Chapter 2

Database Management Systems

Chapter Outline

What You Will Learn in This Chapter

- What are databases and why are formal query systems necessary?
- How is data stored in a relational database?
- What is the basic structure of a query?
- How do you create a basic query?
- What types of computations can be performed in SQL?
- How are subtotals computed?
- How do you use multiple tables in a query?
- How are reports created in SQL Server?
- How do you create a new database?

FAA: Air Safety

For years, the Federal Aviation Administration (FAA) and major airlines have had teams dedicated to analyzing air crashes and using the results to improve air safety. The teams have been successful and their results have undoubtedly made flying safer. But, they have faced an interesting problem—with improved safety and fewer crashes, there is little opportunity for investigation. So, they have turned to analyzing data on all flights and pilot reports. As part of the process, the FAA and the airlines have changed their reporting culture—encouraging pilots and controllers to report all potential problems with no fear of blame. These "quality assurance programs" are designed to collect massive amounts of data to be analyzed for "precursors" or potential problems. Modern planes capture detailed flight data (not just the "black boxes" but ongoing reporting from instruments). For example, Southwest Airlines, which started collecting data in 2003, by 2008 had data on more than one million flights. Don Carter, senior manager of Southwest's flight safety program noted that "We are always asking ourselves, 'What should we be asking this data that we haven't thought of yet?'" [Wilber 2008] US Airways used similar data to identify an unusually high number of "unstabilized" approaches where the planes come in too fast or sink too quickly at the last stages of landing. The carrier changed its training and landing checklists and reduced the rate of unstable approaches by more than 70 percent. They used similar data to rewrite charts to improve landings and visibility at McCarren International airport in Las Vegas. Tom Lulkovich, US Airways director of flight safety noted that "everything is about identifying risk here."

Collecting and organizing huge amounts of data is an important first step. Analysts must also know which questions to ask.

Del Quentin Wilber, "Avoiding Plane Crashes By Crunching Numbers," *The Washington Post*, January 13, 2008. http://www.washingtonpost.com/wp-dyn/content/article/2008/01/12/AR2008011202407.html

Introduction

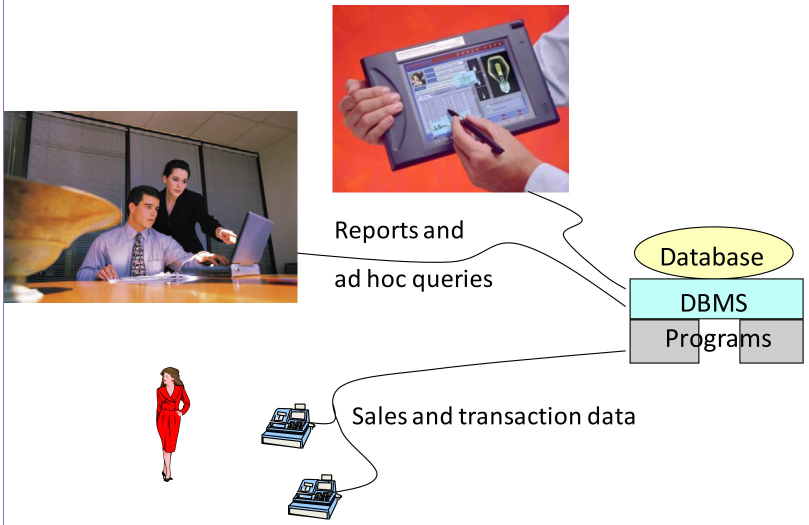
What are databases and why are formal query systems necessary? Most companies use a relational **database management system (DBMS)** to hold transaction data. These databases form the foundation for applications such as accounting systems and Web servers. Many companies have integrated applications or **enterprise resource planning (ERP)** systems that store all data in a DBMS. These transaction databases are designed to be efficient at storing data, and to ensure that the data remains safe even when multiple operations are performed at the same time and even if power fails half way through a process.

A **database** is a collection of related data. The database is typically stored in specific formats and controlled with a DBMS which provides access to the data through standardized connection methods. Without databases, programmers store data in proprietary structures so the data is available only to a specific program. For example, word processing documents are commonly stored in proprietary formats. It usually works for word documents because most people are willing to use the same word processor to edit a document. However, it is critical for basic business data to be accessible to other programs. Most databases have a relatively standard method to provide data to people and to other software. This process usually involves a **query system**, and most existing query systems are based on the **SQL** standard. Following standards is important because it makes it easier to change the underlying DBMS if necessary, and it is easier for programmers and managers to learn a single query method that can be used in most situations.

Figure 2.1 shows the basic roles of the database and DBMS. The DBMS is software that controls access to the database, supports programs, creates reports for standardized data, and has a query language to process ad hoc queries.

Figure 2.1

Database management system. The DBMS stores the database and provides access to programs, creates reports, and processes ad hoc queries.



Most major DBMS vendors have implemented a "natural language" query system at some point. Most have abandoned the work in favor of the SQL standard. The problem with natural languages, such as English, is that they are inherently ambiguous. The risk is relatively high that the computer will not understand a question the same way it was intended by the manager asking the question. The DBMS does the best it can to produce results, and it is difficult for the manager to verify that the computer had the correct understanding of the question. Versions of this problem exist with any query language, but formal systems are defined to work in a specific way. By learning the basic rules of the formal query system, it is possible to ensure that the DBMS returns exactly the data needed to answer a specific question. However, you must learn those basic rules.

The main goal of this chapter is to show how to create common queries in SQL Server. Queries are useful for answering ad hoc questions. In some ways, data mining tools covered in Chapter 3 are easier to use than traditional SQL queries. However, queries are still used to look up basic facts and they are used to configure data for some of the data mining tools. Queries and data warehouses are complementary—a good analyst needs to be able to use both tools. Queries provide direct access to data and support row-by-row computations. Data warehouse tools make it easy to summarize data and compare subtotals and explore data.

Relational Databases

How is data stored in a relational database? To learn to write queries to retrieve data, it is important to understand how data is stored in the database. The most important thing to know is that all data in a relational database is stored in tables. To answer business questions or configure data for analysis, it is necessary to determine which tables hold the data needed. This section introduces the basic concepts of tables, and assumes that the tables and data already exist. It does not attempt to explain how to create tables. Relational databases have to be carefully designed or they do not work well. These design issues are briefly covered in a later section of this chapter, but learning how to design a database is covered in other textbooks (Post 2011).

Tables

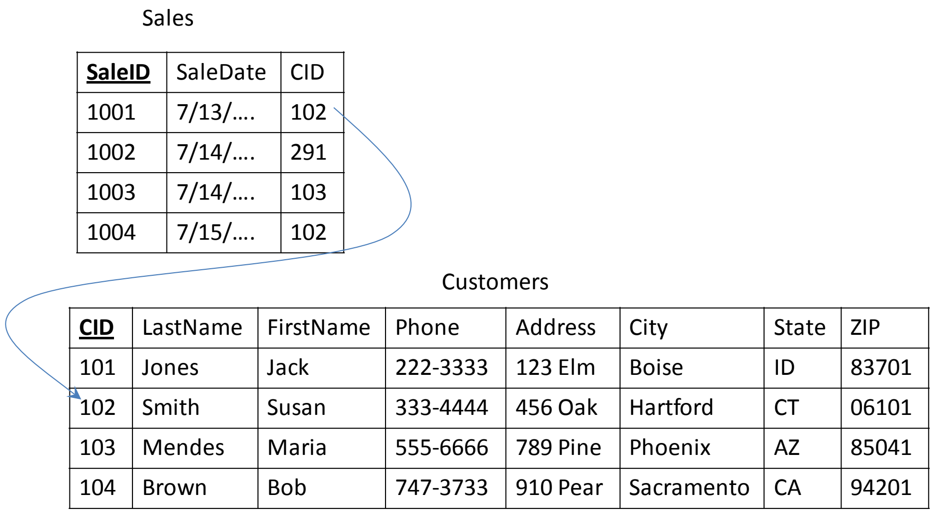
A **table** describes a single concept or object. Its **columns** consist of attributes or properties of the object. Data is stored as a row—where each row represents one instance of data. Figure 2.2 shows part of a table for Customers. Customers are

Figure 2.2

Sample table: Customers. A table represents a single object or event. Columns are properties or attributes that describe the object. Each row is one instance of data.

primary key: Identify a row Columns: Properties/Attributes

CID	LastName	FirstName	Phone	Address	City	State	ZIP
101	Jones	Jack	222-3333	123 Elm	Boise	ID	83701
102	Smith	Susan	333-4444	456 Oak	Hartford	CT	6101
103	Mendes	Maria	555-6666	789 Pine	Phoenix	AZ	85041
104	Brown	Bob	747-3733	910 Pear	Sacramento	CA	94201



Sales

SaleID	SaleDate	CID
1001	7/13/….	102
1002	7/14/….	291
1003	7/14/….	103
1004	7/15/….	102

Customers

CID	LastName	FirstName	Phone	Address	City	State	ZIP
101	Jones	Jack	222-3333	123 Elm	Boise	ID	83701
102	Smith	Susan	333-4444	456 Oak	Hartford	CT	06101
103	Mendes	Maria	555-6666	789 Pine	Phoenix	AZ	85041
104	Brown	Bob	747-3733	910 Pear	Sacramento	CA	94201

Figure 2.3

Data keys link tables. The Sales table holds only the CID value to indicate which customer made the purchase. Details about the customer are stored in one location in the Customers table.

defined in terms of their name, phone, address, city, state, and ZIP or postal code. Most companies collect additional properties about customers, such as e-mail address. Each row represents a single customer, and all data for that customer is stored in that row. The row is identified by the CustomerID (CID) value—it is the primary key for the table. A **primary key** is a column or set of columns that uniquely identifies a row. Every table must have a primary key. In cases of simple objects, the key is usually created within the DBMS. For example, SQL Server uses an **identity** to generate new key values that are guaranteed to be unique. In the example of the Customers, if the ID value 104 is provided, the DBMS can quickly find the data for customer Brown.

Primary keys are often used to link tables. Figure 2.3 illustrates the process with a Sales table. The Sales table holds a value for CID to indicate which customer made the purchase. Details about the customer are stored in a single row in the Customers table with the matching CID value (102). The main benefit to this approach is that data for each customer is stored in only one location—making it easy to find and easy to change. The drawback is that the DBMS needs a method to quickly match data from multiple tables.

Relational databases can have any number of tables—all related through the key values. Corporate databases can easily include hundreds or even thousands of tables. Creating queries to answer business questions ultimately requires that you know which table to use—which requires knowing exactly what each table represents and knowing the meaning of each column. Figure 2.4 shows the relationship diagram for Rolling Thunder Bicycles. Even this relatively basic set of tables is difficult to fit on a single page. Most business queries focus on a few basic tables, such as Bicycle, Customer, City, Employee, BikeParts, Components, Manufacturer, PurchaseOrder, and PurchaseItem. Some tables, such as CustomerTrans, ManufacturerTrans, ModelSize, and BikeTubes are used for internal accounting

Figure 2.4

Relationship diagram for Rolling Thunder Bicycles. Creating queries requires understanding all of the tables and columns, although most business questions focus on a few key areas.

and manufacturing purposes and are rarely queried directly. But, it takes knowledge of the industry and the company to recognize the purpose of each table. One of the first steps in analyzing data is to understand exactly what data is available. Looking through the list of tables provides a first glimpse of that data.

Data Types

Computers deal only with binary data, so all data must be assigned a **data type** that specifies how the data is stored and handled. The basic data types are: Numeric, Text, Date, XML, and binary Object, such as pictures. The catch is that many subtypes exist. Figure 2.5 lists the basic data types in SQL Server. First, note that all text today should use the Unicode ("National") data type—which supports characters in multiple languages. Second, the appropriate data type should be chosen for each data column. Integer values cannot contain decimal data and the size chosen must be able to hold the largest possible value. Monetary values should be stored using the money type, never integers and never float or real which can round off some values. Dates must always be stored with the datetime data type because it supports searches and subtraction of dates to find the number of days between two dates.

Data Type	SQL Server	Size
Text		
Fixed	char	8K
Variable	varchar	8K
Unicode	nchar, nvarchar	4K
Memo	text	2G, 1G
XML	xml	2G
Number		
Byte	tinyint	255
Integer	smallint	+/- 32767
Long	int	+/-2B
64-bits	bigint	18 digits
Fixed precision	decimal(p,s)	p: 1 to 38
Float	real	+/- 1E 38
Double	float	+/- 1E 308
Currency	money	+/- 900.0000 trillion (8 bytes)
Yes/No	bit	0/1
Date/Time	datetime	1/1/1753 – 12/31/9999 (3 ms)
	smalldatetime	1/1/1900 – 6/6/2079 (1 min)
Interval	interval year, …	
Image	image	2GB

Figure 2.5

Data types in SQL Server. Use Unicode whenever possible. Be careful with number types to ensure the correct type is chosen for each piece of data.

