APPENDIX

Solutions to the Exercises

The appendix provides answers to the exercise questions in Chapters 2 through 8.

Chapter 2: Writing Simple SELECT Queries

This section provides solutions to the exercises on writing simple **SELECT** queries.

Solutions to Exercise 2-1: Using the SELECT Statement

Use the AdventureWorksLT2008 database to complete this exercise.

1. Write a **SELECT** statement that lists the customers along with their ID numbers. Include the last names, first names, and company names.

   ```
   SELECT CustomerID, LastName, FirstName, CompanyName
   FROM SalesLT.Customer;
   ```

2. Write a **SELECT** statement that lists the name, product number, and color of each product.

   ```
   SELECT Name, ProductNumber, Color
   FROM SalesLT.Product;
   ```

3. Write a **SELECT** statement that lists the customer ID numbers and sales order ID numbers from the **SalesLT.SalesOrderHeader** table.

   ```
   SELECT CustomerID, SalesOrderID
   FROM SalesLT.SalesOrderHeader;
   ```

4. Answer this question: Why should you specify column names rather than an asterisk when writing the **SELECT** list? Give at least two reasons.

   You would do this to decrease the amount of network traffic and increase the performance of the query, retrieving only the columns needed for the application or report. You can also keep users from seeing confidential information by retrieving only the columns they should see.

Solutions to Exercise 2-2: Filtering Data

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query using a **WHERE** clause that displays all the employees listed in the **HumanResources.Employee** table who have the job title Research and Development Engineer. Display the business entity ID number, the login ID, and the title for each one.

```
SELECT BusinessEntityID, JobTitle, LoginID
FROM HumanResources.Employee
WHERE JobTitle = 'Research and Development Engineer';
```

2. Write a query using a **WHERE** clause that displays all the names in **Person.Person** with the middle name J. Display the first, last, and middle names along with the ID numbers.

```
SELECT FirstName, MiddleName, LastName, BusinessEntityID
FROM Person.Person
WHERE MiddleName = 'J';
```

3. Write a query displaying all the columns of the **Production.ProductCostHistory** table from the rows that were modified on June 17, 2003. Be sure to use one of the features in SQL Server Management Studio to help you write this query.

   In SQL Server Management Studio, expand the AdventureWorks2008 database. Expand Tables. Right-click the **Production.ProductCostHistory** table, and choose "Select table as." Select "Select to" and New Query Editor Window. Then type in the **WHERE** clause.

```
SELECT [ProductID]
      ,[StartDate]
      ,[EndDate]
      ,[StandardCost]
      ,[ModifiedDate]
  FROM [AdventureWorks2008].[Production].[ProductCostHistory]
WHERE ModifiedDate = '2003-06-17';
GO
```

4. Rewrite the query you wrote in question 1, changing it so that the employees who do not have the title Research and Development Engineer are displayed.

```
SELECT BusinessEntityID, JobTitle, LoginID
FROM HumanResources.Employee
WHERE JobTitle <> 'Research and Development Engineer';
```

5. Write a query that displays all the rows from the **Person.Person** table where the rows were modified after December 29, 2000. Display the business entity ID number, the name columns, and the modified date.

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName, ModifiedDate
FROM Person.Person
WHERE ModifiedDate > '2000-12-29';
```

6. Rewrite the last query so that the rows that were not modified on December 29, 2000, are displayed.

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName, ModifiedDate
FROM Person.Person
WHERE ModifiedDate <> '2000-12-29';
```

7. Rewrite the query from question 5 so that it displays the rows modified during December 2000.

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName, ModifiedDate
FROM Person.Person
WHERE ModifiedDate BETWEEN '2000-12-01' AND '2000-12-31';
```

8. Rewrite the query from question 5 so that it displays the rows that were not modified during December 2000.

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName, ModifiedDate
FROM Person.Person
WHERE ModifiedDate NOT BETWEEN '2000-12-01' AND '2000-12-31';
```

9. Explain why a **WHERE** clause should be used in many of your T-SQL queries.

   Most of the time the application or report will not require all the rows. The query should be filtered to include only the required rows to cut down on network traffic and increase SQL Server performance since returning a smaller number of rows is usually more efficient.

Solutions to Exercise 2-3: Filtering with Wildcards

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query that displays the product ID and name for each product from the **Production.Product** table with the name starting with *Chain*.

```
SELECT ProductID, Name
FROM Production.Product
WHERE Name LIKE 'Chain%';
```

2. Write a query like the one in question 1 that displays the products with *helmet* in the name.

```
SELECT ProductID, Name
FROM Production.Product
WHERE Name LIKE '%helmet%';
```

3. Change the last query so that the products without *helmet* in the name are displayed.

```
SELECT ProductID, Name
FROM Production.Product
WHERE Name NOT LIKE '%helmet%';
```

4. Write a query that displays the business entity ID number, first name, middle name, and last name from the `Person.Person` table for only those rows that have *E* or *B* stored in the middle name column.

```
SELECT BusinessEntityID, FirstName, MiddleName, LastName
FROM Person.Person
WHERE MiddleName LIKE '[E,B]';
```

5. Explain the difference between the following two queries:

```
SELECT FirstName
FROM Person.Person
WHERE LastName LIKE 'Ja%es';

SELECT FirstName
FROM Person.Person
WHERE LastName LIKE 'Ja_es';
```

   The first query will return rows with any number of characters replacing the percent sign. The second query will allow only one character to replace the underscore character.

Solutions to Exercise 2-4: Filtering with Multiple Predicates

Use the AdventureWorks2008 database to complete this exercise. Be sure to check you results to assure that they make sense.

1. Write a query displaying the order ID, order date, and total due from the `Sales.SalesOrderHeader` table. Retrieve only those rows where the order was placed during the month of September 2001 and the total due exceeded $1,000.

```
SELECT SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2001-09-01' AND '2001-09-30'
    AND TotalDue > 1000;
```

2. Change the query in question 1 so that only the dates September 1–3, 2001, are retrieved. See whether you can figure out three different ways to write this query.

```
SELECT SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2001-09-01' AND '2001-09-03'
    AND TotalDue > 1000;

SELECT SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
WHERE OrderDate IN ('2001-09-01', '2001-09-02', '2001-09-03')
    AND TotalDue > 1000;

SELECT SalesOrderID, OrderDate, TotalDue
FROM Sales.SalesOrderHeader
WHERE (OrderDate >= '2001-09-01' AND OrderDate <= '2001-09-03')
    AND TotalDue > 1000;
```

3. Write a query displaying the sales orders where the total due exceeds $1,000. Retrieve only those rows where the salesperson ID is 279 or the territory ID is 6.

```
SELECT SalesOrderID, OrderDate, TotalDue, SalesPersonID, TerritoryID
FROM Sales.SalesOrderHeader
WHERE TotalDue > 1000 AND (SalesPersonID = 279 OR TerritoryID = 6);
```

4. Change the query in question 3 so that territory 4 is included.

```
SELECT SalesOrderID, OrderDate, TotalDue, SalesPersonID, TerritoryID
FROM Sales.SalesOrderHeader
WHERE TotalDue > 1000 AND (SalesPersonID = 279 OR TerritoryID IN (6,4));
```

5. Explain when it makes sense to use the **IN** operator.

   You will probably want to use the **IN** operator when you are checking a column for more than one possible value.

Solutions to Exercise 2-5: Working with Nothing

Use the AdventureWorks2008 database to complete this exercise. Make sure you consider how **NULL** values will affect your results.

1. Write a query displaying the **ProductID**, **Name**, and **Color** columns from rows in the **Production.Product** table. Display only those rows where no color has been assigned.

```
SELECT ProductID, Name, Color
FROM Production.Product
WHERE Color IS NULL;
```

2. Write a query displaying the **ProductID**, **Name**, and **Color** columns from rows in the **Production.Product** table. Display only those rows in which the color is not blue.

   Here are two possible solutions:

```
SELECT ProductID, Name, Color
FROM Production.Product
WHERE Color IS NULL OR Color <> 'Blue';

SELECT ProductID, Name, Color
FROM Production.Product
WHERE ISNULL(Color,'') <> 'Blue';
```

3. Write a query displaying **ProductID**, **Name**, **Style**, **Size**, and **Color** from the **Production.Product** table. Include only those rows where at least one of the **Style**, **Size**, or **Color** columns contains a value.

```
SELECT ProductID, Name, Style, Size, Color
FROM Production.Product
WHERE Style IS NOT NULL OR Size IS NOT NULL OR Color IS NOT NULL;
```

Solutions to Exercise 2-6: Performing a Full-Text Search

Use the AdventureWorks2008 database to complete the following tasks. Be sure to take advantage of the full-text indexes in place when writing the queries.

1. Write a query using the **Production.ProductReview** table. Use **CONTAINS** to find all the rows that have the word *socks* in the **Comments** column. Return the **ProductID** and **Comments** columns.

```
SELECT Comments,ProductID
FROM Production.ProductReview
WHERE CONTAINS(Comments,'socks');
```

2. Write a query using the **Production.Document** table. Use **CONTAINS** to find all the rows that have the word *reflector* in any column that is indexed with Full-Text Search. Display the **Title** and **FileName** columns.

```
SELECT Title,FileName
FROM Production.Document
WHERE CONTAINS(*,'reflector');
```

3. Change the query in question 2 so that the rows containing *seat* are not returned in the results.

```
SELECT Title, FileName
FROM Production.Document
WHERE CONTAINS(*,'reflector AND NOT seat')
```

4. Answer this question: When searching a **VARBINARY(MAX)** column that contains Word documents, a **LIKE** search can be used, but the performance will be worse. True or false?

   False, you cannot use **LIKE** with **VARBINARY(MAX)** columns. Use Full-Text searching to search **VARBINARY(MAX)** columns.

Solutions to Exercise 2-7: Sorting Data

Use the AdventureWorks2008 database to complete the exercise to practice sorting the results of your queries.

