @WorkID)
        PRINT '   Sales Price         =  '+CONVERT(Char(12), @TransSalesPrice)
        PRINT ''
        PRINT '*****************************************************'
        -- Print CUSTOMER_ARTIST_INT update
        PRINT ''
        PRINT '   New CUSTOMER_ARTIST_INT row added. '
        PRINT ''
        PRINT '   ArtistID            =  '+CONVERT(Char(6), @ArtistID)
        PRINT '   CustomerID          =  '+CONVERT(Char(6), @CustomerID)
        PRINT ''
        PRINT '*****************************************************'
    END
-- ***** End of stored procedure InsertCustomerWithTransaction *****
```

Figure 10-30

(Continued)

Amazon Ex. 1008.0419

BY THE WAY If we now look at the SQL code that has actually been stored for these two stored procedures (by right-clicking the stored procedure object and then choosing Modify), we will find that SQL Server has added the following lines before the code we wrote:

```
USE [VRG]
GO
/****** Object: StoredProcedure [ {StoredProcedureName} ]
    Script Date: {Date and Time created or altered} ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
```

The **Transact-SQL USE [{DatabaseName}] command** tells the stored procedure to use the VRG database when it is called. This is the SQL command equivalent of selecting the database name in the Available Databases drop-down list in the Microsoft SQL Server Management Studio.

The **Transact-SQL SET ANSI_NULLS ON command** specifies how SQL server handles comparisons of NULL values using equals (=) and not equals (<>) (for more information, see *http://msdn.microsoft.com/en-us/library/ms188048.aspx*). The **Transact-SQL SET QUOTED_IDENTIFIER ON command** specifies that object identifiers (table names, column names, etc.) can be enclosed in double quotes ("), which allows the use of SQL reserved words as objects names. For example, we could run the SQL statement:

```
SELECT "Select"
FROM "FROM"
WHERE "Where" = 'San Francisco';
```

Note that literals (San Francisco) are still enclosed in single quotes. For more information, see *http://msdn.microsoft.com/en-us/library/ms174393.aspx*.

Finally, the **GO command** is not a Transact-SQL statement, but rather a command used by the Microsoft SQL Server Management Studio (and other SQL utilities) to mark the end of a batch of commands so that the utility can process sections of the code (as marked by the GO commands) separately instead of all at once. Declared variables (@RowCount) only exist within a section of code delineated by the GO statement and are cleared after that group of statements is run. For more information, see *http://msdn.microsoft.com/en-us/library/ms188037.aspx.*

Triggers

SQL Server 2008 supports the INSTEAD OF and AFTER triggers, but not a BEFORE trigger. A table may have one or more AFTER triggers for insert, update, and delete actions; AFTER triggers may not be assigned to views. A view or table may have at most one INSTEAD OF trigger for each triggering by an insert, an update, or delete action.

In SQL Server, triggers can roll back the transactions that caused them to be fired. When a trigger executes a ROLLBACK or ROLLBACK TRANSACTION command (two versions of the same command), all work done by the transaction that caused the trigger to be fired will be rolled back. If the trigger contains instructions after the ROLLBACK command, those instructions will be executed. However, any instructions in the transaction after the statement that caused the trigger to be fired will not be executed.