In most cases, the database structure and data types will already exist for any data that managers use. However, managers might be involved in the design and construction of a new data warehouse that consolidates data from other sources. Hence, it is helpful for managers to know the types of data available to hold the consolidated data.

The binary data types are rarely used in data mining. Data mining has been successfully applied to images and large-text files, but these investigations typically use specialized tools and data storage. Relational databases are generally too slow to use for large collections of huge text and image files.

Extensible markup language (XML) files are increasingly common in business, but they present additional complications in data mining. XML data consists of text that is tagged; similar to HTML tags, but any tag names can be used. The drawback to XML is that multiple types and levels of data are stored within a single XML list. Figure 2.6 provides an example of an XML segment used for an order. To analyze this data, the individual elements must be extracted and stored into new tables. Data mining tools, particularly those in SQL Server, are designed to work with relational tables. It would be too slow to extract data from XML segments on the fly. SQL Server supports XQuery and other XML tools used to extract data from XML files, but these tools need to be run once during the setup to unload the data from XML and transfer it into standard tables.

```
<order>
	<orderID>111</orderID>
	<customer>
		<cID>99</cID>
		<lastName>Smith</lastName><firstName>Mary</firstName>
	</customer>
	<itemList>
		<item>
			<itemid>290</itemid><description>red dress</description>
			<salePrice>132.99</salePrice><quantity>1</quantity>
		</item>
			<itemid>171</itemid><description>shoes</description>
			<salePrice>79.89</salePrice><quantity>1</quantity>
		<item>
		</item>
	</itemList>
</order>
```

Figure 2.6

Sample XML data. To analyze this data the individual elements must be extracted and stored into separate tables and rows.

Four Questions to Retrieve Data

What is the basic structure of a query? Every attempt to retrieve data from a relational DBMS requires answering the four basic questions listed in Figure 2.7. The difference among query systems is how you fill in those answers. You need to remember these four questions, but do not worry about the specific order. When you first learn to create queries, you should write down these four questions each time you construct a query. With easy problems, you can almost automatically fill in answers to these questions. With more complex problems, you might fill in partial answers and switch between questions until you completely understand the query.

Notice that in some easy situations you will not have to answer all four questions. Many easy questions involve only one table, so you will not have to worry about joining tables (question 4). As another example, you might want the total sales for the entire company, as opposed to the total sales for a particular employee, so there may not be any constraints (question 2).

Figure 2.7

Four questions to create a query. Every query is built by asking these four questions.

What output do you want to see?
What do you already know (or what constraints are given)?
What tables are involved?
How are the tables joined?

What Output Do You Want to See?

In many ways, this question is the most important. Obviously, the database engine needs to know what you want to see. More importantly, you first have to visualize your output before you can write the rest of the query. In general, a query system answers your query by displaying rows of data for various columns. You have to tell the DBMS which columns to display. However, you can also ask the DBMS to perform some basic computations, so you also need to identify any calculations and totals you need.

You generally answer this question by selecting columns of data from the various tables stored in the database. Of course, you need to know the names of all of the columns to answer this question. Generally, the hardest part in answering this question is to wade through the list of tables and identify the columns you really want to see. The problem is more difficult when the database has hundreds of tables and thousands of columns. Queries are easier to build if you have a copy of the class diagram that lists the tables, their columns, and the relationships that join the tables.

What Do You Already Know?

In most situations you want to restrict your search based on various criteria. For instance, you might be interested in sales on a particular date or sales from only one department. The search conditions must be converted into a standard Boolean notation (phrases connected with AND or OR). The most important part of this step is to write down all the conditions to help you understand the purpose of the query.

What Tables Are Involved?

With only a few tables, this question is easy. With hundreds of tables, it could take a while to determine exactly which ones you need. A good data dictionary with synonyms and comments will make it easier for you (and users) to determine exactly which tables you need for the query. It is also critical that tables be given names that accurately reflect their content and purpose.

One hint in choosing tables is to start with the tables containing the columns listed in the first two questions (output and criteria). Next decide whether other tables might be needed to serve as intermediaries to connect these tables.

How Are the Tables Joined?

In a relational database, tables are connected by data in similar columns. For instance, as shown in Figure 2.3, a Sales table has a CustomerID column. Corresponding data is stored in the Customer table, which also has a CustomerID column. In many cases matching columns in the tables will have the same name (e.g., CustomerID) and this question is easy to answer. The join performs a matching or lookup for the rows. You can think of the result as one giant table and use any of the columns from any of the joined tables. Note that columns are not required to have the same name, so you sometimes have to think a little more carefully. For example, an Order table might have a column for SalesPerson, which is designed to match the EmployeeID key in an Employee table.

Joining tables is usually straightforward as long as the database design is sound. In fact, the query system will automatically use the design to join any tables whenever possible. However, two problems can arise in practice: (1) You should verify that all tables are joined, and (2) Double-check any tables with multiple join conditions.

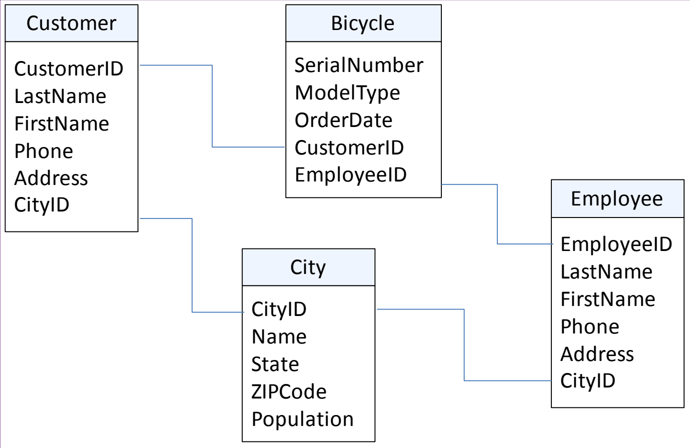


Figure 2.8

Loops with joins usually cause problems. This sample query would return customers ONLY if they live in the same city as the employee who placed the order. Delete the connection from Employee and City to solve the problem.

Technically, it is legal to use tables without adding a join condition. However, when no join condition is explicitly specified, the DBMS creates a **cross join** or Cartesian product between the tables. A cross join matches every row in the first table to every other row in the second table. For example, if both tables have 10 rows, the resulting cross join yields 10*10 = 100 rows of data. If the tables each have 1,000 rows, the resulting join has one million rows! A cross join will seriously degrade performance on any DBMS, so be sure to specify a join condition for every table.

Sometimes table designs have multiple relationship connections between tables. For example, Figure 2.8 shows that the Rolling Thunder Bicycles database joins Customer to City and City to Employee. If a query is built that includes the tables: Customer, City, Bicycle, and Employee; the query builder will automatically include a join relationship between Customer and City as well as a join between Employee and City. Including both joins causes a problem—only data that meets both conditions will be displayed. In the Customer/Employee example, Customers will be returned only if they live in the same city as the employee who sold the bicycle. This query is rarely going to be useful. The solution is to remove the join between Employee and City.

Query Basics

How do you create a basic query? It is best to begin with relatively easy queries. This section presents queries that involve a single table to show the basics of creating a query. Then it covers details on constraints, followed by a discussion on computations and aggregations. Groups and subtotals are then explained.

Figure 2.9 presents several business questions that might arise at the Rolling Thunder Bicycles company. Most of the questions are relatively easy to answer. With a small enough number of rows, it might be possible to hand-search the Bicycle table to find answers. However, the point of this section is to start with relatively easy queries to focus on the basics of creating queries and entering conditions.

- Which bicycles were ordered between 12/1/2008 and 12/15/2008?
- Which race bicycles in 2008 were larger than 61 cm?
- Which race bicycles in 2007 had a list price over 7000?
- In December 2008, which mountain or mountain full suspension bicycles sold for more than 5500?
- What is the total value of all bikes sold in December 2008?
- What it the total value of items on purchase order 101?
- How many Race bikes were sold in November 2008?
- What is the number of bicycles sold of each model type in November 2008?
- Who is the best sales person?
- List the CustomerID of everyone who bought a bicycle on December 1, 2008.
- List Names and Phone numbers for everyone who purchased a bike on 01-DEC-2008.
- List customers from Miami, FL who purchased bicycles in December 2008.
- Which model types were not sold in December 2008?

Figure 2.9

Sample questions for Rolling Thunder Bicycles. The essence of building a query is to convert the business question into the structure required by the DBMS.

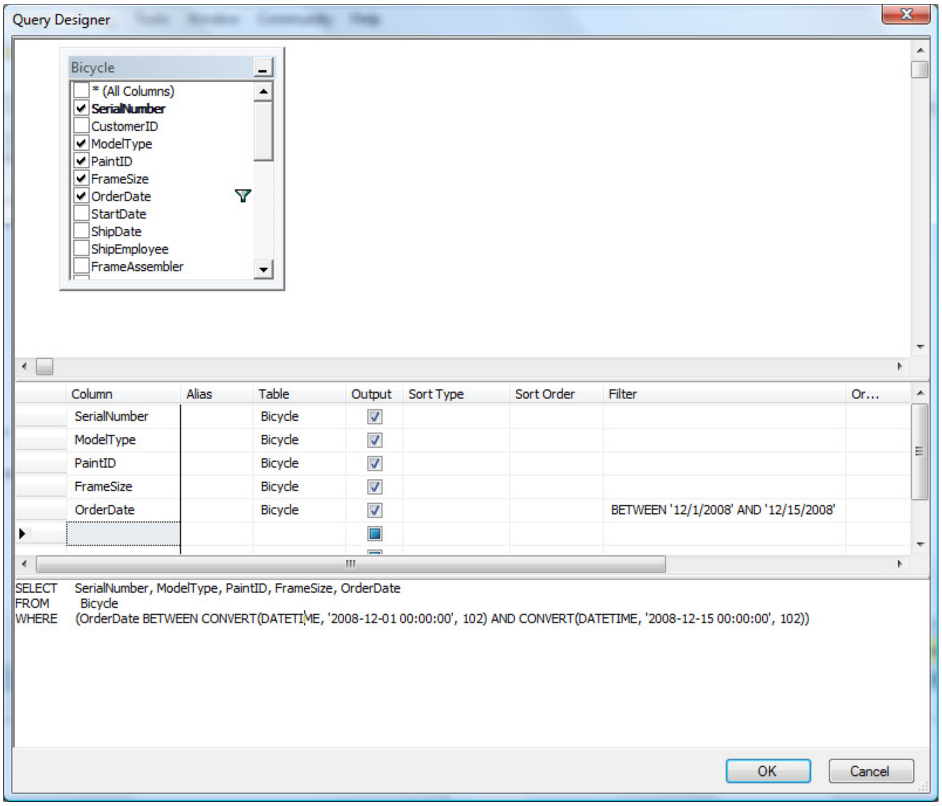
The foundation of queries is that you want to see only some of the columns from a table and that you want to restrict the output to a set of rows that match some criteria. For example, in the first query (animals with yellow color), you might want to see the AnimalID, Category, Breed, and their Color. Instead of listing every animal in the table, you want to restrict the list to just those with a yellow color.

As with most tools today, SQL Server has a query editor to help build queries visually. Ultimately, queries are written in SQL. SQL Server shows the SQL statement in the editor and you can switch back and forth between the two. In many cases, the SQL is easier to read, but the visual editor can be used to create queries with less typing. The query editor can be run through the SQL Server Management Studio. Chapter 3 also shows how the query editor can be used from within the analysis services. Start the Management Studio and log in. Expand the list of databases and right-click the RT database to select the New Query menu item. If the "New Query" button is used on the main menu, you must always start a query with the line: *USE RT;* to ensure the proper database is used. By default, the query editor is set for SQL. Right click and choose the "Design query in Editor" menu option to open the visual designer.

Single Tables

The first query to consider is: *Which bicycles were ordered between 12/1/2008 and 12/15/2008?* Figure 2.10 shows the design editor and the SQL. The two methods utilize the same underlying structure. The designer approach saves some typing, but eventually you need to be able to write the SQL statements. If you write down the SQL keywords, you can fill in the blanks—similar to the way you fill in the designer grid.

The designer will ask you to choose the tables involved. This question involves only one table: Bicycles. You know that because all of the data you want to see and the constraint are based on columns in the Bicycle table. With the table displayed, you can now choose which columns you want to see in the output. The



Column	Alias	Table	Output	Sort Type	Sort Order	Filter	Or...
SerialNumber		Bicycle	✓				
ModelType		Bicycle	✓				
PaintID		Bicycle	✓				
FrameSize		Bicycle	✓				
OrderDate		Bicycle	✓			BETWEEN '12/1/2008' AND '12/15/2008'	

```
SELECT   SerialNumber, ModelType, PaintID, FrameSize, OrderDate
FROM     Bicycle
WHERE    (OrderDate BETWEEN CONVERT(DATETIME, '2008-12-01 00:00:00', 102) AND CONVERT(DATETIME, '2008-12-15 00:00:00', 102))
```

Figure 2.10

Sample query shown in QBE and SQL. Since there is only one table, only three questions need to be answered: What tables? What conditions? What do you want to see?

business question is a little vague, so select SerialNumber, ModelType, PaintId, FrameSize, and OrderDate.