1. Write a query that returns the business entity ID and name columns from the **Person.Person** table. Sort the results by **LastName**, **FirstName**, and **MiddleName**.

```
SELECT BusinessEntityID, LastName, FirstName, MiddleName
FROM Person.Person
ORDER BY LastName, FirstName, MiddleName;
```

2. Modify the query written in question 1 so that the data is returned in the opposite order.

```
SELECT BusinessEntityID, LastName, FirstName, MiddleName
FROM Person.Person
ORDER BY LastName DESC, FirstName DESC, MiddleName DESC;
```

Solutions to Exercise 2-8: Thinking About Performance

Use the AdventureWorks2008 database to complete this exercise. Be sure to turn on the Include Actual Execution Plan setting before you begin. Type the following code into the query window and then complete each question.

```
USE AdventureWorks2008;
GO

--1
SELECT LastName
FROM Person.Person
WHERE LastName = 'Smith';

--2
SELECT LastName
FROM Person.Person
WHERE LastName LIKE 'Sm%';

--3
SELECT LastName
FROM Person.Person
WHERE LastName LIKE '%mith';

--4
SELECT ModifiedDate
FROM Person.Person
WHERE ModifiedDate BETWEEN '2000-01-01' and '2000-01-31';
```

1. Highlight and run queries 1 and 2. Explain why there is no difference in performance between the two queries.

   Query 1 uses an index to perform an index seek on the **LastName** column to find the rows. Since the wildcard in query 2 begins after the beginning of the value, the database engine can also perform an index seek on the **LastName** column to find the rows in this query.
2. Highlight and run queries 2 and 3. Determine which query performs the best, and explain why you think that is the case.

   Query 2 performs the best. Query 2 takes advantage of the index by performing an index seek on the **LastName** column. Since Query 2 contains the wildcard at the beginning of the value, the database engine must check every value in the index.
3. Highlight and run queries 3 and 4. Determine which query performs the best, and explain why you think this is the case.

   Query 3 performs the best. Even though query 3 must scan every value in the index, no index exists to help query 4. The database engine must scan the clustered index, which is the actual table for query 4. Scanning the table performs worse than scanning a nonclustered index.

Chapter 3: Using Functions and Expressions

This section provides solutions to the exercises on using functions and expressions.

Solutions to Exercise 3-1: Writing Expressions Using Operators

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query that displays in the "AddressLine1 (City PostalCode)" format from the **Person.Address** table.

   ```
   SELECT AddressLine1 + ' (' + City + ' ' + PostalCode + ')'
   FROM Person.Address;
   ```

2. Write a query using the **Production.Product** table displaying the product ID, color, and name columns. If the color column contains a **NULL** value, replace the color with *No Color*.

   ```
   SELECT ProductID, ISNULL(Color,'No Color') AS Color, Name
   FROM Production.Product;
   ```

3. Modify the query written in question 2 so that the description of the product is displayed in the "Name: Color" format. Make sure that all rows display a value even if the **Color** value is missing.

   ```
   SELECT ProductID, Name + ISNULL(': ' + Color,'') AS Description
   FROM Production.Product;
   ```

4. Write a query using the **Production.Product** table displaying a description with the "ProductID: Name" format. Hint: You will need to use a function to write this query.

   Here are two possible answers:

   ```
   SELECT CAST(ProductID AS VARCHAR) + ': ' +  Name AS IDName
   FROM Production.Product;

   SELECT CONVERT(VARCHAR, ProductID) + ': ' + Name AS IDName
   FROM Production.Product;
   ```

5. Explain the difference between the **ISNULL** and **COALESCE** functions.

   You can use **ISNULL** to replace a **NULL** value or column with another value or column. You can use **COALESCE** to return the first non-**NULL** value from a list of values or columns.

Solutions to Exercise 3-2: Using Mathematical Operators

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query using the **Sales.SpecialOffer** table. Display the difference between the **MinQty** and **MaxQty** columns along with the **SpecialOfferID** and **Description** columns.

   ```
   SELECT SpecialOfferID, Description, MaxQty - MinQty AS Diff
   FROM Sales.SpecialOffer;
   ```

2. Write a query using the **Sales.SpecialOffer** table. Multiply the **MinQty** column by the **DiscountPct** column. Include the **SpecialOfferID** and **Description** columns in the results.

   ```
   SELECT SpecialOfferID, Description, MinQty * DiscountPct AS Discount
   FROM Sales.SpecialOffer;
   ```

3. Write a query using the **Sales.SpecialOffer** table that multiplies the **MaxQty** column by the **DiscountPCT** column. If the **MaxQty** value is null, replace it with the value 10. Include the **SpecialOfferID** and **Description** columns in the results.

   ```
   SELECT SpecialOfferID, Description, ISNULL(MaxQty,10) * DiscountPct AS Discount
   FROM Sales.SpecialOffer;
   ```

4. Describe the difference between division and modulo.

   When performing division, you divide two numbers, and the result, the quotient, is the answer. If you are using modulo, you divide two numbers, but the reminder is the answer. If the numbers are evenly divisible, the answer will be zero.

Solutions to Exercise 3-3: Using String Functions

Use the AdventureWorks2008 database to complete this exercise. Be sure to refer to the discussion of the functions to help you figure out which ones to use if you need help.

1. Write a query that displays the first 10 characters of the **AddressLine1** column in the **Person.Address** table.

   Here are two possible solutions:

   ```
   SELECT LEFT(AddressLine1,10) AS Address10
   FROM Person.Address;

   SELECT SUBSTRING(AddressLine1,1,10) AS Address10
   FROM Person.Address;
   ```

2. Write a query that displays characters 10 to 15 of the **AddressLine1** column in the **Person.Address** table.

```
SELECT SUBSTRING(AddressLine1,10,6) AS Address10to15
FROM Person.Address;
```

3. Write a query displaying the first name and last name from the **Person.Person** table all in uppercase.

```
SELECT UPPER(FirstName) AS FirstName, UPPER(LastName) AS LastName
FROM Person.Person;
```

4. The product number in the **Production.Product** contains a hyphen (**-**). Write a query that uses the **SUBSTRING** function and the **CHARINDEX** function to display the characters in the product number following the hyphen. Note: there is also a second hyphen in many of the rows; ignore the second hyphen for this question. Hint: Try writing this statement in two steps, the first using the **CHARINDEX** function and the second adding the **SUBSTRING** function.

```
--Step 1
SELECT ProductNumber, CHARINDEX('-',ProductNumber)
FROM Production.Product;

--Step 2
SELECT ProductNumber,
    SUBSTRING(ProductNumber,CHARINDEX('-',ProductNumber)+1,25) AS ProdNumber
FROM Production.Product;
```

Solutions to Exercise 3-4: Using Date Functions

Use the AdventureWorks2008 database to complete Exercise 3-4.

1. Write a query that calculates the number of days between the date an order was placed and the date that it was shipped using the **Sales.SalesOrderHeader** table. Include the **SalesOrderID**, **OrderDate**, and **ShipDate** columns.

```
SELECT SalesOrderID, OrderDate, ShipDate,
    DATEDIFF(d,OrderDate,ShipDate) AS NumberOfDays
FROM Sales.SalesOrderHeader;
```

2. Write a query that displays only the date, not the time, for the order date and ship date in the **Sales.SalesOrderHeader** table.

```
--Use any of the styles that return only date
SELECT CONVERT(VARCHAR,OrderDate,1) AS OrderDate,
    CONVERT(VARCHAR, ShipDate,1) AS ShipDate
FROM Sales.SalesOrderHeader;
```

3. Write a query that adds six months to each order date in the **Sales.SalesOrderHeader** table. Include the **SalesOrderID** and **OrderDate** columns.

```
SELECT SalesOrderID, OrderDate, DATEADD(m,6,OrderDate) Plus6Months
FROM Sales.SalesOrderHeader;
```

4. Write a query that displays the year of each order date and the numeric month of each order date in separate columns in the results. Include the **SalesOrderID** and **OrderDate** columns.

   Here are two possible solutions:

```
SELECT SalesOrderID, OrderDate, YEAR(OrderDate) AS OrderYear,
    MONTH(OrderDate) AS OrderMonth
FROM Sales.SalesOrderHeader;
SELECT SalesOrderID, OrderDate, DATEPART(yyyy,OrderDate) AS OrderYear,
    DATEPART(m,OrderDate) AS OrderMonth
FROM Sales.SalesOrderHeader;
```

5. Change the query written in question 4 to display the month name instead.

```
SELECT SalesOrderID, OrderDate, DATEPART(yyyy,OrderDate) AS OrderYear,
    DATENAME(m,OrderDate) AS OrderMonth
FROM Sales.SalesOrderHeader;
```

Solutions to Exercise 3-5: Using Mathematical Functions

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query using the **Sales.SalesOrderHeader** table that displays the **SubTotal** rounded to two decimal places. Include the **SalesOrderID** column in the results.

```
SELECT SalesOrderID, ROUND(SubTotal,2) AS SubTotal
FROM Sales.SalesOrderHeader;
```

2. Modify the query in question 1 so that the **SubTotal** is rounded to the nearest dollar but still displays two zeros to the right of the decimal place.

```
SELECT SalesOrderID, ROUND(SubTotal,0) AS SubTotal
FROM Sales.SalesOrderHeader;
```

3. Write a query that calculates the square root of the **SalesOrderID** value from the **Sales.SalesOrderHeader** table.

```
SELECT SQRT(SalesOrderID) AS OrderSQRT
FROM Sales.SalesOrderHeader;
```

4. Write a statement that generates a random number between 1 and 10 each time it is run.

```
SELECT CAST(RAND() * 10 AS INT) + 1;
```

Solutions to Exercise 3-6: Using System Functions

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query using the **HumanResources.Employee** table to display the **BusinessEntityID** column. Also include a **CASE** statement that displays "Even" when the **BusinessEntityID** value is an even number or "Odd" when it is odd. Hint: Use the modulo operator.

```
SELECT BusinessEntityID,
    CASE BusinessEntityID % 2 WHEN 0 THEN 'Even' ELSE 'Odd' END
FROM HumanResources.Employee;
```

2. Write a query using the **Sales.SalesOrderDetail** table to display a value ("Under 10" or "10–19" or "20–29" or "30–39" or "40 and over") based on the **OrderQty** value by using the **CASE** function. Include the **SalesOrderID** and **OrderQty** columns in the results.

```
SELECT SalesOrderID, OrderQty,
    CASE WHEN OrderQty BETWEEN 0 AND 9 THEN 'Under 10'
         WHEN OrderQty BETWEEN 10 AND 19 THEN '10-19'
         WHEN OrderQty BETWEEN 20 AND 29 THEN '20-29'
         WHEN OrderQty BETWEEN 30 AND 39 THEN '30-39'
         ELSE '40 and over' end AS range
FROM Sales.SalesOrderDetail;
```