Amazon Ex. 1008.0420

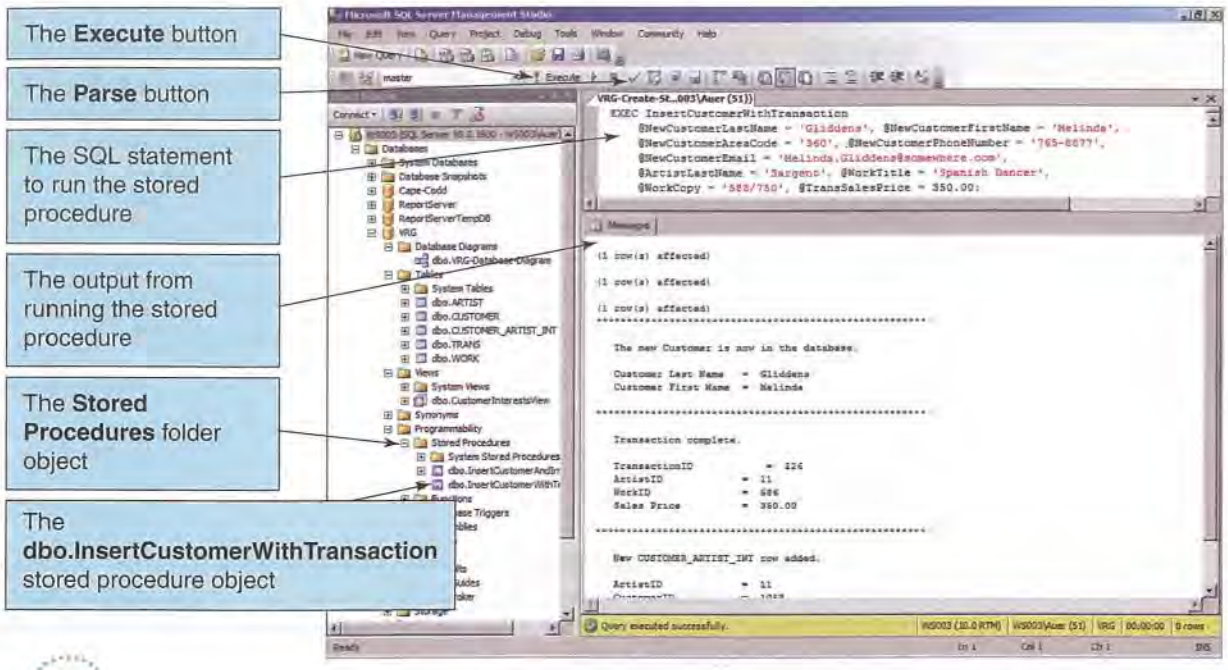


Figure 10-31

Running InsertCustomerWithTransaction Stored Procedure

For insert and update triggers, the new values for every column of the table being processed will be stored in a pseudotable named *inserted*. If, for example, a new row is being added to the ARTIST table, the pseudotable *inserted* will have five columns: LastName, FirstName, Nationality, DateOfBirth, and DateDeceased. Similarly, for update and delete commands, the old values for every column of the table being updated or deleted will be stored in the pseudotable named *deleted*. You will see how to use these pseudotables in the examples that follow.

The next four sections illustrate four triggers for each of the trigger functions described in Chapter 7. The best way to create these triggers is by keying them into an SQL script (using CREATE TRIGGER or ALTER TRIGGER as the first statement) in the Microsoft SQL Server Management Studio SQL editor. You can create a trigger by right-clicking the Triggers folder object for each table, but this only creates a skeleton trigger template for you to edit. It is better to create and test your own SQL script. For the VRG database, use an SQL script named VRG-Create-Triggers.sql, with introductory comments and comment lines to separate the triggers and the SQL statements used to test them. Use the previously discussed highlighting technique to control which SQL statement is actually being run.

A Trigger for Setting Default Values

Triggers can be used to set default values that are more complex than those that can be set with the Default constraint on a column definition. For example, the View Ridge Gallery has a pricing policy that says that the default AskingPrice of an artwork depends on whether the piece has been in the gallery before. If not, the default AskingPrice is twice the AcquisitionPrice. If the artwork has been in the gallery before, the default price is the larger of twice the AcquisitionPrice or the AcquisitionPrice plus the average net gain of the work in the past. The TRANS_AfterInsertSetAskingPrice trigger shown in Figure 10-32 implements this pricing policy.

Note the use of the AFTER keyword in this trigger code. We can use the keywords FOR or AFTER here, but both of these indicate an AFTER trigger—SQL Server does not support BEFORE triggers.

In the trigger, after the variables are declared, the new values of WorkID and AcquisitionPrice are obtained from the *inserted* pseudo table. Then, a SELECT FROM dbo.TRANS is executed to count the number of rows with the given WorkID. This trigger uses the system function @@rowCount, which contains the number of rows processed in the preceding Transact-SQL statement. Its value must be checked immediately after the statement is executed or saved in a variable to be checked later. Here, the trigger immediately uses the value of @@rowCount.

Amazon Ex. 1008.0421