The next step is to enter the criteria that you already know. In this example, you are looking for bicycles ordered between two specific dates. Date conditions can be entered with standard comparison operators (<, >, >=, and so on). However, dates are usually given as a range and ranges are easiest to enter with a BETWEEN condition. On the same row as the order date, scroll to the right, in the filter column enter: BETWEEN ‘12/1/2008’ AND ‘12/15/2008’. Dates must be enclosed in single parentheses and entered in the data format set for the local computer. Dates can also be entered as ’01-DEC-2008’ to avoid confusion between month and day numbers. The SQL statement uses the CONVERT function to explicitly define the date format. Click the OK button to close the designer. Click the “! Execute” button to run the query and see the bicycles that match the date condition.

The four basic questions are answered by filling out blanks on the query design grid. (1) The output to be displayed is placed as a field on the grid. (2) The constraints are entered as criteria or conditions under the appropriate fields. (3) The tables involved are displayed at the top (and often under each field name). (4) The

SELECT	columns	What do you want to see?
FROM	tables	What tables are involved?
JOIN	conditions	How are the tables joined?
WHERE	criteria	What are the constraints?

Figure 2.11

The basic SQL SELECT command matches the four questions you need to create a query. The uppercase letters are used in this text to highlight the SQL keywords. They can also be typed in lowercase.

table joins are shown as connecting lines among the tables. The one drawback to query design systems is that you have to answer the most difficult question first: Identifying the tables involved. The query design system uses the table list to provide a list of the columns you can choose. Keep in mind that you can always add more tables as you work on the problem.

Introduction to SQL

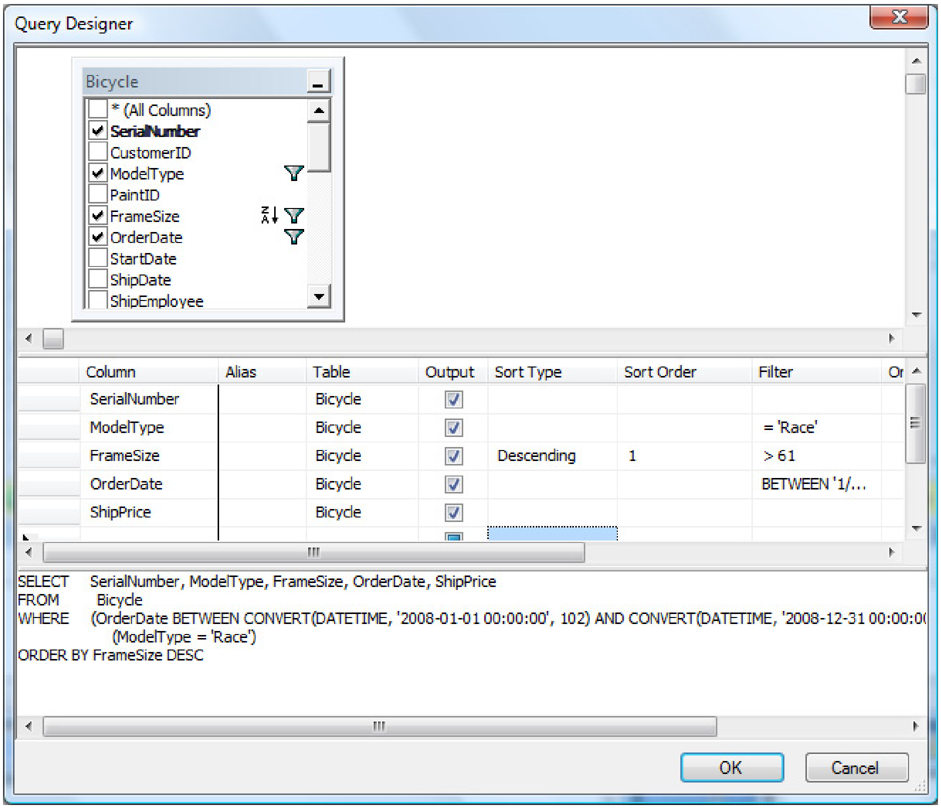
SQL is a powerful query language. However, unlike the design editor, you generally have to type in the entire statement. Perhaps the greatest strength of SQL is that it is a standard that most vendors of DBMS software support. Hence, once you learn the base language, you will be able to create queries on all of the major systems in use today. Note that some people pronounce SQL as “sequel,” partly treating it as a descendant of vendor’s early DBMS called quel. Also, “Sequel” is easier to say than “ess-cue-el.”

The most commonly used command in SQL is the SELECT statement, which is used to retrieve data from tables. A simple version of the command is shown in Figure 2.11, which contains the four basic parts: **SELECT**, **FROM**, **JOIN**, and **WHERE**. These parts match the basic questions needed by every query. In the example in Figure 2.11, notice the similarity between the editor and SQL approaches. The four basic questions are answered by entering items after each of the four main keywords. When you write SQL statements, it is best to write down the keywords and then fill in the blanks. You can start by listing the columns you want to see as output, then write the constraints in the WHERE clause. By looking at the columns you used, it is straightforward to identify the tables involved. You can use the class diagram to understand how the tables are joined.

Sorting the Output

Database systems treat tables as collections of data. For efficiency the DBMS is free to store the table data in any manner or any order that it chooses. Yet in most cases you will want to display the results of a query in a particular order. The SQL **ORDER BY** clause is an easy and fast means to display the output in any order you choose. As shown in Figure 2.12, simply list the columns you want to sort. The default is ascending (A to Z or low to high with numbers). Add the phrase **DESC** (for descending) after a column to sort from high to low. In QBE you select the sort order on the QBE grid.

In some cases you will want to sort columns that do not contain unique data. For example, many customers will have the same last name (such as Smith). In



Column	Alias	Table	Output	Sort Type	Sort Order	Filter	Or
SerialNumber		Bicycle	✓				
ModelType		Bicycle	✓			= 'Race'	
FrameSize		Bicycle	✓	Descending	1	> 61	
OrderDate		Bicycle	✓			BETWEEN '1/...	
ShipPrice		Bicycle	✓				

```
SELECT   SerialNumber, ModelType, FrameSize, OrderDate, ShipPrice
FROM     Bicycle
WHERE    (OrderDate BETWEEN CONVERT(DATETIME, '2008-01-01 00:00:00', 102) AND CONVERT(DATETIME, '2008-12-31 00:00:0
           (ModelType = 'Race')
ORDER BY FrameSize DESC
```

Figure 2.12

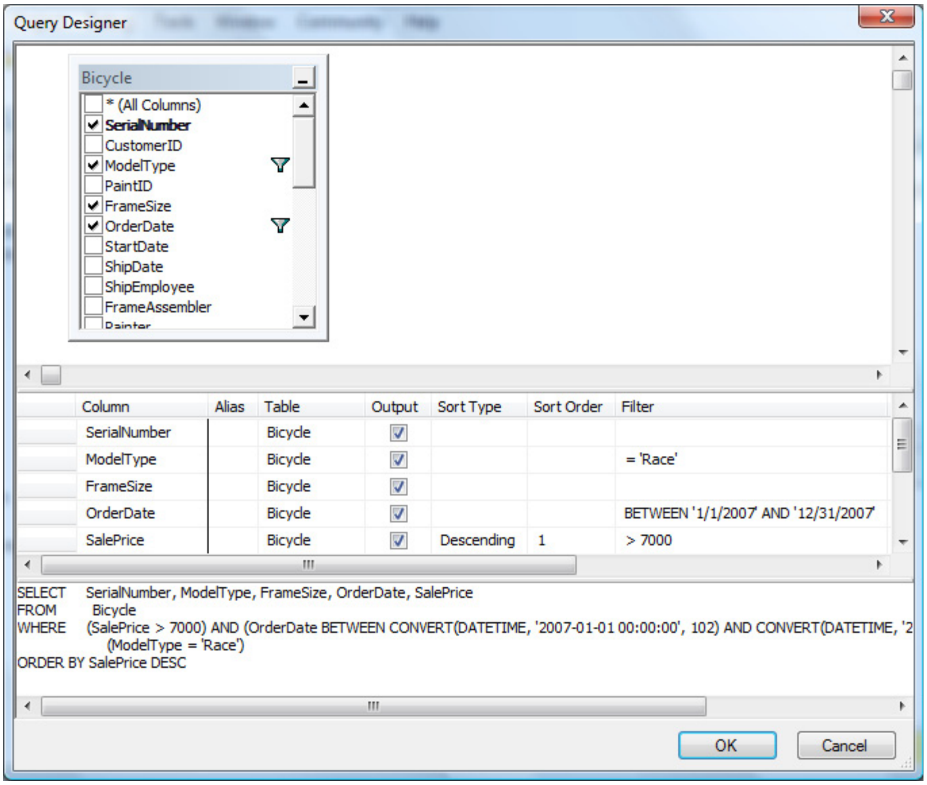
The ORDER BY clause sorts the output rows. The default is to sort in ascending order, adding the keyword DESC after a column name results in a descending sort. When columns like Category contain duplicate data, use a second column (e.g., Breed) to sort the rows within each category.

these cases, a secondary sort column is usually added. Names are generally sorted by LastName and then FirstName. So all customers with the last name Smith will be subsorted by the first name. This additional column is only used if the rows in the prior column contain identical data. In SQL, simply listed the columns left to right in the order to be sorted, such as ORDER BY LastName, FirstName. In the design editor, the priority is given by an index in the Sort Order column..

Criteria

In most questions, identifying the output columns and the tables is straightforward. If there are hundreds of tables, it might take a while to decide exactly which tables and columns are needed, but it is just an issue of perseverance. On the other hand, identifying constraints and specifying them correctly can be more challenging. More importantly if you make a mistake on a constraint, you will still get a result. The problem is that it will not be the answer to the question you asked—and it is often difficult to see that you made a mistake.

The primary concept of constraints is based on **Boolean algebra**, which you learned in mathematics. In practice, the term simply means that various conditions



Column	Alias	Table	Output	Sort Type	Sort Order	Filter
SerialNumber		Bicycle	✓			
ModelType		Bicycle	✓			= 'Race'
FrameSize		Bicycle	✓			
OrderDate		Bicycle	✓			BETWEEN '1/1/2007' AND '12/31/2007'
SalePrice		Bicycle	✓	Descending	1	> 7000

```
SELECT    SerialNumber, ModelType, FrameSize, OrderDate, SalePrice
FROM      Bicycle
WHERE     (SalePrice > 7000) AND (OrderDate BETWEEN CONVERT(DATETIME, '2007-01-01 00:00:00', 102) AND CONVERT(DATETIME, '2
            (ModelType = 'Race')
ORDER BY SalePrice DESC
```

Figure 2.13

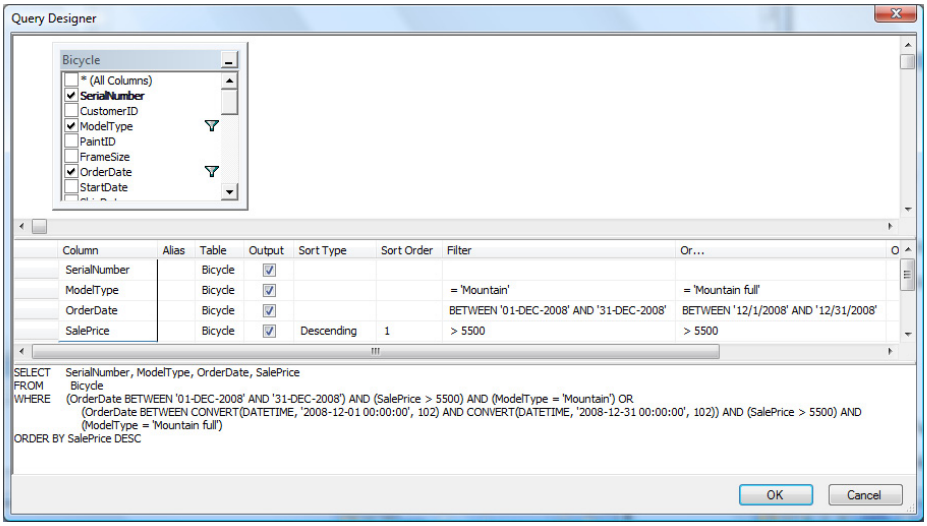
Criteria with AND connectors. Which race bicycles sold for more than 7000 in 2007?

are connected with AND and OR clauses. Sometimes you will also use a **NOT** statement, which negates or reverses the truth of the statement that follows it. For example, NOT (ModelType = 'Race') means you are interested in all models except Race.