3. Using the **Person.Person** table, build the full names using **Title**, **FirstName**, **MiddleName**, **LastName**, and **Suffix** columns. Check the table definition to see which columns allow **NULL** values, and use the **COALESCE** function on the appropriate columns.

```
SELECT COALESCE(Title + ' ','') + FirstName +
    COALESCE(' ' + MiddleName,'') + ' ' + LastName +
    COALESCE(', ' + Suffix,'')
FROM Person.Person;
```

4. Look up the **SERVERPROPERTY** function in Books Online. Write a statement that displays the edition, instance name, and machine name using this function.

```
SELECT SERVERPROPERTY('Edition'),
   SERVERPROPERTY('InstanceName'),
   SERVERPROPERTY('MachineName');
```

Solutions to Exercise 3-7: Using Functions in the WHERE and ORDER BY Clauses

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query using the **Sales.SalesOrderHeader** table to display the orders placed during 2001 by using a function. Include the **SalesOrderID** and **OrderDate** columns in the results.

```
SELECT SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader
WHERE YEAR(OrderDate) = 2001;
```

2. Write a query using the **Sales.SalesOrderHeader** table listing the sales in order of the month the order was placed and then the year the order was placed. Include the **SalesOrderID** and **OrderDate** columns in the results.

```
SELECT SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader
ORDER BY MONTH(OrderDate), YEAR(OrderDate);
```

3. Write a query that displays the **PersonType** and the name columns from the **Person.Person** table. Sort the results so that rows with a **PersonType** of **IN**, **SP**, or **SC** sort by **LastName**. The other rows should sort by **FirstName**. Hint: Use the **CASE** function.

```
SELECT PersonType, FirstName, MiddleName, LastName
FROM Person.Person
ORDER BY CASE WHEN PersonType IN ('IN','SP','SC') THEN LastName
         ELSE FirstName END;
```

Solutions to Exercise 3-8: Thinking About Performance

Use the AdventureWorks2008 database to complete this exercise. Make sure you have the Include Actual Execution Plan setting toggled on before starting this exercise.

1. Type in and execute the following code. View the execution plans once query execution completes, and explain whether one query performs better than the other and why.

```
USE AdventureWorks2008;
GO
```

```
--1
SELECT Name
FROM Production.Product
WHERE Name LIKE 'B%';

--2
SELECT Name
FROM Production.Product
WHERE CHARINDEX('B',Name) = 1;
```

Query 1 performs better because it performs an index seek on the **Name** column. Query 2 must scan the entire index, applying the function to each value of **Name**.

2. Type in and execute the following code. View the execution plans once query execution completes, and explain whether one query performs better than the other and why.

```
USE AdventureWorks2008;
GO

--1
SELECT LastName
FROM Person.Person
WHERE LastName LIKE '%i%';

--2
SELECT LastName
FROM Person.Person
WHERE CHARINDEX('i',LastName) > 0;
```

The queries have the same performance because both queries must scan the index. Query 1 contains a wildcard at the beginning of the search term, and query 2 has a function that takes the column name as an argument.

Chapter 4: Querying Multiple Tables

This section provides solutions to the exercises on querying multiple tables.

Solutions to Exercise 4-1: Writing Inner Joins

Use the AdventureWorks2008 to complete this exercise.

1. The **HumanResources.Employee** table does not contain the employee names. Join that table to the **Person.Person** table on the **BusinessEntityID** column. Display the job title, birth date, first name, and last name.

```
SELECT JobTitle, BirthDate, FirstName, LastName
FROM HumanResources.Employee AS E
INNER JOIN Person.Person AS P ON E.BusinessEntityID = P.BusinessEntityID;
```

2. The customer names also appear in the **Person.Person** table. Join the **Sales.Customer** table to the **Person.Person** table. The **BusinessEntityID** column in the **Person.Person** table matches the **PersonID** column in the **Sales.Customer** table. Display the **CustomerID**, **StoreID**, and **TerritoryID** columns along with the name columns.

```
SELECT CustomerID, StoreID, TerritoryID, FirstName, MiddleName, LastName
FROM Sales.Customer AS C
INNER JOIN Person.Person AS P ON C.PersonID = P.BusinessEntityID;
```

3. Extend the query written in question 2 to include the **Sales.SalesOrderHeader** table. Display the **SalesOrderID** column along with the columns already specified. The **Sales.SalesOrderHeader** table joins the **Sales.Customer** table on **CustomerID**.

```
SELECT c.CustomerID, StoreID, c.TerritoryID, FirstName, MiddleName,
    LastName, SalesOrderID
FROM Sales.Customer AS C
INNER JOIN Person.Person AS P ON C.PersonID = P.BusinessEntityID
INNER JOIN Sales.SalesOrderHeader AS S ON S.CustomerID = C.CustomerID;
```

4. Write a query that joins the **Sales.SalesOrderHeader** table to the **Sales.SalesPerson** table. Join the **BusinessEntityID** column from the **Sales.SalesPerson** table to the **SalesPersonID** column in the **Sales.SalesOrderHeader** table. Display the **SalesOrderID** along with the **SalesQuota** and **Bonus**.

```
SELECT SalesOrderID, SalesQuota, Bonus
FROM Sales.SalesOrderHeader AS S
INNER JOIN Sales.SalesPerson AS SP
    ON S.SalesPersonID = SP.BusinessEntityID;
```

5. Add the name columns to the query written in question 4 by joining on the **Person.Person** table. See whether you can figure out which columns will be used to write the join.

   You can join the **Person.Person** table on the **SalesOrderHeader** table or the **Sales.SalesPerson** table.

```
SELECT SalesOrderID, SalesQuota, Bonus, FirstName, MiddleName, LastName
FROM Sales.SalesOrderHeader AS S
INNER JOIN Sales.SalesPerson AS SP ON S.SalesPersonID = SP.BusinessEntityID
INNER JOIN Person.Person AS P ON SP.BusinessEntityID = P.BusinessEntityID;
```

```
SELECT SalesOrderID, SalesQuota, Bonus, FirstName, MiddleName, LastName
FROM Sales.SalesOrderHeader AS S
INNER JOIN Sales.SalesPerson AS SP ON S.SalesPersonID = SP.BusinessEntityID
INNER JOIN Person.Person AS P ON S.SalesPersonID = P.BusinessEntityID;
```

6. The catalog description for each product is stored in the **Production.ProductModel** table. Display the columns that describe the product from the **Production.Product** table, such as the color and size along with the catalog description for each product.

```
SELECT PM.CatalogDescription, Color, Size
FROM Production.Product AS P
INNER JOIN Production.ProductModel AS PM ON P.ProductModelID = PM.ProductModelID;
```

7. Write a query that displays the names of the customers along with the product names that they have purchased. Hint: Five tables will be required to write this query!

```
SELECT FirstName, MiddleName, LastName, Prod.Name
FROM Sales.Customer AS C
INNER JOIN Person.Person AS P ON C.PersonID = P.BusinessEntityID
INNER JOIN Sales.SalesOrderHeader AS SOH ON C.CustomerID = SOH.CustomerID
INNER JOIN Sales.SalesOrderDetail AS SOD
    ON SOH.SalesOrderID = SOD.SalesOrderID
INNER JOIN Production.Product AS Prod ON SOD.ProductID = Prod.ProductID;
```

Solutions to Exercise 4-2: Writing Outer Joins

Use the AdventureWorks2008 and AdventureWorks (question 7) databases to complete this exercise.

1. Write a query that displays all the products along with the **SalesOrderID** even if an order has never been placed for that product. Join to the **Sales.SalesOrderDetail** table using the **ProductID** column.

```
SELECT SalesOrderID, P.ProductID, P.Name
FROM Production.Product AS P
LEFT OUTER JOIN Sales.SalesOrderDetail
    AS SOD ON P.ProductID = SOD.ProductID;
```

2. Change the query written in question 1 so that only products that have not been ordered show up in the query.

```
SELECT SalesOrderID, P.ProductID, P.Name
FROM Production.Product AS P
LEFT OUTER JOIN Sales.SalesOrderDetail
    AS SOD ON P.ProductID = SOD.ProductID
WHERE SalesOrderID IS NULL;
```

3. Write a query that returns all the rows from the **Sales.SalesPerson** table joined to the **Sales.SalesOrderHeader** table along with the **SalesOrderID** column even if no orders match. Include the **SalesPersonID** and **SalesYTD** columns in the results.

```
SELECT SalesOrderID, SalesPersonID, SalesYTD
FROM Sales.SalesPerson AS SP
LEFT OUTER JOIN Sales.SalesOrderHeader AS SOH
    ON SP.BusinessEntityID = SOH.SalesPersonID;
```

4. Change the query written in question 3 so that the salesperson's name also displays from the **Person.Person** table.

```
SELECT SalesOrderID, SalesPersonID, SalesYTD, FirstName,
    MiddleName, LastName
FROM Sales.SalesPerson AS SP
LEFT OUTER JOIN Sales.SalesOrderHeader AS SOH
    ON SP.BusinessEntityID = SOH.SalesPersonID
LEFT OUTER JOIN Person.Person AS P
    ON P.BusinessEntityID = SP.BusinessEntityID;
```

5. The **Sales.SalesOrderHeader** table contains foreign keys to the **Sales.CurrencyRate** and **Purchasing.ShipMethod** tables. Write a query joining all three tables, making sure it contains all rows from **Sales.SalesOrderHeader**. Include the **CurrencyRateID**, **AverageRate**, **SalesOrderID**, and **ShipBase** columns.

```
SELECT CR.CurrencyRateID, CR.AverageRate, SM.ShipBase, SalesOrderID
FROM Sales.SalesOrderHeader AS SOH
LEFT OUTER JOIN Sales.CurrencyRate AS CR
    ON SOH.CurrencyRateID = CR.CurrencyRateID
LEFT OUTER JOIN Purchasing.ShipMethod AS SM
    ON SOH.ShipMethodID = SM.ShipMethodID;
```

6. Write a query that returns the **BusinessEntityID** column from the **Sales.SalesPerson** table along with every **ProductID** from the **Production.Product** table.

```
SELECT SP.BusinessEntityID, P.ProductID
FROM Sales.SalesPerson AS SP CROSS JOIN Production.Product AS P;
```

7. Starting with the query written in Listing 4-13, join the table **a** to the **Person.Contact** table to display the employee's name. The **EmployeeID** column joins the **ContactID** column.

```
USE AdventureWorks;
GO
SELECT a.EmployeeID AS Employee,
    a.Title AS EmployeeTitle,
    b.EmployeeID AS ManagerID,
    b.Title AS ManagerTitle,
    c.FirstName, c.MiddleName, c.LastName
FROM HumanResources.Employee AS a
LEFT OUTER JOIN HumanResources.Employee AS b
ON a.ManagerID = b.EmployeeID
LEFT OUTER JOIN Person.Contact AS c ON a.EmployeeID = c.ContactID;
```

Solutions to Exercise 4-3: Writing Subqueries

Use the AdventureWorks2008 database to complete this exercise.