Consider the example in Figure 2.13. The first step is to note that three conditions define the business question: date, model type, and price. The second step is to recognize that all of these conditions need to be true at the same time, so they are connected by AND. As the database system examines each row, it evaluates all three clauses. If any one clause is false, the row is skipped.

Notice that the SQL statement is straightforward—just write the three conditions and connect them with an AND clause. The designer is a little trickier. Every condition listed in the same filter column is connected with an AND clause. Conditions in different filter columns are joined with an OR clause. You have to be careful creating (and reading) designer statements, particularly when there are many different criteria columns.

Consider an example with an "OR" connector. In December 2008, which mountain or mountain full suspension bicycles sold for more than 5500? Figure 2.14 shows one way to build the query in the designer. The two ModelType conditions can be written in separate filter columns which creates the OR condition. However, with this approach, the date and price conditions have to be duplicated



Column	Alias	Table	Output	Sort Type	Sort Order	Filter	Or...
SerialNumber		Bicycle	✓				
ModelType		Bicycle	✓			= 'Mountain'	= 'Mountain full'
OrderDate		Bicycle	✓			BETWEEN '01-DEC-2008' AND '31-DEC-2008'	BETWEEN '12/1/2008' AND '12/31/2008'
SalePrice		Bicycle	✓	Descending	1	> 5500	> 5500

```
SELECT    SerialNumber, ModelType, OrderDate, SalePrice
FROM       Bicycle
WHERE     (OrderDate BETWEEN '01-DEC-2008' AND '31-DEC-2008') AND (SalePrice > 5500) AND (ModelType = 'Mountain') OR
               (OrderDate BETWEEN CONVERT(DATETIME, '2008-12-01 00:00:00', 102) AND CONVERT(DATETIME, '2008-12-31 00:00:00', 102)) AND (SalePrice > 5500) AND
               (ModelType = 'Mountain full')
ORDER BY SalePrice DESC
```

Figure 2.14

Criteria with OR connectors. In December 2008, which mountain or mountain full suspension bicycles sold for more than 5500? Or conditions (Mountain or Mountain full) can be written in different filter columns. But the other conditions have to be copied along as well.

and listed in both filter columns. Each filter column is treated as a separate list of conditions. All conditions within that column are connected with AND statements. If you leave out the date and price conditions on the second filter column, it will match all full suspension bicycles regardless of date or price. In all cases, test the query! Run it and check the rows to ensure they meet all of the business conditions.

Figure 2.15 shows a way to simplify the query using SQL. Enter the ModelType condition with the OR connector in parentheses. Then add the date and SalePrice conditions and insert the AND connectors. By using parentheses to isolate the OR conditions, the other statements can be listed one time. Switch to the design editor to see how the editor handles this change. In the main query editor, highlight the entire query and then switch to the design editor. The ModelType statement with the OR connector is listed completely in a single filter cell. For

Figure 2.15

SQL criteria with OR connectors. Parentheses are used to isolate the OR condition for Mountain or Mountain full model types. Then the AND conditions are listed only once. Convert to design view to see how it is handled in the editor.

```
SELECT    SerialNumber, ModelType, OrderDate, SalePrice
FROM      Bicycle
WHERE     (OrderDate BETWEEN '01-DEC-2008' AND '31-DEC-2008')
                  AND (SalePrice >5500)
                  AND (ModelType='Mountain' OR ModelType='Mountain full')
ORDER BY SalePrice DESC;
```

Comparisons	Examples
Operators	<, =, >, <>, >=, BETWEEN, LIKE, IN
Numbers	SalePrice > 6000
Test	
Simple Match one character Match many	LastName > 'Jones' License LIKE 'A_ _82_' LastName LIKE 'S%'
Dates	SaleDate BETWEEN '01-DEC-2008' AND '15-DEC-2008'
Missing Data	LastName Is NULL
Negation	FirstName IS NOT NULL NT (ModelType='Race'
Sets	ModelType IN ('Race', 'Road', 'Tour')

Figure 2.16

SQL criteria with OR connectors. Parentheses are used to isolate the OR condition for Mountain or Mountain full model types. Then the AND conditions are listed only once. Convert to design view to see how it is handled in the editor.

complex statements, it is generally best to rely on SQL to create the conditions. The conditions are easier to create and easier to read in SQL. However, in all cases, build the query in steps and test it at each step.

Actually, the mountain bike model types in RT were specifically named to support another way to include both the Mountain or Mountain full models without the hassle of using an OR clause. Notice that both begin with the word "Mountain." Consequently, whenever both types are desired, the condition can be written: ModelType LIKE 'Mountain%'.

Useful WHERE Clauses

Most database systems provide the comparison operators displayed in Figure 2.16. Standard numeric data can be compared with equality and inequality operators. Text comparisons are usually made with the LIKE operator for pattern matching. For all text criteria, you need to know if the system uses case-sensitive comparisons. By default, Microsoft SQL Server is not case-sensitive, so you can type the pattern or condition using any case. If case-sensitivity is desired, the default collation method can be changed to one that is case-sensitive. If you do not know which case was used, you can use the UPPER function to convert to upper case and then write the pattern using capital letters.

The **BETWEEN** clause is not required, but it saves some typing and makes some conditions a little clearer. The clause (SaleDate BETWEEN '01-DEC-2008' AND '15-DEC-2008' is equivalent to (SaleDate >= '01-DEC-2008' AND SaleDate <= '15-DEC-2008'). The date syntax shown here can be used on most database systems. Some systems allow you to use shorter formats, but on others, you will have to specify a conversion format. These conversion functions are not standard.

```
SaleItem(SaleID, ItemID, SalePrice, Quantity)
SELECT SaleID, ItemID, SalePrice, Quantity,
  SalePrice*Quantity As Extended
FROM SaleItem;
```

SaleID	ItemID	Price	Quantity	Extended
24	25	2.70	3	8.10
24	26	5.40	2	10.80
24	27	31.50	1	31.50

Figure 2.17

Computations. Basic computations (+ - * /) can be performed on numeric data in a query. The new display column should be given a meaningful name.

Another useful condition is to test for missing data with the NULL comparison. Two common forms are IS NULL and IS NOT NULL. Be careful—the statement (City = NULL) will not work with most systems, because NULL is not a value. You must use (City IS NULL) instead. Unfortunately, conditions with the equality sign are not flagged as errors. The query will run—it just will never match anything.

Computations

What types of computations can be performed in SQL? The statistical computations used for data mining are handled later by the business intelligence tools which use a programming language. However, some simple computations can be performed directly within SQL. Queries are used for two types of computations: aggregations and simple arithmetic on a row-by-row basis. Sometimes the two types of calculations are combined. Consider the row-by-row computations first.

Basic Arithmetic Operators

SQL and the designer can both be used to perform basic computations on each row of data. This technique can be used to automate basic tasks and to reduce the amount of data storage. Consider a common order or sales form. As Figure 2.17 shows, the basic tables would include a list of items purchased: SaleItem(SaleID, ItemID, SalePrice, Quantity). In most situations you would need to multiply SalePrice by Quantity to get the total value for each item ordered. Because this computation is well defined (without any unusual conditions), there is no point in storing the result—it can be recomputed whenever it is needed. Simply build a query and add one more column. The new column uses elementary algebra and lists a name: SalePrice*Quantity AS Extended. Remember that the computations are performed for each row in the query.

Most systems provide additional mathematical functions. For example, basic mathematical functions such as absolute value, logarithms, and trigonometric functions are usually available. Although these functions provide extended capabilities, always remember that they can operate only on data stored in one row of a table or query at a time.

Sum	Avg	Min	Max	Count
StDev	StDevP	Var	VarP	

Figure 2.18

SQL Server Aggregation functions. Sum, Avg, and Count are most common. The standard format is Sum(column) but it is also possible to use Count (DISTINCT column) to examine only the unique values.

Aggregation

Databases for business often require the computation of totals and subtotals. Note that the data mining approach in Chapter 3 is a better way to examine many subtotals, but sometimes SQL subtotals are useful to configure data for other analyses. Hence, query systems provide functions for **aggregation** of data. The common functions listed in Figure 2.18 can operate across several rows of data and return one value. The most commonly used functions are Sum, Avg, and Count which are similar to those available in spreadsheets. SQL Server also supports the DISTINCT clause to examine only the unique values; such as using Count(DISTINCT ModelType) to count each model type only one time. Be careful when using Sum and Count. Count simply counts the number of rows regardless of the data value. Sum adds the values selected. Sum is useful for monetary or quantity columns, Count is often used for key or ID columns.

With SQL, the functions are simply added as part of the SELECT statement. Figure 2.19 shows the SQL command to compute the total value of bicycles sold in December 2008. These basic queries are often easier to write in SQL than in the designer.

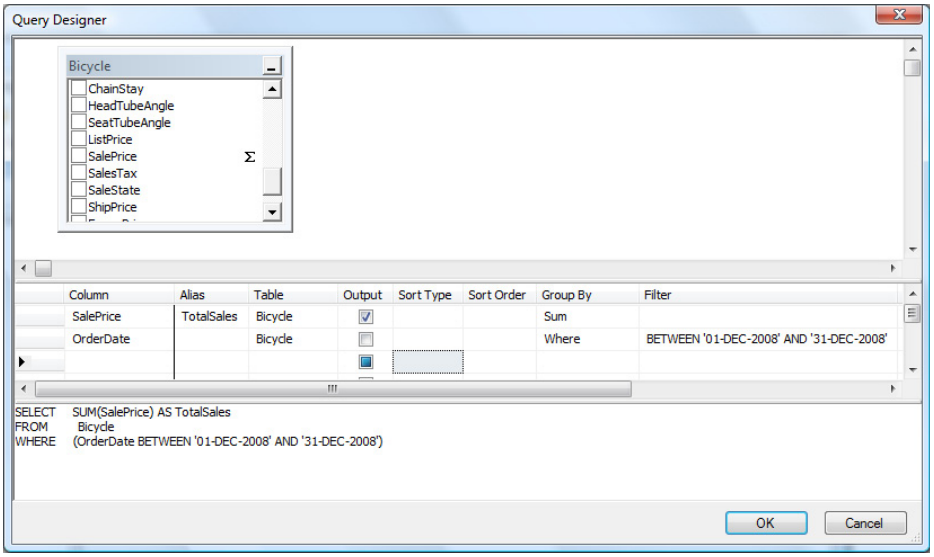
The designer can be used to build queries with totals. As shown in Figure 2.20, the first trick is to right-click the table area and choose the option to “Add Group By” to add the column with the Group By options. Use the drop-down list to select Sum for the SalePrice data. The second big trick is to select the WHERE option for the OrderDate. By default, the option is GROUP BY, which means that the totals will be computed for each item in the column (day). To compute just a single total across all days, the WHERE option switches the condition to the WHERE clause of the SQL statement. This difference is explained in the next section.

The **row-by-row calculations** can also be combined with an aggregate function. The example in Figure 2.21 asks for the total value of a particular purchase

Figure 2.19

Aggregation functions. SQL statement to obtain total sales of bicycles in December 2008.

```
SELECT SUM(SalePrice) As TotalSales
FROM Bicycle
WHERE OrderDate BETWEEN ‘01-DEC-2008’ AND ‘31-DEC-2008’;
```



Column	Alias	Table	Output	Sort Type	Sort Order	Group By	Filter
SalePrice	TotalSales	Bicycle	✓			Sum	
OrderDate		Bicycle				Where	BETWEEN '01-DEC-2008' AND '31-DEC-2008'

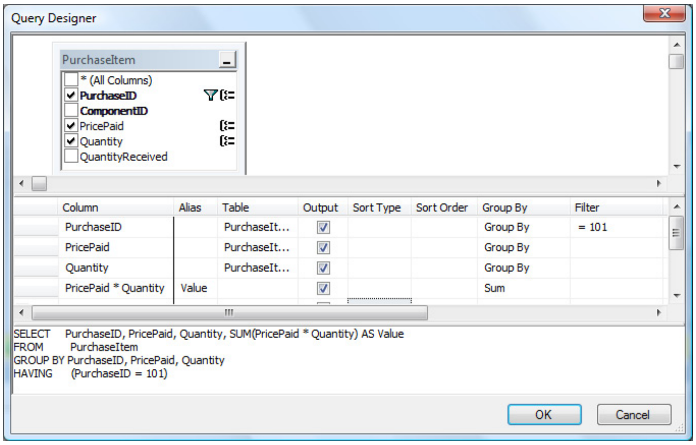
```
SELECT   SUM(SalePrice) AS TotalSales
FROM     Bicycle
WHERE    (OrderDate BETWEEN '01-DEC-2008' AND '31-DEC-2008')
```

Figure 2.20

Aggregation functions in the designer. Right-click the table area and choose "Add Group By" to see the Group By column. Select the Sum option for SalePrice and the WHERE option for the date.