1. Using a subquery, display the product names and product ID numbers from the **Production.Product** table that have been ordered.

```
SELECT ProductID, Name
FROM Production.Product
WHERE ProductID IN (SELECT ProductID FROM Sales.SalesOrderDetail);
```

2. Change the query written in question 1 to display the products that have not been ordered.

```
SELECT ProductID, Name
FROM Production.Product
WHERE ProductID NOT IN (
    SELECT ProductID FROM Sales.SalesOrderDetail
    WHERE ProductID IS NOT NULL);
```

3. If the **Production.ProductColor** table is not part of the AdventureWorks2008 database, run the code in Listing 4-11 to create it. Write a query using a subquery that returns the rows from the **Production.ProductColor** table that are not being used in the **Production.Product** table.

```
SELECT Color
FROM Production.ProductColor
WHERE Color NOT IN (
    SELECT Color FROM Production.Product WHERE Color IS NOT NULL);
```

4. Write a query that displays the colors used in the **Production.Product** table that are not listed in the **Production.ProductColor** table using a subquery. Use the keyword **DISTINCT** before the column name to return each color only once.

```
SELECT DISTINCT Color
FROM Production.Product
WHERE Color NOT IN (
    SELECT Color FROM Production.ProductColor WHERE Color IS NOT NULL);
```

5. Write a **UNION** query that combines the **ModifiedDate** from **Person.Person** and the **HireDate** from **HumanResources.Employee**.

```
SELECT ModifiedDate
FROM Person.Person
UNION
SELECT HireDate
FROM HumanResources.Employee;
```

Solutions to Exercise 4-4: Exploring Derived Tables and Common Table Expressions

Use the AdventureWorks2008 database to complete this exercise.

1. Using a derived table, join the **Sales.SalesOrderHeader** table to the **Sales.SalesOrderDetail** table. Display the **SalesOrderID**, **OrderDate**, and **ProductID** columns in the results. The **Sales.SalesOrderDetail** table should be inside the derived table query.

```
SELECT SOH.SalesOrderID, SOH.OrderDate, ProductID
FROM Sales.SalesOrderHeader AS SOH
INNER JOIN (
    SELECT SalesOrderID, ProductID
    FROM Sales.SalesOrderDetail) AS SOD
    ON SOH.SalesOrderID = SOD.SalesOrderID;
```

2. Rewrite the query in question 1 with a common table expression.

```
WITH SOD AS (
    SELECT SalesOrderID, ProductID
    FROM Sales.SalesOrderDetail
    )
SELECT SOH.SalesOrderID, SOH.OrderDate, ProductID
FROM Sales.SalesOrderHeader AS SOH
INNER JOIN SOD ON SOH.SalesOrderID = SOD.SalesOrderID;
```

3. Write a query that displays all customers along with the orders placed in 2001. Use a common table expression to write the query and include the **CustomerID**, **SalesOrderID**, and **OrderDate** columns in the results.

```
WITH SOH AS (
    SELECT SalesOrderID, OrderDate, CustomerID
    FROM Sales.SalesOrderHeader
    WHERE OrderDate BETWEEN '1/1/2001' AND '12/31/2001'
    )
SELECT C.CustomerID, SalesOrderID, OrderDate
FROM Sales.Customer AS C
LEFT OUTER JOIN SOH ON C.CustomerID = SOH.CustomerID;
```

Solutions to Exercise 4-5: Thinking About Performance

Use the AdventureWorks2008 database to complete this exercise.

Run the following code to add and populate a new column, **OrderID**, to the **Sales.SalesOrderDetail** table. After running the code, the new column will contain the same data as the **SalesOrderID** column.

```
USE AdventureWorks2008;
GO
ALTER TABLE Sales.SalesOrderDetail ADD OrderID INT NULL;
GO
UPDATE Sales.SalesOrderDetail SET OrderID = SalesOrderID;
```

1. Make sure that the Include Actual Execution Plan is turned on before running the following code. View the execution plans, and explain why one query performs better than the other.

```
--1
SELECT o.SalesOrderID,d.SalesOrderDetailID
FROM Sales.SalesOrderHeader AS o
INNER JOIN Sales.SalesOrderDetail AS d ON o.SalesOrderID = d.SalesOrderID;
```

```
--2
SELECT o.SalesOrderID,d.SalesOrderDetailID
FROM Sales.SalesOrderHeader AS o
INNER JOIN Sales.SalesOrderDetail AS d
    ON o.SalesOrderID = d.OrderID;
```

Query 1, which joins the **Sales.SalesOrderDetail** table to **Sales.SalesOrderHeader** on the **SalesOrderID** column, performs better because there is a nonclustered index defined on the **SalesOrderID** column. There is not an index on the new **OrderID** column, so a clustered index scan is performed on the **Sales.SalesOrderDetail** table to join the tables in query 2.

2. Compare the execution plans of the derived table example (Listing 4-18) and the CTE example (Listing 4-19). Explain why the query performance is the same or why one query performs better than the other.

   The performance of the two queries is the same. These two techniques are just different ways to do the same thing in this case.

Chapter 5: Grouping and Summarizing Data

This section provides solutions to the exercises on grouping and summarizing data.

Solutions to Exercise 5-1: Using Aggregate Functions

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query to determine the number of customers in the **Sales.Customer** table.

   ```
   SELECT COUNT(*) AS CountOfCustomers
   FROM Sales.Customer;
   ```

2. Write a query that lists the total number of products ordered. Use the **OrderQty** column of the **Sales.SalesOrderDetail** table and the **SUM** function.

   ```
   SELECT SUM(OrderQty) AS TotalProductsOrdered
   FROM Sales.SalesOrderDetail;
   ```

3. Write a query to determine the price of the most expensive product ordered. Use the **UnitPrice** column of the **Sales.SalesOrderDetail** table.

   ```
   SELECT MAX(UnitPrice) AS MostExpensivePrice
   FROM Sales.SalesOrderDetail;
   ```

4. Write a query to determine the average freight amount in the **Sales.SalesOrderHeader** table.

```
SELECT AVG(Freight) AS AverageFreight
FROM Sales.SalesOrderHeader;
```

5. Write a query using the **Production.Product** table that displays the minimum, maximum, and average **ListPrice**.

```
SELECT MIN(ListPrice) AS Minimum,
    MAX(ListPrice) AS Maximum,
    AVG(ListPrice) AS Average
FROM Production.Product;
```

Solutions to Exercise 5-2: Using the GROUP BY Clause

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query that shows the total number of items ordered for each product. Use the **Sales.SalesOrderDetail** table to write the query.

```
SELECT SUM(OrderQty) AS TotalOrdered, ProductID
FROM Sales.SalesOrderDetail
GROUP BY ProductID;
```

2. Write a query using the **Sales.SalesOrderDetail** table that displays a count of the detail lines for each **SalesOrderID**.

```
SELECT COUNT(*) AS CountOfOrders, SalesOrderID
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID;
```

3. Write a query using the **Production.Product** table that lists a count of the products in each product line.

```
SELECT COUNT(*) AS CountOfProducts, ProductLine
FROM Production.Product
GROUP BY ProductLine;
```

4. Write a query that displays the count of orders placed by year for each customer using the **Sales.SalesOrderHeader** table.

```
SELECT CustomerID, COUNT(*) AS CountOfSales, YEAR(OrderDate) AS OrderYear
FROM Sales.SalesOrderHeader
GROUP BY CustomerID, YEAR(OrderDate);
```

Solutions to Exercise 5-3: Using the HAVING Clause

Use the AdventureWorks2008 to complete this exercise.

1. Write a query that returns a count of detail lines in the **Sales.SalesOrderDetail** table by **SalesOrderID**. Include only those sales that have more than three detail lines.

```
SELECT COUNT(*) AS CountOfDetailLines, SalesOrderID
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
HAVING COUNT(*) > 3;
```

2. Write a query that creates a sum of the **LineTotal** in the **Sales.SalesOrderDetail** table grouped by the **SalesOrderID**. Include only those rows where the sum exceeds 1,000.

```
SELECT SUM(LineTotal) AS SumOfLineTotal, SalesOrderID
FROM Sales.SalesOrderDetail
GROUP BY SalesOrderID
HAVING SUM(LineTotal) > 1000;
```

3. Write a query that groups the products by **ProductModelID** along with a count. Display the rows that have a count that equals 1.

```
SELECT ProductModelID, COUNT(*) AS CountOfProducts
FROM Production.Product
GROUP BY ProductModelID
HAVING COUNT(*) = 1;
```

4. Change the query in question 3 so that only the products with the color blue or red are included.

```
SELECT ProductModelID, COUNT(*) AS CountOfProducts, Color
FROM Production.Product
WHERE Color IN ('Blue','Red')
GROUP BY ProductModelID, Color
HAVING COUNT(*) = 1;
```

Solutions to Exercise 5-4: Using DISTINCT

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query using the **Sales.SalesOrderDetail** table to come up with a count of unique **ProductID** values that have been ordered.

```
SELECT COUNT(DISTINCT ProductID) AS CountOFProductID
FROM Sales.SalesOrderDetail;
```

2. Write a query using the **Sales.SalesOrderHeader** table that returns the count of unique **TerritoryID** values per customer.

```
SELECT COUNT(DISTINCT TerritoryID) AS CountOfTerritoryID, CustomerID
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

Solutions to Exercise 5-5: Using Aggregate Queries with More Than One Table

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query joining the **Person.Person**, **Sales.Customer**, and **Sales.SalesOrderHeader** tables to return a list of the customer names along with a count of the orders placed.

```
SELECT COUNT(*) AS CountOfOrders, FirstName, MiddleName, LastName
FROM Person.Person AS P
INNER JOIN Sales.Customer AS C ON P.BusinessEntityID = C.PersonID
INNER JOIN Sales.SalesOrderHeader AS SOH ON C.CustomerID = SOH.CustomerID
GROUP BY FirstName, MiddleName, LastName;
```

2. Write a query using the **Sales.SalesOrderHeader**, **Sales.SalesOrderDetail**, and **Production.Product** tables to display the total sum of products by **ProductID** and **OrderDate**.