Figure 2.21

Multiply values and then compute totals. What is the total value of items on purchase order 101?



Column	Alias	Table	Output	Sort Type	Sort Order	Group By	Filter
PurchaseID		PurchaseIt...	✓			Group By	= 101
PricePaid		PurchaseIt...	✓			Group By	
Quantity		PurchaseIt...	✓			Group By	
PricePaid * Quantity	Value		✓			Sum	

```
SELECT   PurchaseID, PricePaid, Quantity, SUM(PricePaid * Quantity) AS Value
FROM     PurchaseItem
GROUP BY PurchaseID, PricePaid, Quantity
HAVING   (PurchaseID = 101)
```

SELECT PurchaseID,
PricePaid, Quantity,
SUM(PricePaid * Quantity)
AS Value
FROM PurchaseItem
WHERE (PurchaseID = 101)
GROUP BY PurchaseID,
PricePaid, Quantity

Task	SQL Server
Strings	

Concatenation
Length
Upper case
Lower case
Partial string
	FName + ‘ ‘ + LName
	Length(LName)
	Upper(LName)
	Lower(LName)
	Substring(LName,2,3)

Dates	

Today
Month
Day
Year
Date arithmetic
	GetDate()
	DateName(month, myDate), Month(myDate)
	DatePart(day, myDate)
	DatePart(year, myDate), Year(myDate)
	DateAdd
	DateDiff

Formatting	Str(item, length, decimal) Cast, Convert
Numbers	
Math functions Exponentiation Aggregation Statistics	Cos, Sin, Tan, Sqrt Power(2, 3) Min, Max, Sum, Count, Avg, StDev, Var, LinRegSlope, Correlation

Figure 2.22

SQL functions. Hundreds of internal functions exist in SQL Server, but these are the most common.

order. To get total value, the database must first calculate Quantity * PricePaid for each row and then get the total of that column. This example computes the total for just one specific order (101).

There is one important restriction to remember with aggregation. The query can display either the details or the totals—not both at the same time. This constraint can sometimes be avoided by using complex queries, but it is best to decide in advance if the SQL should display details or totals, then use other tools to manipulate the results.

Note that several aggregate functions can be computed at the same time. For example, the Sum, Average, and Count can be displayed at the same time: SELECT Sum(Quantity), Avg(Quantity), Count(Quantity) From PurchaseItem. In fact, if you need all three values, you should compute them at one time. Consider what happens if you have a table with a million rows of data. If you write three separate queries, the DBMS has to make three passes through the data. By combining the computations in one query, you cut the total query time to one-third. With huge tables or complex systems, these minor changes in a query can make the difference between a successful application and one that takes days to run.

Functions

The SELECT command also supports functions that perform calculations on the data. These calculations include numeric forms such as the trigonometric func-

```
SELECT Count(SerialNumber) As Nbikes
FROM Bicycle
WHERE ModelType='Race'
          AND OrderDate BETWEEN '01-NOV-2008' AND '30-NOV-2008'
```

Figure 2.23

SQL subtotal introduction. How many Race bikes were sold in November 2008? Use SQL and not the designer.

tions, string function such as concatenating two strings, date arithmetic functions, and formatting functions to control the display of the data. Unfortunately, these functions are not standardized, so each DBMS vendor has different function names and different capabilities. Figure 2.22 lists some of the common functions used in SQL Server.

String operations are relatively useful. Concatenation is one of the more powerful functions, because it combines data from multiple columns into a single display field. It is particularly useful to combine a person's last and first names. Other common string functions convert the data to all lowercase or all uppercase characters. The length function counts the number of characters in the string column. A substring function is used to return a selected portion of a string. For example, you might choose to display only the first 20 characters of a long title.

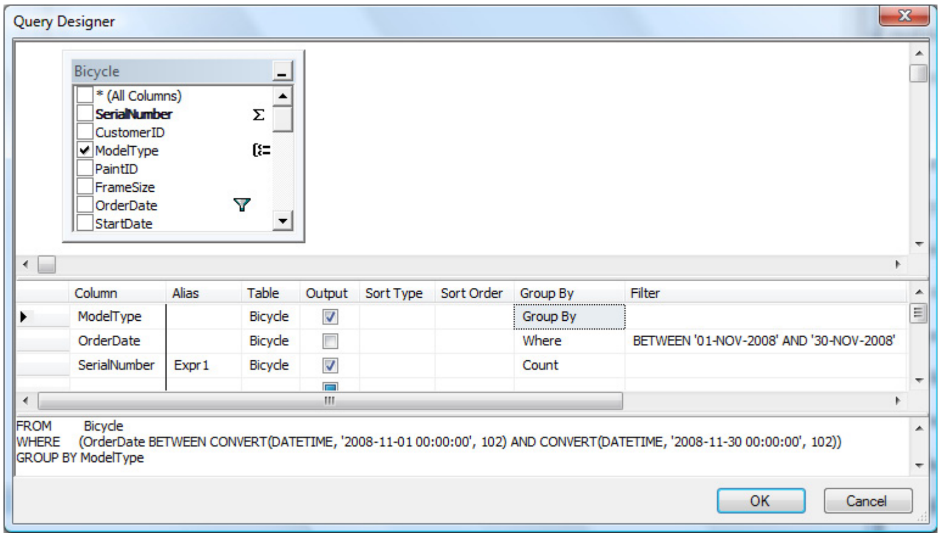
The powerful date functions are often used in business applications. Date columns can be subtracted to obtain the number of days between two dates. Additional functions exist to get the current date and time or to extract the month, day, or year parts of a date column. Date arithmetic functions can be used to add (or subtract) months, weeks, or years to a date. The Convert function can be used to specify the format of a date. It is also used for other types of data, such as setting a fixed number of decimal points or displaying a currency sign.

It is also possible to define custom functions. SQL Server uses T-SQL to enable you to create programming code in SQL that can define relatively complex operations on data and return a new function value. For even more complex calculations, it is possible to create new functions using a .NET procedural language such as C#. These tasks are covered in database textbooks and typically handled by database programmers instead of managers.

Subtotals and GROUP BY

How are subtotals computed? The previous section hinted at the use of totals but one of the most powerful features of SQL was left for this section: subtotals. Many business questions involve the use of subtotals: Who are the best customers? Which employee sold the most bicycles in November? Which model type was the most popular in 2008? All of these questions require totals or counts for each value: The total sales for each customer, total sales for each employee, and count of bicycles for each model type.

To illustrate, consider the question: How many Race bikes were sold in November 2008? As shown in Figure 2.23, the SQL is straightforward—simply use the Count function and a WHERE clause for ModelType and OrderDate. To test the function, stick with the SQL and avoid the designer.



Column	Alias	Table	Output	Sort Type	Sort Order	Group By	Filter
ModelType		Bicycle	✓			Group By	
OrderDate		Bicycle				Where	BETWEEN '01-NOV-2008' AND '30-NOV-2008'
SerialNumber	Expr1	Bicycle	✓			Count	

```
FROM      Bicycle
WHERE     (OrderDate BETWEEN CONVERT(DATETIME, '2008-11-01 00:00:00', 102) AND CONVERT(DATETIME, '2008-11-30 00:00:00', 102))
GROUP BY ModelType
```

Figure 2.24

Subtotal with designer. How many bikes of each model type were sold in November 2008?

Before building the query in the designer, it is useful to generalize the question to require the use of subtotals: What is the number of bicycles sold of each model type in November 2008? The only difference is that this query does not require the ModelType to be constrained. Instead, it requires the DBMS to find each Model-Type, restrict the date to the desired range, and count the number of each type of model. Although it sounds complex, SQL handles the setup easily.

Build a new query using the designer. Use the Bicycle table and add only the columns: ModelType, OrderDate, and SerialNumber. Figure 2.24 shows the setup. Add the Group By column and leave Group By as the mode for the ModelType because the business question calls for a value "for each" model type. Enter the date filter and set the Group mode to Where. If this mode is left on Group By, the

Figure 2.25

Subtotal in SQL with results.

```
SELECT    ModelType, COUNT(SerialNumber) AS NBikes
FROM      Bicycle
WHERE     (OrderDate
          BETWEEN CONVERT(DATETIME, '2008-11-01 00:00:00', 102)
          AND CONVERT(DATETIME, '2008-11-30 00:00:00', 102))
GROUP BY ModelType;

Mountain        13
Mountain full   57
Race            52
Road            47
Tour            13
```

```
SELECT    ModelType, COUNT(SerialNumber) AS NBikes
FROM      Bicycle
WHERE     (OrderDate
          BETWEEN CONVERT(DATETIME, ‘2008-11-01 00:00:00’, 102)
          AND CONVERT(DATETIME, ‘2008-11-30 00:00:00’, 102))
GROUP BY ModelType
HAVING Count(SerialNumber) > 15;
```

Figure 2.26

Limiting the output with a HAVING clause. The GROUP BY clause with the Count function provides a count of the number of animals in each category. The HAVING clause restricts the output to only those categories having a count greater than 15.

query will return values for each date—which is not called for in the business question. Finally, set Count for the SerialNumber. Almost any column could be counted since the DBMS simply counts the number of rows returned, but SerialNumber is best because it indicates that bicycles are being counted and because it is the primary key it avoids potential problems with duplicate or missing values.

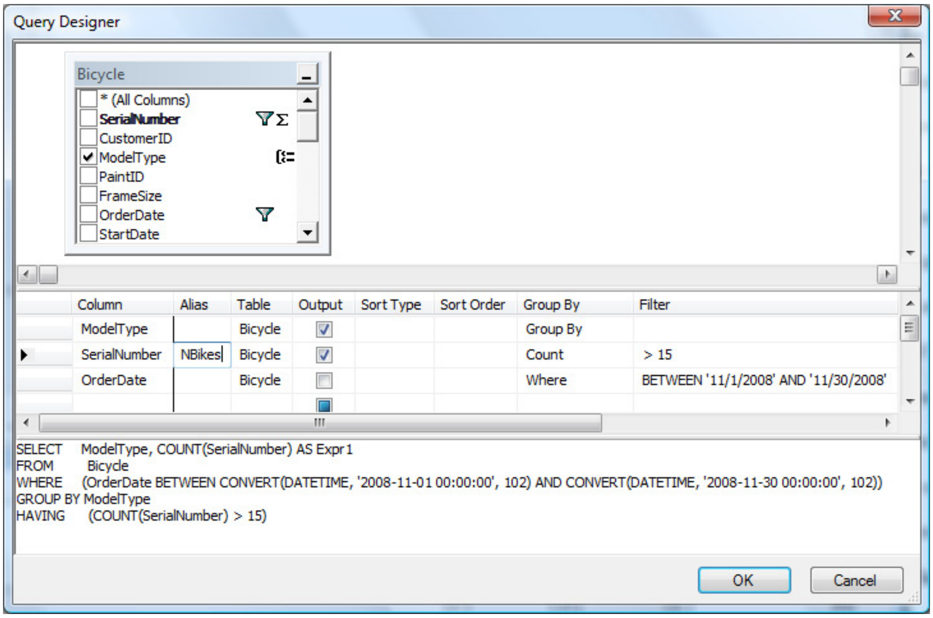
Figure 2.25 shows the SQL statement for the subtotal computation along with the results. Your results might differ slightly because the data in the database could have been altered. The alias (NBikes) has been added to provide a more descriptive title for the result column instead of Expr1. To obtain subtotals, the only new step is to add the GROUP BY clause. The **GROUP BY** statement can be used only with one of the aggregate functions (Sum, Avg, Count, and so on). With the GROUP BY statement, the DBMS looks at all the data, finds the unique items in the group, and then performs the aggregate function for each item in the group.

By default, the output will generally be sorted by the group items. However, for business questions, it is common to sort (ORDER BY) based on the computation. Be careful about adding multiple columns to the GROUP BY clause. The subtotals will be computed for each distinct item in the entire GROUP BY clause. Including additional columns (particularly date) might lead to a more detailed breakdown than desired.

Conditions on Totals (HAVING)

The GROUP BY clause is powerful and provides useful information for making decisions. In cases involving many groups, you might want to restrict the output list, particularly when some of the groups are relatively minor. For example, the simple Mountain and Tour types are not ordered much. In analyzing sales the managers might prefer to focus on the top-selling categories.

One way to reduce the amount of data displayed is to add the **HAVING** clause. The HAVING clause is a condition that applies to the GROUP BY output. In the example presented in Figure 2.26, the managers want to skip any model type category that has fewer than 15 animals. Notice that the SQL statement simply adds one line. The same condition can be added to the criteria grid in the designer query. The HAVING clause is powerful and works much like a WHERE statement. Just be sure that the conditions you impose apply to the computations indicated by the GROUP BY clause.