```
SELECT SUM(OrderQty) SumOfOrderQty, P.ProductID, SOH.OrderDate
FROM Sales.SalesOrderHeader AS SOH
INNER JOIN Sales.SalesOrderDetail AS SOD
    ON SOH.SalesOrderID = SOD.SalesOrderDetailID
INNER JOIN Production.Product AS P ON SOD.ProductID = P.ProductID
GROUP BY P.ProductID, SOH.OrderDate;
```

Solutions to Exercise 5-6: Isolating Aggregate Query Logic

Use the AdventureWorks2008 database to complete this exercise.

1. Write a query that joins the **HumanResources.Employee** table to the **Person.Person** table so that you can display the **FirstName**, **LastName**, and **HireDate** columns for each employee. Display the **JobTitle** along with a count of employees for the title. Use a derived table to solve this query.

```
SELECT FirstName, LastName, e.JobTitle, HireDate, CountOfTitle
FROM HumanResources.Employee AS e
INNER JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID
INNER JOIN (
    SELECT COUNT(*) AS CountOfTitle, JobTitle
    FROM HumanResources.Employee
    GROUP BY JobTitle) AS j ON e.JobTitle = j.JobTitle;
```

2. Rewrite the query from question 1 using a CTE.

```
WITH j AS (SELECT COUNT(*) AS CountOfTitle, JobTitle
    FROM HumanResources.Employee
    GROUP BY JobTitle)
SELECT FirstName, LastName, e.JobTitle, HireDate, CountOfTitle
FROM HumanResources.Employee AS e
INNER JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID
INNER JOIN j ON e.JobTitle = j.JobTitle;
```

3. Rewrite the query from question 1 using the **OVER** clause.

```
SELECT FirstName, LastName, e.JobTitle, HireDate,
    COUNT(*) OVER(PARTITION BY JobTitle) AS CountOfTitle
FROM HumanResources.Employee AS e
INNER JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID
```

4. Display the **CustomerID**, **SalesOrderID**, and **OrderDate** for each **Sales.SalesOrderHeader** row as long as the customer has placed at least five orders. Use any of the techniques from this section to come up with the query.

   Here are three possible solutions:

```
--subquery
SELECT CustomerID, SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader
WHERE CustomerID IN
    (SELECT CustomerID
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID
    HAVING COUNT(*) > 4);
```

```
--CTE
WITH c AS (
    SELECT CustomerID
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID
    HAVING COUNT(*) > 4)
SELECT c.CustomerID, SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader AS SOH
INNER JOIN c ON SOH.CustomerID = c.CustomerID;

--derived table
SELECT c.CustomerID, SalesOrderID, OrderDate
FROM Sales.SalesOrderHeader AS SOH
INNER JOIN (
    SELECT CustomerID
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID
    HAVING COUNT(*) > 4) AS c ON SOH.CustomerID = c.CustomerID;
```

Solutions to Exercise 5-7: Thinking About Performance

Use the AdventureWorks2008 database to complete this exercise.

1. Make sure that the Include Actual Execution Plan setting is turned on before typing and executing the following code. Compare the execution plans to see whether the CTE query performs better than the **OVER** clause query.

```
USE AdventureWorks2008;
GO
--1
WITH SumSale AS
   (SELECT SUM(TotalDue) AS SumTotalDue,
        CustomerID
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID)
SELECT o.CustomerID, TotalDue,
    TotalDue / SumTotalDue * 100 AS PercentOfSales
FROM SumSale INNER JOIN Sales.SalesOrderHeader AS o
ON SumSale.CustomerID = o.CustomerID
ORDER BY CustomerID;
```

```
--2
SELECT CustomerID, TotalDue,
    TotalDue / SUM(TotalDue) OVER(PARTITION BY CustomerID) * 100 AS PercentOfSales
FROM Sales.SalesOrderHeader
ORDER BY CustomerID;
```

The performance is about the same for this example.

2. The following queries each contain two calculations: percent of sales by customer and percent of sales by territory. Type in and execute the code to see the difference in performance. Make sure the Include Actual Execution Plan setting is turned on before running the code.

```
USE AdventureWorks2008;
GO

--1
WITH SumSale AS
   (SELECT SUM(TotalDue) AS SumTotalDue,
        CustomerID
    FROM Sales.SalesOrderHeader
    GROUP BY CustomerID),
 TerrSales AS
    (SELECT SUM(TotalDue) AS SumTerritoryTotalDue, TerritoryID
     FROM Sales.SalesOrderHeader
     GROUP BY TerritoryID )
SELECT o.CustomerID, TotalDue,
    TotalDue / SumTotalDue * 100 AS PercentOfCustSales,
    TotalDue / SumTerritoryTotalDue * 100 AS PercentOfTerrSales
FROM SumSale
INNER JOIN Sales.SalesOrderHeader AS o ON SumSale.CustomerID = o.CustomerID
INNER JOIN TerrSales ON TerrSales.TerritoryID = o.TerritoryID
ORDER BY CustomerID;

--2
SELECT CustomerID, TotalDue,
    TotalDue / SUM(TotalDue) OVER(PARTITION BY CustomerID) * 100 AS
PercentOfCustSales,
    TotalDue / SUM(TotalDue) OVER(PARTITION BY TerritoryID) * 100 AS
PercentOfTerrSales
FROM Sales.SalesOrderHeader
ORDER BY CustomerID;
```

In this case, the CTE in query 1 performs better.

Chapter 6: Manipulating Data

This section provides solutions to the exercises on manipulating data.

Solutions to Exercise 6-1: Inserting New Rows

Use the AdventureWorksLT2008 database to complete this exercise.

Run the following code to create the required tables. You can also download the code from this book's page at **http://www.apress.com** to save typing time.

```
USE AdventureWorksLT2008;
GO
IF  EXISTS (SELECT * FROM sys.objects
            WHERE object_id = OBJECT_ID(N'[dbo].[demoProduct]')
               AND type in (N'U'))
DROP TABLE [dbo].[demoProduct]
GO

CREATE TABLE [dbo].[demoProduct](
    [ProductID] [INT] NOT NULL PRIMARY KEY,
    [Name] [dbo].[Name] NOT NULL,
    [Color] [NVARCHAR](15) NULL,
    [StandardCost] [MONEY] NOT NULL,
    [ListPrice] [MONEY] NOT NULL,
    [Size] [NVARCHAR](5) NULL,
    [Weight] [DECIMAL](8, 2) NULL,
);
IF  EXISTS (SELECT * FROM sys.objects
            WHERE object_id = OBJECT_ID(N'[dbo].[demoSalesOrderHeader]')
                AND type in (N'U'))
DROP TABLE [dbo].[demoSalesOrderHeader]
GO

CREATE TABLE [dbo].[demoSalesOrderHeader](
    [SalesOrderID] [INT] NOT NULL PRIMARY KEY,
    [SalesID] [INT] NOT NULL IDENTITY,
    [OrderDate] [DATETIME] NOT NULL,
    [CustomerID] [INT] NOT NULL,
    [SubTotal] [MONEY] NOT NULL,
    [TaxAmt] [MONEY] NOT NULL,
    [Freight] [MONEY] NOT NULL,
```

```
    [DateEntered] [DATETIME],
    [TotalDue]  AS (ISNULL(([SubTotal]+[TaxAmt])+[Freight],(0))),
    [RV] ROWVERSION NOT NULL);
GO

ALTER TABLE [dbo].[demoSalesOrderHeader] ADD  CONSTRAINT
[DF_demoSalesOrderHeader_DateEntered]
DEFAULT (GETDATE()) FOR [DateEntered];

GO
IF  EXISTS (SELECT * FROM sys.objects
    WHERE object_id = OBJECT_ID(N'[dbo].[demoAddress]')
    AND type in (N'U'))
DROP TABLE [dbo].[demoAddress]
GO

CREATE TABLE [dbo].[demoAddress](
    [AddressID] [INT] NOT NULL IDENTITY PRIMARY KEY,
    [AddressLine1] [NVARCHAR](60) NOT NULL,
    [AddressLine2] [NVARCHAR](60) NULL,
    [City] [NVARCHAR](30) NOT NULL,
    [StateProvince] [dbo].[Name] NOT NULL,
    [CountryRegion] [dbo].[Name] NOT NULL,
    [PostalCode] [NVARCHAR](15) NOT NULL
);
```

1. Write a **SELECT** statement to retrieve data from the **SalesLT.Product** table. Use these values to insert five rows into the **dbo.demoProduct** table using literal values. Write five individual **INSERT** statements.

   The rows you choose to insert may vary.

   ```
   SELECT ProductID, Name, Color,
       StandardCost, ListPrice, Size, Weight
   FROM SalesLT.Product;

   INSERT INTO dbo.demoProduct(ProductID, Name, Color,
       StandardCost, ListPrice, Size, Weight)
   VALUES (680,'HL Road Frame - Black, 58','Black',1059.31,1431.50,'58',1016.04);

   INSERT INTO dbo.demoProduct(ProductID, Name, Color,
       StandardCost, ListPrice, Size, Weight)
   VALUES (706,'HL Road Frame - Red, 58','Red',1059.31, 1431.50,'58',1016.04);
   ```

```
INSERT INTO dbo.demoProduct(ProductID, Name, Color,
    StandardCost, ListPrice, Size, Weight)
VALUES (707,'Sport-100 Helmet, Red','Red',13.0863,34.99,NULL,NULL);

INSERT INTO dbo.demoProduct(ProductID, Name, Color,
    StandardCost, ListPrice, Size, Weight)
VALUES (708,'Sport-100 Helmet, Black','Black',13.0863,34.99,NULL,NULL);
INSERT INTO dbo.demoProduct(ProductID, Name, Color,
    StandardCost, ListPrice, Size, Weight)
VALUES (709,'Mountain Bike Socks, M','White',3.3963,9.50,'M',NULL);
```

2. Insert five more rows into the **dbo.demoProduct** table. This time write one **INSERT** statement.

   The rows you choose to insert may vary.