Column	Alias	Table	Output	Sort Type	Sort Order	Group By	Filter
ModelType		Bicycle	✓			Group By	
SerialNumber	NBikes	Bicycle	✓			Count	> 15
OrderDate		Bicycle	☐			Where	BETWEEN '11/1/2008' AND '11/30/2008'

```
SELECT    ModelType, COUNT(SerialNumber) AS Expr1
FROM      Bicycle
WHERE     (OrderDate BETWEEN CONVERT(DATETIME, '2008-11-01 00:00:00', 102) AND CONVERT(DATETIME, '2008-11-30 00:00:00', 102))
GROUP BY ModelType
HAVING    (COUNT(SerialNumber) > 15)
```

Figure 2.27

WHERE versus HAVING. Count the bicycles sold in November 2008 by model type and show only those where the result is greater than 15. It is hard to see the difference between these conditions in the designer. SQL is often easier to see that WHERE conditions are applied before the GROUP BY and HAVING is applied to the results.

WHERE versus HAVING

At first glance, WHERE and HAVING look very similar, and choosing the proper clause can be confusing. Yet it is crucial that you understand the difference. If you make a mistake, the DBMS will give you an answer, but it will not be the answer to the question you want. The key is that the WHERE statement applies to every single row in the original table. The HAVING statement applies only to the subtotal output from a GROUP BY query. To add to the confusion, you can even combine WHERE and HAVING clauses in a single query—because you might want to look at only some rows of data and then limit the display on the subtotals.

Figure 2.27 shows the query in the designer that includes both the WHERE and HAVING conditions. Looking at the layout, it is difficult to determine the difference between the two conditions. The WHERE entry in the GROUP BY column is required to shift a condition to the WHERE statement instead of the HAVING statement; and this change can be hard to remember. In most cases, you should check the SQL to ensure that the conditions are structured correctly. The WHERE clause is used to filter rows before any calculations take place. The HAVING condition simply limits the output of the GROUP BY results. In general, it is best to use WHERE clauses whenever possible because they immediately cut the number of rows to be investigated. The SQL Server query processor has a decent optimizer so making a mistake will not seriously hurt performance, but to understand the difference, it is best to think of the WHERE clause as the initial filter.

```
Try:
SELECT EmployeeID, Max(SalePrice)
FROM Bicycle
GROUP BY EmployeeID;
Which returns a list of employees and the most expensive bicycle sold by each employee.
It does not return the totals.

Try:
SELECT EmployeeID, Max(Sum(SalePrice))
 FROM Bicycle
GROUP BY EmployeeID;
Which does not even run.

Answer:
SELECT EmployeeID, Sum(SalePrice) AS Revenue
FROM Bicycle
GROUP BY EmployeeID
ORDER BY Sum(SalePrice) DESC
```

Figure 2.28

Find the best salesperson. Several tempting methods do not work. The solution is to compute the total sales for each employee using the GROUP BY statement and use ORDER BY to sort the results. To display less than the entire list, use a HAVING clause or enter TOP 5 just after the SELECT clause.

The Best and the Worst

Think about the business question, *Who is the best salesperson?* How would you build a SQL statement to answer that question? To begin, you have to decide if "best" is measured in quantity, revenue, or profit. For now, simply use revenue. A common temptation is to write a query similar to SELECT EmployeeID, Max(SalePrice) FROM Bicycle GROUP BY EmployeeID. This query will run. It will return a list of employees and the most expensive bicycle sold by each employee; but it will not sum the prices. A step closer might be SELECT EmployeeID, Max(Sum(SalePrice)) FROM Bicycle GROUP BY EmployeeID. But this query will not run because the database cannot compute the maximum until after it has computed the sum. So, the best answer is to use: SELECT EmployeeID, Sum(SalePrice) AS Revenue FROM Bicycle GROUP BY EmployeeID ORDER BY Sum(SalePrice) DESC. This query will compute the total sale prices for each employee and display the result in descending order—the best salespeople will be at the top of the list.

The advantage to this approach is that it shows other rows that might be close to the "best" entry, which is information that might be valuable to the decision maker. The one drawback to this approach is that it returns the complete list of items sold. Generally, most businesspeople will want to see more than just the top or bottom item, so it is not a serious drawback—unless the list is too long. In that case, you can use the HAVING command to reduce the length of the list. SQL Server also supports the TOP command to restrict a list without knowing anything about the data. Simply add TOP 5 to the SELECT statement: SELECT TOP 5 EmployeeID, Sum(SalePrice As Revenue…. Of course, the number 5 is arbitrary and any value can be used. In practice, the Min and Max functions are rarely used.

SELECT CustomerID
FROM Bicycle
WHERE OrderDate = ‘01-DEC-2008’;
	19114
	31202
	31335
	31651
	32631
	32687

Figure 2.29

List the CustomerID of everyone who bought a bicycle on December 1, 2008. Most people would prefer to see the names and phone numbers of the customers—those attributes are in the Customer table.

Multiple Tables

How do you use multiple tables in a query? All the examples so far have used a single table—to keep the discussion centered on the specific topics. In practice, however, you often need to combine data from several tables. In fact, the strength of a DBMS is its ability to combine data from multiple tables.

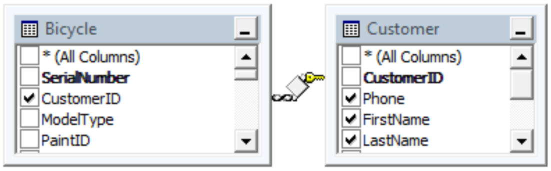
The essence of a relational database is to split data into separate pieces that are linked through the primary keys. This approach is efficient at storing data but it requires a query system to join the tables back together. From a query design perspective, SQL has a straightforward method of connecting tables. For example, the Bicycle table contains just the CustomerID to identify the specific customer. Most people would prefer to see the customer name and other attributes. This additional data is stored in the Customer table—along with the CustomerID. The objective is to take the CustomerID from the Sale table and look up the matching data in the Customer table.

Joining Tables

With modern query languages, combining data from multiple tables is straightforward. You simply specify which tables are involved and how the tables are connected. The query designer is particularly easy to use for this process. To understand the process, first consider the business question posed in Figure 2.29: list the CustomerID of everyone who bought something on 01-DEC-2008..

Most managers would prefer to see the customer name instead of CustomerID. However, the name is stored in the Customer table because it would be a waste of space to copy all of the attributes to every table that referred to the customer. If you had these tables only as printed reports, you would have to take the CustomerID from the sale reports and find the matching row in the Customer table to get the customer name. Of course, it would be time-consuming to do the matching by hand. The query system can do it easily.

As illustrated in Figure 2.30, the designer approach is somewhat easier than the SQL syntax. However, the concept is the same. First, identify the two tables involved (Bicycle and Customer). In the designer, select the tables from a list and they are displayed at the top of the form with the JOIN drawn as a line. In SQL, enter the table names on the FROM line. Second, specify which columns are matched in each table. In this case match CustomerID in the Bicycle table to the CustomerID in the Customer table. Most of the time the column names will be the same, but they could be different.



```
SELECT    Bicycle.CustomerID, Customer.FirstName,
          Customer.LastName, Customer.Phone
FROM      Bicycle
INNER JOIN Customer
          ON Bicycle.CustomerID = Customer.CustomerID
WHERE     (Bicycle.OrderDate = '01-DEC-2008')
```

CID	FirstName	LastName	Phone
19114	James	Dugan	(405) 015-0612
31202	Michael	Macdonald	(803) 408-1618
31335	Theresa	Raux	(540) 689-8730
31651	Kwok	Lawrence	(850) 136-9460
32631	Joseph	Despirito	(814) 732-0324
32687	B	Wagman	(863) 989-8229

Figure 2.30

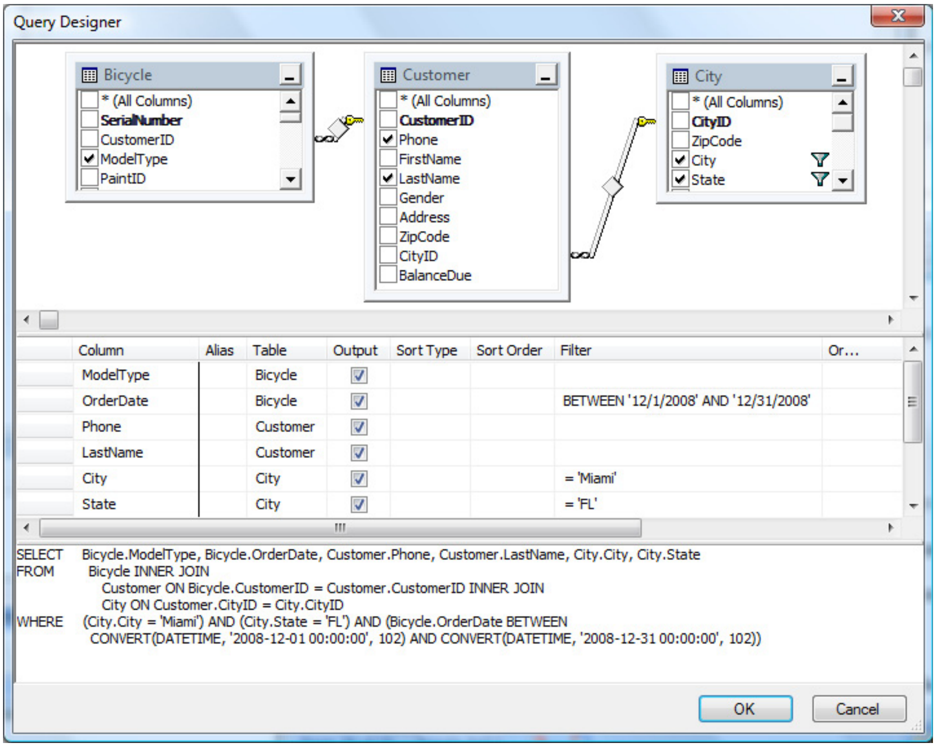
Joining tables causes the rows to be matched based on the columns in the JOIN statement. You can then use data from either table. The business question is, List the names and phone numbers of customers who bought a bicycle on December 1, 2008.

In SQL, tables are connected with the JOIN statement. The syntax for a JOIN is: FROM Bicycle INNER JOIN Customer ON Bicycle.CustomerID = Customer.CustomerID. The order of the tables does not matter. Notice that the concepts for SQL and the designer are the same: List both tables and which columns in the two tables are matched. SQL requires more typing. When multiple tables are involved, it is often easier to handle the joins in the designer. To add more tables in SQL, simply add another INNER JOIN statement with its associated ON clause to specify the columns.

Identifying Columns in Different Tables

Examine how the columns are specified in the SQL JOIN statement. Because the column CustomerID is used in both tables, it would not make sense to write CustomerID = CustomerID. The DBMS would not know what you meant. To keep track of which column you want, you must also specify the name of the table: Sale.CustomerID. Actually, you can use this syntax anytime you refer to a column. You are required to use the full table.column name only when the same column name is used in more than one table.

SQL Server supports two more levels: the database and schema. The schema is typically a name assigned to a role, and in many cases it is dbo (short for database owner). So, a fully-named table would be: database.schema.table. For example: SELECT * FROM RT.dbo.ModelType.



Column	Alias	Table	Output	Sort Type	Sort Order	Filter	Or...
ModelType		Bicycle	✓				
OrderDate		Bicycle	✓			BETWEEN '12/1/2008' AND '12/31/2008'	
Phone		Customer	✓				
LastName		Customer	✓				
City		City	✓			= 'Miami'	
State		City	✓			= 'FL'	

```
SELECT   Bicycle.ModelType, Bicycle.OrderDate, Customer.Phone, Customer.LastName, City.City, City.State
FROM      Bicycle INNER JOIN
            Customer ON Bicycle.CustomerID = Customer.CustomerID INNER JOIN
            City ON Customer.CityID = City.CityID
WHERE    (City.City = 'Miami') AND (City.State = 'FL') AND (Bicycle.OrderDate BETWEEN
          CONVERT(DATETIME, '2008-12-01 00:00:00', 102) AND CONVERT(DATETIME, '2008-12-31 00:00:00', 102))
```

Figure 2.31

Joining multiple tables. The designer makes it easy to connect multiple tables—simply add them to the list and verify the column connections.

Joining Many Tables

A query can use data from several different tables. The process is similar regardless of the number of tables. Each table you want to add must be joined to one other table through a data column. If you cannot find a common column, either the table design is wrong or you need to find a third table that contains links to both tables.