```
INSERT INTO dbo.demoProduct(ProductID, Name, Color,
StandardCost, ListPrice, Size, Weight)
VALUES (711,'Sport-100 Helmet, Blue','Blue',
        13.0863,34.99,NULL,NULL),
    (712,'AWC Logo Cap','Multi',6.9223,
     8.99,NULL,NULL),
    (713,'Long-Sleeve Logo Jersey,S','Multi',
     38.4923,49.99,'S',NULL),
    (714,'Long-Sleeve Logo Jersey,M','Multi',
     38.4923,49.99,'M',NULL),
    (715,'Long-Sleeve Logo Jersey,L','Multi',
     38.4923,49.99,'L',NULL);
```

3. Write an **INSERT** statement that inserts all the rows into the **dbo.demoSalesOrderHeader** table from the **SalesLT.SalesOrderHeader** table. Hint: Pay close attention to the properties of the columns in the **dbo.demoSalesOrderHeader** table.

   Don't insert a value into the **SalesID**, **DateEntered**, and **RV** columns.

```
INSERT INTO dbo.demoSalesOrderHeader(
    SalesOrderID, OrderDate, CustomerID,
    SubTotal, TaxAmt, Freight)
SELECT SalesOrderID, OrderDate, CustomerID,
    SubTotal, TaxAmt, Freight
FROM SalesLT.SalesOrderHeader;
```

4. Write a **SELECT INTO** statement that creates a table, **dbo.tempCustomerSales**, showing every **CustomerID** from the **SalesLT.Customer** along with a count of the orders placed and the total amount due for each customer.

```
SELECT COUNT(ISNULL(SalesOrderID,0)) AS CountOfORders, c.CustomerID,
    SUM(TotalDue) AS TotalDue
INTO dbo.tempCustomerSales
FROM SalesLT.Customer AS c
LEFT JOIN SalesLT.SalesOrderHeader AS soh ON c.CustomerID = soh.CustomerID
GROUP BY c.CustomerID;
```

5. Write an **INSERT** statement that inserts all the products into the **dbo.demoProduct** table from the **SalesLT.Product** table that have not already been inserted. Do not specify literal **ProductID** values in the statement.

   Here are two possible solutions:

```
INSERT INTO dbo.demoProduct (ProductID, Name, Color, StandardCost,
    ListPrice, Size, Weight)
SELECT p.ProductID, p.Name, p.Color, p.StandardCost, p.ListPrice,
    p.Size, p.Weight
FROM SalesLT.Product AS p
LEFT OUTER JOIN dbo.demoProduct AS dp ON p.ProductID = dp.ProductID
WHERE dp.ProductID IS NULL;

INSERT INTO dbo.demoProduct (ProductID, Name, Color, StandardCost,
    ListPrice, Size, Weight)
SELECT ProductID, Name, Color, StandardCost, ListPrice,
    Size, Weight
FROM SalesLT.Product
WHERE ProductID NOT IN (
    SELECT ProductID FROM dbo.demoProduct WHERE ProductID IS NOT NULL);
```

6. Write an **INSERT** statement that inserts all the addresses into the **dbo.demoAddress** table from the **SalesLT.Address** table. Before running the **INSERT** statement, type and run the command so that you can insert values into the **AddressID** column.

```
SET IDENTITY_INSERT dbo.demoAddress ON;

INSERT INTO dbo.demoAddress(AddressID,AddressLine1,AddressLine2,
    City,StateProvince,CountryRegion,PostalCode)
```

```
SELECT AddressID,AddressLine1,AddressLine2,
    City,StateProvince,CountryRegion,PostalCode
FROM SalesLT.Address;

--to turn the setting off
SET IDENTITY_INSERT dbo.demoAddress OFF;
```

Solutions to Exercise 6-2: Deleting Rows

Use the AdventureWorksLT2008 database to complete this exercise. Before starting the exercise, run code Listing 6-9 to re-create the demo tables.

1. Write a query that deletes the rows from the **dbo.demoCustomer** table where the **LastName** values begin with the letter *S*.

   ```
   DELETE FROM dbo.demoCustomer
   WHERE LastName LIKE 'S%'
   ```

2. Delete the rows from the **dbo.demoCustomer** table if the customer has not placed an order or if the sum of the **TotalDue** from the **dbo.demoSalesOrderHeader** table for the customer is less than $1,000.

   Here are two possible solutions:

   ```
   WITH Sales AS (
       SELECT C.CustomerID
       FROM dbo.demoCustomer AS C
       LEFT OUTER JOIN dbo.demoSalesOrderHeader AS SOH
       ON C.CustomerID = SOH.CustomerID
       GROUP BY c.CustomerID
       HAVING SUM(ISNULL(TotalDue,0)) < 1000)
   DELETE C
   FROM dbo.demoCustomer AS C
   INNER JOIN Sales ON C.CustomerID = Sales.CustomerID;

   DELETE FROM dbo.demoCustomer
   WHERE CustomerID IN (
       SELECT C.CustomerID
       FROM dbo.demoCustomer AS C
       LEFT OUTER JOIN dbo.demoSalesOrderHeader AS SOH
       ON C.CustomerID = SOH.CustomerID
       GROUP BY c.CustomerID
       HAVING SUM(ISNULL(TotalDue,0)) < 1000);
   ```

3. Delete the rows from the **dbo.demoProduct** table that have never been ordered.

   Here are two possible solutions:

```
DELETE P
FROM dbo.demoProduct AS P
LEFT OUTER JOIN dbo.demoSalesOrderDetail AS SOD ON P.ProductID = SOD.ProductID
WHERE SOD.ProductID IS NULL;

DELETE FROM dbo.demoProduct
WHERE ProductID NOT IN
    (SELECT ProductID
    FROM dbo.demoSalesOrderDetail
    WHERE ProductID IS NOT NULL);
```

Solutions to Exercise 6-3: Updating Existing Rows

Use the AdventureWorksLT2008 database to complete this exercise. Run the code in Listing 6-9 to re-create tables used in this exercise.

1. Write an **UPDATE** statement that changes all **NULL** values of the **AddressLine2** column in the **dbo.demoAddress** table to *N/A*.

```
UPDATE dbo.demoAddress SET AddressLine2 = 'N/A'
WHERE AddressLine2 IS NULL;
```

2. Write an **UPDATE** statement that increases the **ListPrice** of every product in the **dbo.demoProduct** table by 10 percent.

```
UPDATE dbo.demoProduct SET ListPrice *= 1.1;
```

3. Write an **UPDATE** statement that corrects the **UnitPrice** with the **ListPrice** of each row of the **dbo.demoSalesOrderDetail** table by joining the table on the **dbo.demoProduct** table.

```
UPDATE SOD
SET UnitPrice = P.ListPrice
FROM SalesLT.SalesOrderDetail AS SOD
INNER JOIN dbo.demoProduct AS P ON SOD.ProductID = P.ProductID;
```

4. Write an **UPDATE** statement that updates the **SubTotal** column of each row of the **dbo.demoSalesOrderHeader** table with the sum of the **LineTotal** column of the **dbo.demoSalesOrderDemo** table.

```
WITH SOD AS(
    SELECT SUM(LineTotal) AS TotalSum, SalesOrderID
    FROM dbo.demoSalesOrderDetail
    GROUP BY SalesOrderID)
UPDATE SOH Set SubTotal = TotalSum
FROM dbo.demoSalesOrderHeader AS SOH
INNER JOIN SOD ON SOH.SalesOrderID = SOD.SalesOrderID;
```

Solutions to Exercise 6-4: Using Transactions

Use the AdventureWorksLT2008 database to this exercise. Run the following script to create a table for this exercise:

```
IF OBJECT_ID('dbo.Demo') IS NOT NULL BEGIN
    DROP TABLE dbo.Demo;
END;
GO
CREATE TABLE dbo.Demo(ID INT PRIMARY KEY, Name VARCHAR(25));
```

1. Write a transaction that includes two insert statements to add two rows to the `dbo.Demo` table.

   Here's a possible solution:

   ```
   BEGIN TRAN
       INSERT INTO dbo.Demo(ID,Name)
       VALUES (1,'Test1');

       INSERT INTO dbo.Demo(ID,Name)
       VALUES(2,'Test2');
   COMMIT TRAN;
   ```

2. Write a transaction that includes two insert statements to add two more rows to the `dbo.Demo` table. Attempt to insert a letter instead of a number into the `ID` column in one of the statements. Select the data from the `dbo.Demo` table to see which rows made it into the table.

Here's a possible solution:

```
BEGIN TRAN
    INSERT INTO dbo.Demo(ID,Name)
    VALUES(3,'Test3');

    INSERT INTO dbo.Demo(ID,Name)
    VALUES('a','Test4');
COMMIT TRAN;
GO
SELECT ID,Name
FROM dbo.Demo;
```

Chapter 7: Understanding T-SQL Programming Logic

This section provides solutions to the exercises on understanding T-SQL programming logic.

Solutions to Exercise 7-1: Using Variables

Use the AdventureWorks2008 database to complete this exercise.

1. Write a script that declares an integer variable called **@myInt**. Assign 10 to the variable, and then print it.

   ```
   DECLARE @myInt INT = 10;
   PRINT @myInt;
   ```

2. Write a script that declares a **VARCHAR(20)** variable called **@myString**. Assign **This is a test** to the variable, and print it.

   ```
   DECLARE @myString VARCHAR(20) = 'This is a test';
   PRINT @myString;
   ```

3. Write a script that declares two integer variables called **@MaxID** and **@MinID**. Use the variables to print the highest and lowest **SalesOrderID** values from the **Sales.SalesOrderHeader** table.

   ```
   DECLARE @MaxID INT, @MinID INT;
   SELECT @MaxID = MAX(SalesOrderID),
       @MinID = MIN(SalesOrderID)
   FROM Sales.SalesOrderHeader;
   PRINT 'Max: ' + CONVERT(VARCHAR,@MaxID);
   PRINT 'Min: ' + CONVERT(VARCHAR, @MinID);
   ```

4. Write a script that declares an integer variable called **@ID**. Assign the value **70000** to the variable. Use the variable in a **SELECT** statement that returns all the **SalesOrderID** values from the **Sales.SalesOrderHeader** table that have a **SalesOrderID** greater than the value of the variable.

```
DECLARE @ID INTEGER = 70000;
SELECT SalesOrderID
FROM Sales.SalesOrderHeader
WHERE SalesOrderID > @ID;
```

5. Write a script that declares three variables, one integer variable called **@ID**, an **NVARCHAR(50)** variable called **@FirstName**, and an **NVARCHAR(50)** variable called **@LastName**. Use a **SELECT** statement to set the value of the variables with the row from the **Person.Person** table with **BusinessEntityID = 1**. Print a statement in the "BusinessEntityID: FirstName LastName" format.

```
DECLARE @ID INT, @FirstName NVARCHAR(50), @LastName NVARCHAR(50);
SELECT @ID = BusinessEntityID, @FirstName = FirstName,
    @LastName = LastName
FROM Person.Person
WHERE BusinessEntityID = 1;
PRINT CONVERT(NVARCHAR,@ID) + ': ' + @FirstName + ' ' + @LastName;
```

6. Write a script that declares an integer variable called **@SalesCount**. Set the value of the variable to the total count of sales in the **Sales.SalesOrderHeader** table. Use the variable in a **SELECT** statement that shows the difference between the **@SalesCount** and the count of sales by customer.

```
DECLARE @SalesCount INT;
SELECT @SalesCount = COUNT(*)
FROM Sales.SalesOrderHeader;

SELECT @SalesCount - COUNT(*) AS CustCountDiff, CustomerID
FROM Sales.SalesOrderHeader
GROUP BY CustomerID;
```

Solutions to Exercise 7-2: Using the IF…ELSE Construct

Use the AdventureWorks2008 database to complete this exercise.

1. Write a batch that declares an integer variable called **@Count** to save the count of all the **Sales.SalesOrderDetail** records. Add an **IF** block that that prints "Over 100,000" if the value exceeds 100,000. Otherwise, print "100,000 or less."

```
DECLARE @Count INT;
SELECT @Count = COUNT(*)
FROM Sales.SalesOrderDetail;
```

```
IF @Count > 100000 BEGIN
    PRINT 'Over 100,000';
END
ELSE BEGIN
    PRINT '100,000 or less.';
END;
```

2. Write a batch that contains nested **IF** blocks. The outer block should check to see whether the month is October or November. If that is the case, print "The month is " and the month name. The inner block should check to see whether the year is even or odd and print the result. You can modify the month to check to make sure the inner block fires.

```
IF MONTH(GETDATE()) IN (10,11) BEGIN
    PRINT 'The month is ' + DATENAME(mm,GETDATE());
    IF YEAR(GETDATE()) % 2 = 0 BEGIN
        PRINT 'The year is even.';
    END
    ELSE BEGIN
        PRINT 'The year is odd.';
    END
END;
```

3. Write a batch that uses **IF EXISTS** to check to see whether there is a row in the **Sales.SalesOrderHeader** table that has **SalesOrderID = 1**. Print "There is a SalesOrderID = 1" or "There is not a SalesOrderID = 1" depending on the result.

```
IF EXISTS(SELECT * FROM Sales.SalesOrderHeader
    WHERE SalesOrderID = 1) BEGIN
    PRINT 'There is a SalesOrderID = 1';
END
ELSE BEGIN
    PRINT 'There is not a SalesOrderID = 1';
END;
```

Solutions to Exercise 7-3: Using WHILE

Use the AdventureWorks2008 database to complete this exercise.

1. Write a script that contains a **WHILE** loop that prints out the letters *A* to *Z*. Use the function **CHAR** to change a number to a letter. Start the loop with the value 65.

Here is an example that uses the **CHAR** function:

```
DECLARE @Letter CHAR(1);
SET @Letter = CHAR(65);
PRINT @Letter;

DECLARE @Count INT = 65;
WHILE @Count < 91 BEGIN
    PRINT CHAR(@Count);
    SET @Count += 1;
END;
```

2. Write a script that contains a **WHILE** loop nested inside another **WHILE** loop. The counter for the outer loop should count up from 1 to 100. The counter for the inner loop should count up from 1 to 5. Print the product of the two counters inside the inner loop.

```
DECLARE @i INTEGER = 1;
DECLARE @j INTEGER;

WHILE @i <= 100 BEGIN
    SET @j = 1;
    WHILE @j <= 5 BEGIN
        PRINT @i * @j;
        SET @j += 1;
    END;
    SET @i += 1;
END;
```

3. Change the script in question 2 so the inner loop exits instead of printing when the counter for the outer loop is evenly divisible by 5.

```
DECLARE @i INTEGER = 1;
DECLARE @j INTEGER;

WHILE @i <= 100 BEGIN
    SET @j = 1;
    WHILE @j <= 5 BEGIN
        IF @i % 5 = 0 BEGIN
            PRINT 'Breaking out of loop.'
            BREAK;
        END;
        PRINT @i * @j;
        SET @j += 1;
    END;
```

```
    SET @i += 1;
END;
```

4. Write a script that contains a **WHILE** loop that counts up from 1 to 100. Print "Odd" or "Even" depending on the value of the counter.

```
DECLARE @Count INT = 1;
WHILE @Count <= 100 BEGIN
    IF @Count % 2 = 0 BEGIN
        PRINT 'Even';
    END
    ELSE BEGIN
        PRINT 'Odd';
    END
    SET @Count += 1;
END;
```

Solutions to Exercise 7-4: Handling Errors

Use AdventureWorks2008 to complete this exercise.

1. Write a statement that attempts to insert a duplicate row into the **HumanResources.Department** table. Use the **@@ERROR** function to display the error.

```
DECLARE @Error INT;
INSERT INTO HumanResources.Department(DepartmentID,Name,GroupName,ModifiedDate)
VALUES (1,'Engineering','Research and Development',GETDATE());
SET @Error = @@ERROR;
IF @Error > 0 BEGIN
    PRINT @Error;
END;
```

2. Change the code you wrote in question 1 to use **TRY…CATCH**. Display the error number, message, and severity.

```
BEGIN TRY
    INSERT INTO HumanResources.Department(DepartmentID,Name,GroupName,ModifiedDate)
    VALUES (1,'Engineering','Research and Development',GETDATE());
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber,ERROR_MESSAGE() AS ErrorMessage,
        ERROR_SEVERITY() AS ErrorSeverity;
END CATCH;
```

3. Change the code you wrote in question 2 to raise a custom error message instead of the actual error message.

```
BEGIN TRY
    INSERT INTO HumanResources.Department(DepartmentID,Name,GroupName,ModifiedDate)
    VALUES (1,'Engineering','Research and Development',GETDATE());
END TRY
BEGIN CATCH
    RAISERROR('You attempted to insert a duplicate!',16,1);
END CATCH;
```

Solutions to Exercise 7-5: Creating Temporary Tables and Table Variables

Use the AdventureWorks2008 database to complete this exercise.

1. Create a temp table called **#CustomerInfo** that contains **CustomerID**, **FirstName**, and **LastName** columns. Include **CountOfSales** and **SumOfTotalDue** columns. Populate the table with a query using the **Sales.Customer**, **Person.Person**, and **Sales.SalesOrderHeader** tables.

```
CREATE TABLE #CustomerInfo(
    CustomerID INT, FirstName VARCHAR(50),
    LastName VARCHAR(50),CountOfSales INT,
    SumOfTotalDue MONEY);
GO
INSERT INTO #CustomerInfo(CustomerID,FirstName,LastName,
    CountOfSales, SumOfTotalDue)
SELECT C.CustomerID, FirstName, LastName,COUNT(*),SUM(TotalDue)
FROM Sales.Customer AS C
INNER JOIN Person.Person AS P ON C.CustomerID = P.BusinessEntityID
INNER JOIN Sales.SalesOrderHeader AS SOH ON C.CustomerID = SOH.CustomerID
GROUP BY C.CustomerID, FirstName, LastName ;
```

2. Change the code written in question 1 to use a table variable instead of a temp table.

```
DECLARE @CustomerInfo TABLE (
    CustomerID INT, FirstName VARCHAR(50),
    LastName VARCHAR(50),CountOfSales INT,
    SumOfTotalDue MONEY);

INSERT INTO @CustomerInfo(CustomerID,FirstName,LastName,
    CountOfSales, SumOfTotalDue)
```

```
SELECT C.CustomerID, FirstName, LastName,COUNT(*),SUM(TotalDue)
FROM Sales.Customer AS C
INNER JOIN Person.Person AS P ON C.CustomerID = P.BusinessEntityID
INNER JOIN Sales.SalesOrderHeader AS SOH ON C.CustomerID = SOH.CustomerID
GROUP BY C.CustomerID, FirstName, LastName ;
```

3. Create a table variable with two integer columns, one of them an **INDENTITY** column. Use a **WHILE** loop to populate the table with 1,000 random integers using the following formula. Use a second **WHILE** loop to print the values from the table variable one by one.

```
CAST(RND() * 10000 AS INT) + 1
```

Here's a possible solution:

```
DECLARE @test TABLE (ID INTEGER NOT NULL IDENTITY, Random INT)
DECLARE @Count INT = 1;
DECLARE @Value INT;

WHILE @Count <= 1000 BEGIN
    SET @Value = CAST(RAND()*10000 AS INT) + 1;
    INSERT INTO @test(Random)
    VALUES(@Value);
    SET @Count += 1;
END;
SET @Count = 1;
WHILE @Count <= 1000 BEGIN
    SELECT @Value = Random
    FROM @test
    WHERE ID = @Count;
    PRINT @Value;
    SET @Count += 1;
END;
```

Chapter 8: Moving Logic to the Database

This section provides solutions to the exercises on moving logic to the database.

Solutions to Exercise 8-1: Creating Tables

Use the AdventureWorks2008 database to complete this exercise.

1. Create a table called **dbo.testCustomer**. Include a **CustomerID** that is an identity column primary key. Include **FirstName** and **LastName** columns. Include an **Age** column with a check constraint specifying that the value must be less than 120. Include an **Active** column that is one character with a default of **Y** and allows only **Y** or **N**. Add some rows to the table.

   Here's a possible solution:

```
IF OBJECT_ID ('dbo.testCustomer') IS NOT NULL BEGIN
    DROP TABLE dbo.testCustomer;
END;
GO

CREATE TABLE dbo.testCustomer (
    CustomerID INT NOT NULL IDENTITY PRIMARY KEY,
    FirstName VARCHAR(25), LastName VARCHAR(25),
    Age INT, Active CHAR(1) DEFAULT 'Y',
    CONSTRAINT ch_testCustomer_Age CHECK (Age < 120),
    CONSTRAINT ch_testCustomer_Active CHECK (Active IN ('Y','N'))
);
GO

INSERT INTO dbo.testCustomer(FirstName, LastName,Age)
VALUES ('Kathy','Morgan',35),('Lady B.','Kellenberger',14),
    ('Luke','Moore',30);
```

2. Create a table called **dbo.testOrder**. Include a **CustomerID** column that is a foreign key pointing to **dbo.testCustomer**. Include an **OrderID** column that is an identity column primary key. Include an **OrderDate** column that defaults to the current date and time. Include a **ROWVERSION** column. Add some rows to the table.