Consider the example in Figure 2.31: List customers from Miami, FL who purchased bicycles in December 2008. An important step is to identify the tables needed. For large problems involving several tables, it is best to first list the columns you want to see as output and the ones involved in the constraints. In the example, the name and phone number you want to see are in the Customer table. The city name and state are in the City table, and the OrderDate is in the Bicycle table. Fortunately, all three of these tables are connected to each other and you do not need to search for an intermediate connection table.

When the database contains a large number of tables, complex queries can be challenging to build. You need to be familiar with the tables to determine which tables contain the columns you want to see. For large databases, an entity-relationship diagram (ERD) or a class diagram can show how the tables are connected.

When you first see it, the SQL 92 syntax for joining more than two tables can look confusing. In practice, it is best not to memorize the syntax. When you are

```
SELECT          Bicycle.ModelType, Bicycle.OrderDate, Customer.Phone,
        Customer.LastName, City.City, City.State
FROM    Bicycle
INNER JOIN      Customer
                ON Bicycle.CustomerID = Customer.CustomerID
INNER JOIN      City
                ON Customer.CityID = City.CityID
WHERE(City.City = 'Miami')
        AND (City.State = 'FL')
        AND (Bicycle.OrderDate BETWEEN
                CONVERT(DATETIME, '2008-12-01 00:00:00', 102)
                AND CONVERT(DATETIME, '2008-12-31 00:00:00', 102))
```

Figure 2.32

Joining multiple tables. The designer makes it easy to connect multiple tables—simply add them to the list and verify the column connections.

first learning SQL, understanding the concept of the JOIN is far more important than worrying about syntax. Figure 2.32 shows the syntax needed to join three tables. To handle multiple joins use the FROM statement to list the first table and add an INNER JOIN new_table ON table.column = new_table.column statement for each table that needs to be added. SQL Server does not require parentheses to specify the order of the joins, but be careful when adding new tables. The column to be connected must already exist in the joined tables. In the Bicycle-Customer-City example, the joins have to be built either as Bicycle-Customer-City or City-Customer-Bicycle. A set of joins from City-Bicycle-Customer will not work because the Bicycle table does not include the CityID.

When in doubt, use the designer. Occasionally, it is necessary to change the joins created automatically in the designer. For example, the designer might include an extra join, such as connecting the Employee table to the City table. Deleting joins is straightforward—select the join and press the Delete key or right-click the join and choose the Remove option. Creating a new join is only a little harder. Select the column in one table (such as CustomerID), drag the column and drop it on top of the matching column in the second table. The direction of the drag is unimportant, but it often helps to expand or adjust the tables in advance so both tables clearly display the columns to be connected.

Views: Saved Queries

Complex queries are often easier to solve by breaking them into smaller pieces. An initial query can be saved as a **view**, which is simply a saved query. New queries can use views as if they were another table, and the joins are the same. Views are also useful for controlling access to data. Instead of giving a user access to an entire table, a view can be created that retrieves a restricted set of rows and columns. Other workers can use this view without knowing anything about the underlying table. This approach is useful for combining multiple tables into a single view so that the joins are hidden within the view. To create a view that retrieves data, first build the query and test it. Then add one line to the top of the query: CREATE VIEW view_name AS. Enter a unique and memorable name as

```
CREATE VIEW December2008Sales AS
SELECT   SerialNumber, CustomerID, ModelType, PaintID, FrameSize,
         OrderDate, StartDate, ShipDate, LetterStyleID, StoreID,
         EmployeeID, SalePrice, ListPrice, SalesTax, SaleState, ShipPrice
FROM     Bicycle
WHERE    (OrderDate BETWEEN
              CONVERT(DATETIME, '2008-12-01 00:00:00', 102)
              AND CONVERT(DATETIME, '2008-12-31 00:00:00', 102))
```

Figure 2.33

Creating a View. Build and test the query—bicycle sales in December 2008. Then add the first line to create the view with a unique name.

the view_name and run this new command. Instead of displaying the data, the system will create the query with the given name. Figure 2.33 shows how to create a view that lists bicycle sales only for December 2008. This view is needed for the next section.

Saved queries are useful when a problem has multiple parts. Consider the question: Which customers purchased bicycles in both 2007 and 2008? At first glance the question seems easy. You might try a single query:

```
SELECT *
FROM Customer
INNER JOIN Bicycle
      ON Customer.CustomerID=Bicycle.CustomerID
WHERE Year(OrderDate)=2007 AND Year(OrderDate)=2008;
```

This query will run but it will never return any matches because a date can never be both 2007 and 2008 at the same time. Instead, the question has to be split into two queries. SQL supports subqueries that enable the second part to be embedded into a single query, but subqueries are beyond the scope of this book. It is easier to simply create two views—one for each year. Save them as Customers2007 and Customers2008 and set the single appropriate year condition in each query. The list of customers can then be obtained by creating a new query that joins the two views. The JOIN condition establishes the "AND" condition that a customer fall into both lists:

```
SELECT *
FROM Customers2007
INNER JOIN Customers2008
  ON Customers2007.CustomerID=Customers2008.
CustomerID;
```

The key is to recognize when two separate queries are needed. No secret formula exists—you simply have to logically evaluate the business question and decide which conditions can be handled at one time and which ones require separate lists. Just remember that when you save a query as a view to give it a name that explains the data in the view so that it can be recognized later.

```
SELECT ModelType.ModelType
FROM ModelType
LEFT JOIN December2008Sales
  ON ModelType.ModelType=December2008Sales.ModelType
WHERE December2008Sales.SerialNumber Is Null;
```

Figure 2.34

Left Join example. Which model types were not sold in December 2008? Left Join is needed because Inner Join would display only model types that were sold.

LEFT JOIN

You might be wondering why the syntax for the join is INNER JOIN instead of just the word "Join." The difference is important because another form of the Join command exists. An inner join connects two tables by matching only the items in the first table that are equal to the items in the second table. Items in either table that do not match are ignored or dropped from the display. For most queries, this action is appropriate as a simple matching system. However, a special type of business question arises that is difficult to answer with the inner join approach. (It can be handled with subqueries, but these are not covered in this book.) The basic question is: Because the database records things that did happen, how can you find things that did not happen? This question seems strange at first, but consider the simple business question for Rolling Thunder Bicycles: Which model types were **not** sold in December 2008?

The December2008Sales view created in the previous section lists all bicycles ordered in December 2008, which are things that did happen. Where does the database store things that did not happen? Nowhere. The solution is to take the list of all model types (ModelType table), subtract out the model types that were sold, and the model types that remain are the ones that were not sold. Figure 2.34 shows the query needed to find the result. The ModelType table is connected to the December2008Sales view using a Left Join. The Left Join specifies that all rows from the left table are to be included in the results—even if no matching data exists in the right table. When an entry in the left, ModelType, table has no matching value in the sales table, the join enters a Null value for all columns in the sales table. Hence, the Is Null condition returns only those rows in the left table that have no matching values in the right (sales) table. In this case, the result is: Hybrid and Track model types. Figure 2.35 shows some random sample data from the query using the Left Join. The Null values for the Hybrid and Track model types are highlighted in red. All other model types have matching data, so only the two types meet the Is Null condition to indicate that they were not sold.

UNION

Joins are used to select columns from multiple tables or views. On the other hand,, sometimes it is useful to combine rows of data from multiple tables or views. This trick is useful when similar data is stored in separate tables. For example, consider a company that has two divisions (East and West) and keeps a separate employee table for each division: EmployeesEast and EmployeesWest. Most queries are

ModelType	Serial	CID	ModelType
Mountain full	37774	10433	Mountain full
Mountain full	37486	31386	Mountain full
Mountain	37232	31132	Mountain
Race	38080	31980	Race
Race	38123	32023	Race
Road	38359	32259	Road
Hybrid	Null	Null	Null
Tour	38787	32687	Tour
Track	Null	Null	Null

Figure 2.35

Left Join example. Sample random rows from the left join. The Null values for the Model Types that do not exist in the Sales table are highlighted in red.

based on a single table, but sometimes the company wants to retrieve data from both tables. The UNION command is the solution:

```
SELECT EID, LastName, FirstName, Gender, Phone, 'East'
As Division
FROM EmployeesEast
UNION
SELECT EID, LastName, FirstName, Gender, Phone, 'West'
As Division
FROM EmployeesWest
```

Figure 2.36 shows the result of the query. Notice that the columns must match exactly. Also, note the use of the created column Division to track the division for each employee after the data has been merged. In most cases, Union queries are saved as views and used as the basis for other searches. Union queries are useful for merging data from multiple sources—hence they appear in data warehouse situations when data comes from many places.

Data Manipulation

All of the queries covered so far have been SELECT statements—queries to retrieve data. From a data mining perspective, these are the most useful queries because they are used to set up data for analyses. They are also used to find answers to ad hoc questions. However, it is worth knowing that SQL is capable of other types of commands. Specifically, SQL has **data manipulation** and **data definition** commands. The data definitions commands are used to create tables, indexes, and other structures. For example, they are used to load the databases for this book, but the details are not covered in this book. On the other hand, the data manipulation commands are often used when configuring and loading data for a data warehouse. Even if you do not need to create the commands as a manager, it is worthwhile to get a glimpse of the power of the commands so you know what tasks can be handled by SQL. The three main commands are: UPDATE to change

EID	LastName	FirstName	Gender	Phone	Division
113	Jones	Jack	Male	2222	East
114	Smith	Sarah	Female	4444	East
225	Hart	Hank	Male	6624	West
256	Eccles	Ephraim	Male	4432	West

Figure 2.36

UNOIN example. Combining rows from two employee tables (East and West). Note that the columns in the two SELECT statements must match exactly.

data values, INSERT to copy data, and DELETE to delete rows of data. All three rely heavily on the WHERE concepts covered in the SQL command.

UPDATE

The SQL UPDATE command alters data in existing rows. Typically, it is applied to a single table at a time. The command operates on a single row at a time, but the WHERE clause can be used to apply the operation to multiple rows of data. Figure 2.37 shows a example of the UPDATE command that increases the list price of components by ten percent. The WHERE clause restricts the updates to components introduced in 2006 or later.

The UPDATE command can change multiple columns at the same time—all on the same row of data. The basic syntax is to separate the columns with commas: SET Col1=x, Col2=y, Col3=6.

To be safe, all updates should be tested first as SELECT statements. This approach is useful for ensuring the WHERE clause and computations are correct. For example, the example would be tested as:

```
SELECT 1.10*ListPrice AS NewPrice
FROM Component
WHERE Year >= 2006
```

INSERT

The INSERT command has two forms—one to insert single rows of data at a time and the other to copy rows of data from one table and insert them into a second table. The first method is useful for transactions, but the second approach is more useful for data warehouses where insert commands are useful for transferring data. Figure 2.38 shows an example of using INSERT to copy data. It assumes that a new table exists to hold bicycle sales data.

Always test the SELECT statement first! Then add the INSERT line at the top to send the results into the new table. For loading data warehouses, the SELECT statement is often used to make minor changes to the data as it is being transferred. For example, using an exchange rate table, monetary data could be converted to a standard currency.

```
UPDATE Component
SET ListPrice = 1.10*ListPrice
WHERE Year>=2006
```

Figure 2.37

UPDATE command. Increase the List Price of components that were introduced in 2006 or later.

DELETE

The SQL DELETE command is powerful—probably too powerful to trust. Avoid using it—besides how often do you really want to delete data? The basic format is:

```
DELETE
FROM Manufacturer
WHERE ManufacturerID =1000;
```

Do not run the command—the ID was specifically chosen because it does not match any manufacturers in the Rolling Thunder database, so it will not actually delete any rows. Any DELETE command should first be tested by writing it as a SELECT * statement. When you are completely satisfied that the WHERE clause is correct and that the data absolutely must be deleted, the SELECT statement can be replaced with the DELETE line.

However, DELETE commands are dangerous. Tables are interrelated, and deleting a row from one table causes cascade deletes in the other tables. For instance, deleting a manufacturer deletes all purchase orders and components associated with that vendor. Deleting components removes them from the BikeParts table, so it could remove information on which parts were installed on a bicycle. Deleting customers or employees is even more dangerous because those objects affect many tables.

If data is accidentally deleted, it might be possible to recover by entering a ROLLBACK command, if it is entered early enough. SQL Server keeps a transaction log of changes and has the ability to reset the database to an earlier point in the log—but only if the changes have not been fully committed. In other cases, it might be necessary to find and restore a backup copy of the database. To be safe, avoid deleting data. Still, it is useful in data warehouses where some tables may be emptied before loading new data.