```
IF OBJECT_ID('dbo.testOrder') IS NOT NULL BEGIN
    DROP TABLE dbo.testOrder;
END;
GO
CREATE TABLE dbo.testOrder (CustomerID INT NOT NULL,
    OrderID INT NOT NULL IDENTITY PRIMARY KEY,
    OrderDate DATETIME DEFAULT GETDATE(),
    RW ROWVERSION,
    CONSTRAINT fk_testOrders FOREIGN KEY (CustomerID)
        REFERENCES dbo.testCustomer(CustomerID)
    );
GO
```

```
INSERT INTO dbo.testOrder (CustomerID)
VALUES (1),(2),(3);
```

3. Create a table called **dbo.testOrderDetail**. Include an **OrderID** column that is a foreign key pointing to **dbo.testOrder**. Include an integer **ItemID** column, a **Price** column, and a **Qty** column. The primary key should be a composite key composed of **OrderID** and **ItemID**. Create a computed column called **LineItemTotal** that multiplies **Price** times **Qty**. Add some rows to the table.

```
IF OBJECT_ID('dbo.testOrderDetail') IS NOT NULL BEGIN
    DROP TABLE dbo.testOrderDetail;
END;
GO
CREATE TABLE dbo.testOrderDetail(
    OrderID INT NOT NULL, ItemID INT NOT NULL,
    Price Money NOT NULL, Qty INT NOT NULL,
    LineItemTotal AS (Price * Qty),
    CONSTRAINT pk_testOrderDetail PRIMARY KEY (OrderID, ItemID),
    CONSTRAINT fk_testOrderDetail FOREIGN KEY (OrderID)
        REFERENCES dbo.testOrder(OrderID)
);

GO
INSERT INTO dbo.testOrderDetail(OrderID,ItemID,Price,Qty)
VALUES (1,1,10,5),(1,2,5,10);
```

Solutions to Exercise 8-2: Creating Views

Use the AdventureWorks2008 database to complete this exercise.

1. Create a view called **dbo.vw_Products** that displays a list of the products from the **Production.Product** table joined to the **Production.ProductCostHistory** table. Include columns that describe the product and show the cost history for each product. Test the view by creating a query that retrieves data from the view.

```
IF OBJECT_ID('dbo.vw_Products') IS NOT NULL BEGIN
    DROP VIEW dbo.vw_Products;
END;
GO
CREATE VIEW dbo.vw_Products AS (
    SELECT P.ProductID, P.Name, P.Color, P.Size, P.Style,
        H.StandardCost, H.EndDate, H.StartDate
    FROM Production.Product AS P
```

```
    INNER JOIN Production.ProductCostHistory AS H ON P.ProductID = H.ProductID
    );

GO
SELECT ProductID, Name, Color, Size, Style, StandardCost,
    EndDate, StartDate
FROM dbo.vw_Products;
```

2. Create a view called **dbo.vw_CustomerTotals** that displays the total sales from the **TotalDue** column per year and month for each customer. Test the view by creating a query that retrieves data from the view.

```
IF OBJECT_ID('dbo.vw_CustomerTotals') IS NOT NULL BEGIN
    DROP VIEW dbo.vw_CustomerTotals;
END;
GO
CREATE VIEW dbo.vw_CustomerTotals AS (
    SELECT C.CustomerID, YEAR(OrderDate) AS OrderYear,
        MONTH(OrderDate) AS OrderMonth, SUM(TotalDue) AS TotalSales
    FROM Sales.Customer AS C
    INNER JOIN Sales.SalesOrderHeader AS SOH ON C.CustomerID = SOH.CustomerID
    GROUP BY C.CustomerID, YEAR(OrderDate), MONTH(OrderDate)
    );
GO
SELECT CustomerID, OrderYear, OrderMonth, TotalSales
FROM dbo.vw_CustomerTotals;
```

Solutions to Exercise 8-3: Creating User-Defined Functions

Use the AdventureWorks2008 database to complete this exercise.

1. Create a user-defined function called **dbo.fn_AddTwoNumbers** that accepts two integer parameters. Return the value that is the sum of the two numbers. Test the function.

```
IF OBJECT_ID('dbo.fn_AddTwoNumbers') IS NOT NULL BEGIN
    DROP FUNCTION dbo.fn_AddTwoNumbers;
END;
GO

CREATE FUNCTION dbo.fn_AddTwoNumbers (@NumberOne INT, @NumberTwo INT)
RETURNS INT AS BEGIN
    RETURN @NumberOne + @NumberTwo;
END;
```

```
GO
SELECT dbo.fn_AddTwoNumbers(1,2);
```

2. Create a user-defined function called **dbo.Trim** that takes a **VARCHAR(250)** parameter. This function should trim off the spaces from both the beginning and the end of a string. Test the function.

```
IF OBJECT_ID('dbo.Trim') IS NOT NULL BEGIN
    DROP FUNCTION dbo.Trim;
END
GO
CREATE FUNCTION dbo.Trim (@Expression VARCHAR(250))
RETURNS VARCHAR(250) AS BEGIN
    RETURN LTRIM(RTRIM(@Expression));
END;
GO
SELECT '*' + dbo.Trim('  test  ') + '*';
```

3. Create a function called **dbo.fn_RemoveNumbers** that removes any numeric characters from a **VARHCHAR(250)** string. Test the function. Hint: The **ISNUMERIC** function checks to see whether a string is numeric. Check Books Online to see how to use it.

```
IF OBJECT_ID('dbo.fn_RemoveNumbers') IS NOT NULL BEGIN
     DROP FUNCTION dbo.fn_RemoveNumbers;
END;
GO
CREATE FUNCTION dbo.fn_RemoveNumbers (@Expression VARCHAR(250))
RETURNS VARCHAR(250) AS BEGIN
    DECLARE @NewExpression VARCHAR(250) = '';
    DECLARE @Count INT = 1;
    DECLARE @Char CHAR(1);
    WHILE @Count <= LEN(@Expression) BEGIN
        SET @Char = SUBSTRING(@Expression,@Count,1);
        IF ISNUMERIC(@Char) = 0 BEGIN
            SET @NewExpression += @Char;
        END
        SET @Count += 1;
    END;
RETURN @NewExpression;
END;
GO
SELECT dbo.fn_RemoveNumbers('abc 123 baby you and me');
```

4. Write a function called **dbo.fn_FormatPhone** that takes a string of ten numbers. The function will format the string into this phone number format: "(###) ###-####." Test the function.

```
IF OBJECT_ID('dbo.fn_FormatPhone') IS NOT NULL BEGIN
    DROP FUNCTION dbo.fn_FormatPhone;
END;
GO
CREATE FUNCTION dbo.fn_FormatPhone (@Phone VARCHAR(10))
RETURNS VARCHAR(14) AS BEGIN
    DECLARE @NewPhone VARCHAR(14);
    SET @NewPhone = '(' + SUBSTRING(@Phone,1,3) + ') ';
    SET @NewPhone = @NewPhone + SUBSTRING(@Phone,4,3) + '-';
    SET @NewPhone = @NewPhone + SUBSTRING(@Phone,7,4)
    RETURN @NewPhone;
END;
GO
SELECT dbo.fn_FormatPhone('5555551234');
```

Solutions to Exercise 8-4: Creating Stored Procedures

Use the AdventureWorks2008 database to complete this exercise.

1. Create a stored procedure called **dbo.usp_CustomerTotals** instead of the view from question 2 in Exercise 8-2. Test the stored procedure.

```
IF OBJECT_ID('dbo.usp_CustomerTotals') IS NOT NULL BEGIN
    DROP PROCEDURE dbo.usp_CustomerTotals;
END;
GO
CREATE PROCEDURE dbo.usp_CustomerTotals AS
    SELECT C.CustomerID, YEAR(OrderDate) AS OrderYear,
        MONTH(OrderDate) AS OrderMonth, SUM(TotalDue) AS TotalSales
    FROM Sales.Customer AS C
    INNER JOIN Sales.SalesOrderHeader AS SOH ON C.CustomerID = SOH.CustomerID
    GROUP BY C.CustomerID, YEAR(OrderDate), MONTH(OrderDate)

GO
EXEC dbo.usp_CustomerTotals;
```

2. Modify the stored procedure created in question 1 to include a parameter **@CustomerID**. Use the parameter in the **WHERE** clause of the query in the stored procedure. Test the stored procedure.

```
IF OBJECT_ID('dbo.usp_CustomerTotals') IS NOT NULL BEGIN
    DROP PROCEDURE dbo.usp_CustomerTotals;
END;
GO
CREATE PROCEDURE dbo.usp_CustomerTotals @CustomerID INT AS
    SELECT C.CustomerID, YEAR(OrderDate) AS OrderYear,
        MONTH(OrderDate) AS OrderMonth, SUM(TotalDue) AS TotalSales
    FROM Sales.Customer AS C
    INNER JOIN Sales.SalesOrderHeader AS SOH ON C.CustomerID = SOH.CustomerID
    WHERE C.CustomerID = @CustomerID
    GROUP BY C.CustomerID, YEAR(OrderDate), MONTH(OrderDate)

GO
EXEC dbo.usp_CustomerTotals 17910;
```

3. Create a stored procedure called **dbo.usp_ProductSales** that accepts a **ProductID** for a parameter and has an **OUTPUT** parameter that returns the number sold for the product. Test the stored procedure.

```
IF OBJECT_ID('dbo.usp_ProductSales') IS NOT NULL BEGIN
    DROP PROCEDURE dbo.usp_ProductSales;
END;
GO
CREATE PROCEDURE dbo.usp_ProductSales @ProductID INT,
    @TotalSold INT = NULL OUTPUT AS

    SELECT @TotalSold = SUM(OrderQty)
    FROM Sales.SalesOrderDetail
    WHERE ProductID = @ProductID;

GO
DECLARE @TotalSold INT;
EXEC dbo.usp_ProductSales @ProductID = 776, @TotalSold =  @TotalSold OUTPUT;
PRINT @TotalSold;
```

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

XYZ