Figure 2.38

INSERT command to transfer data. Test the SELECT query first and then add the INSERT command to send the results to a new table.

```
INSERT INTO newTable (SerialNumber, CustomerID, OrderDate, ModelType)
SELECT SerialNumber, CustomerID, OrderDate, ModelType
FROM Bicycle
WHERE OrderDate BETWEEN ’01-JAN-2008’ AND ’31-DEC-2008’;
```

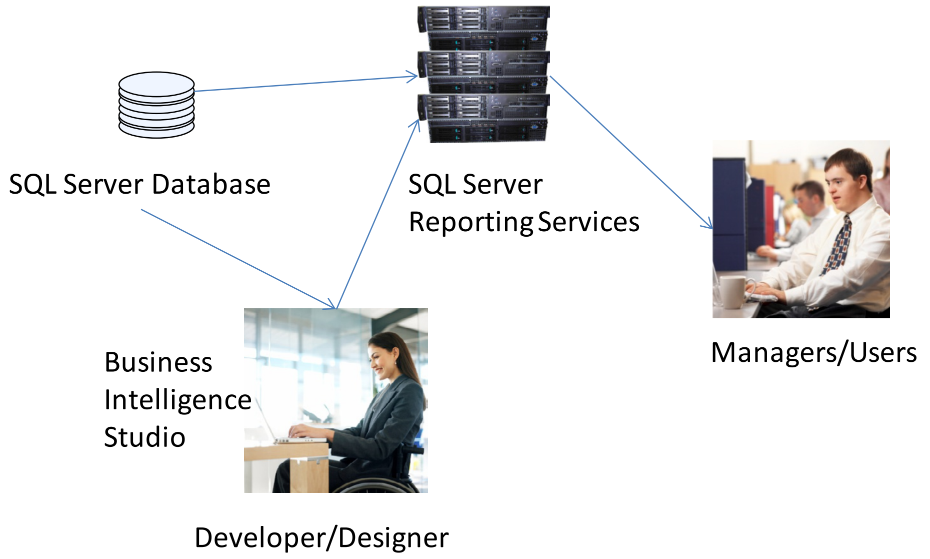


Figure 2.39

SQL Server Reports. Reports are designed and created with the Business Intelligence Studio. They are compiled and deployed to the Reporting Services server which connects to the database server. Users can work with the reports using a Web browser.

SQL Server Reports

How are reports created in SQL Server? Queries are useful but they do not have much in the way of formatting options. Reports combine queries with layout and format options to produce more useful information. In the "old days" reports were designed to be printed on paper. Now, SQL Server Reporting Services can run as a Web server and generate fixed reports as well as interactive reports that provide some initial exploration of the data to present both details and subtotals.

SQL Server reports require a computer running the Reporting Services and the **SQL Server Business Intelligence (BI)** (or Visual Studio) client tools. As shown in Figure 2.39, reports are created on a developer computer using the BI studio. The finished reports are compiled and deployed to a server running the SQL Server Reporting Services. This tool creates a Web interface, so managers can browse to the server and retrieve the reports using a standard Web browser. For development purposes, all of the components (SQL Server database, SQL Server Reporting Services, and the Business Intelligence Studio) can be installed on a single computer. However, if you are using a laptop, the Reporting Services are generally turned off by default to reduce the processing demands during typical use. To test reports, you will need to use the Computer Manager and Windows Services to start the Reporting Services running.

Administration Configuration

Reporting Services has a couple of administrative twists. Particularly when Reporting Services is installed on a Windows 7 operating system, you will probably

- Start browser/Internet Explorer with: Run as Administrator
- Open report folder: http://<server>/Reports http://localhost/Reports
- Use Tools/Options to add site as trusted site: http://localhost
- At reports site: Click the Properties tab
- Click button: New Role Assignment
- Enter your Windows user account: <domain>\<user>
- Click the option for: Content Manager
- Click the OK button

- Click the Site Settings link (top right)
- Select Security tab/option (left side)
- Click: New Role Assignment
- Enter your Windows user account: <domain>\<user>
- Check option: System Administrator
- Click OK

Figure 2.40

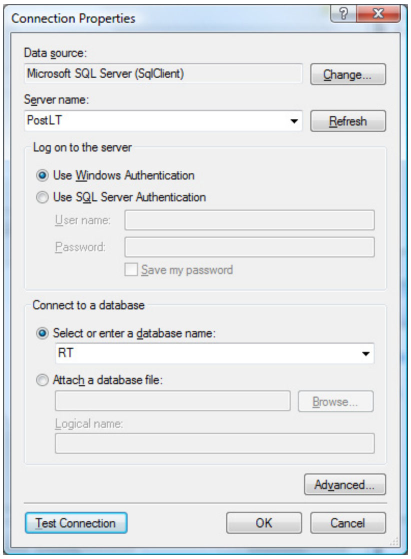
SQL Server Reporting Services configuration. With Vista and Windows 7 it is usually necessary to add the developer account as an administrator to Reporting Services.

need to add the developer account (yours) as an administrator to the Reporting Services. The first twist is that this process is handled through a Web browser, not through the standard Database console. Figure 2.40 shows the basic steps. Reporting Services typically has given high-level permissions only to the Windows Administrator account, so the browser must be started with “Run as Administrator.” Use the localhost name if the Reporting Services are running on the development computer—otherwise, replace localhost with the name of the server running the services. Setting these security permissions is required to enable you to create and save reports on the server.

Creating a Report

The Business Intelligence Studio has a report wizard that makes it relatively painless to create a basic report. The wizard is useful for establishing the database connections and defining the structure of the report. Once the report has been created, you can edit the design, improve the design and layout, and add more features if needed. Begin by starting the BI Studio and creating a new Report project. Choose a location to store the project files on your computer and enter a name that describes the project.

Visual Studio has several windows that display menus, code, properties, and report designs. All of these windows can be moved, resized, or closed. If a window is closed, search the menu options to open it when needed. To create a new report, in the Solution Explorer window, right-click the Reports entry and choose the option to add a new report. Follow the basic Wizard steps. As shown in Figure 2.41, the first step in creating a report is to establish a connection to the database. A connection requires the name of the server running the SQL Server DBMS, which could be a separate server. For most teaching purposes, the database is installed on the developer’s computer, so localhost usually works well as the server name. The login credentials must be entered for connecting to the database. Using the Windows account works for databases on the local computer. Choose the name of



Connection Properties

Data source: Microsoft SQL Server (SqlClient)

Server name: PostLT

Log on to the server

[✓ Use Windows Authentication]

Use SQL Server Authentication

Connect to a database

[✓ Select or enter a database name:] RT

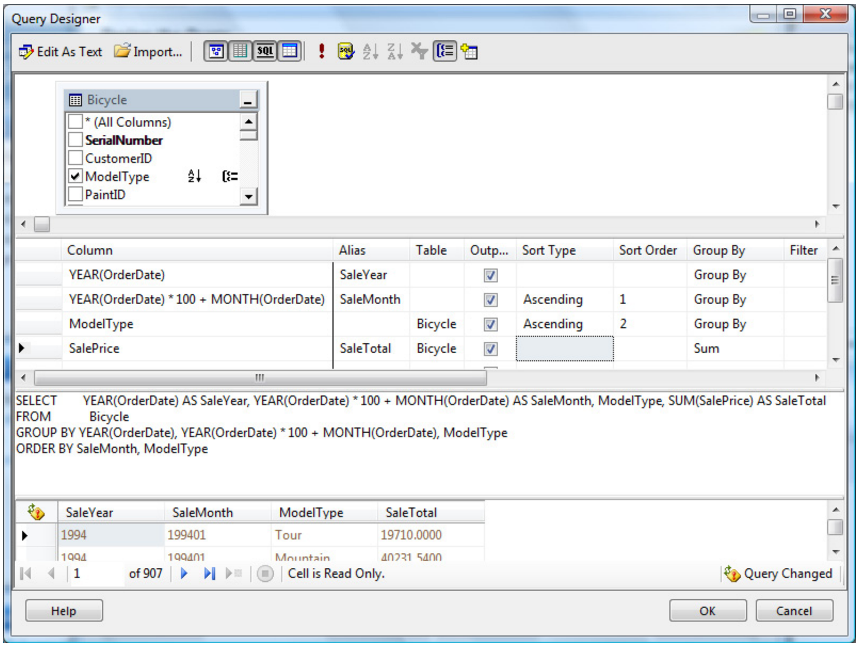
Attach a database file:

Figure 2.41

The first step to create a report. Establish a connection to the database. The server name, login credentials, and database name are needed.

Figure 2.42

Query to retrieve the detail for the report. The two date calculations return the Year and the YearMonth (such as 200901). The lowest detail shown on the report will be sales by model type by month.



Query Designer

Bicycle: * (All Columns), **SerialNumber**, CustomerID, [✓ ModelType], PaintID

Column	Alias	Table	Outp...	Sort Type	Sort Order	Group By	Filter
YEAR(OrderDate)	SaleYear		✓			Group By	
YEAR(OrderDate) * 100 + MONTH(OrderDate)	SaleMonth		✓	Ascending	1	Group By	
ModelType		Bicycle	✓	Ascending	2	Group By	
SalePrice	SaleTotal	Bicycle	✓			Sum	

```
SELECT      YEAR(OrderDate) AS SaleYear, YEAR(OrderDate) * 100 + MONTH(OrderDate) AS SaleMonth, ModelType, SUM(SalePrice) AS SaleTotal
FROM        Bicycle
GROUP BY YEAR(OrderDate), YEAR(OrderDate) * 100 + MONTH(OrderDate), ModelType
ORDER BY SaleMonth, ModelType
```

SaleYear	SaleMonth	ModelType	SaleTotal
1994	199401	Tour	19710.0000
1994	199401	Mountain	40231.5400

1 of 907 Cell is Read Only. Query Changed

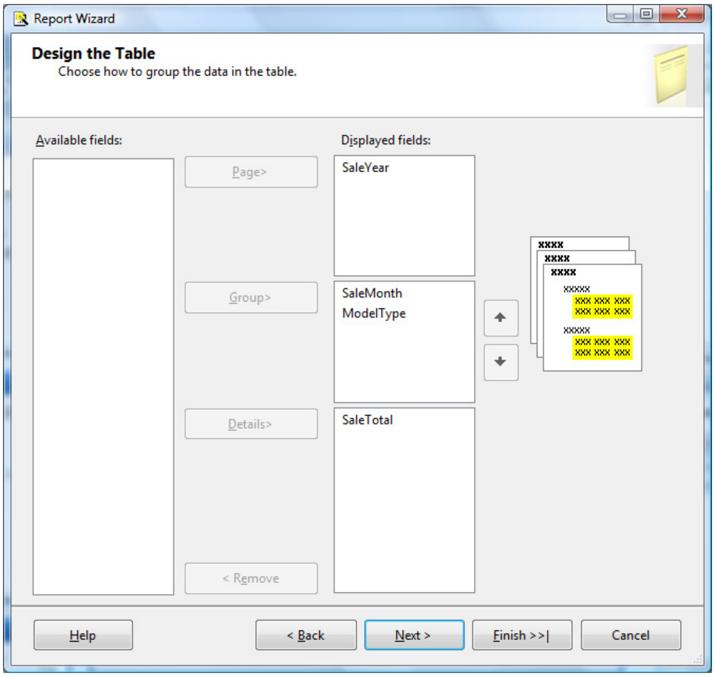


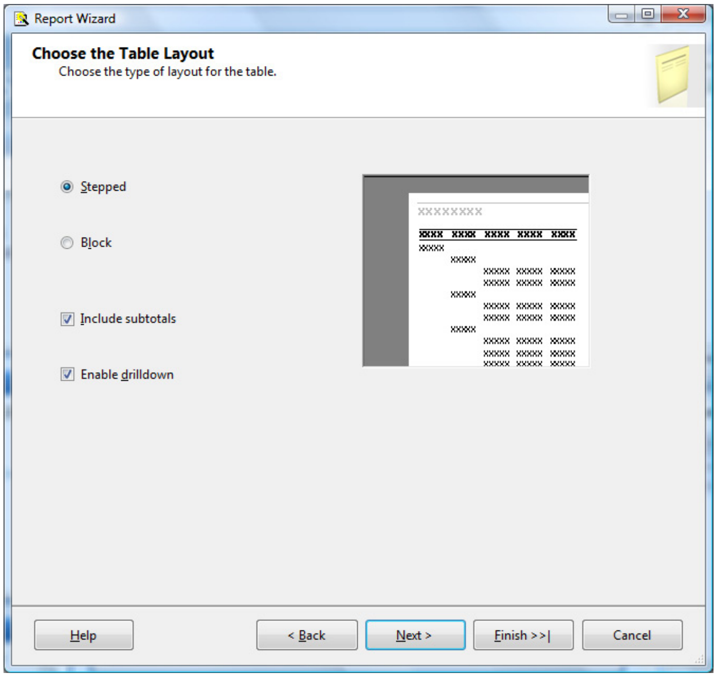
Figure 2.43

Report structure or levels. Before this step, choose the Tabular report layout instead of Matrix. Move columns from the Available fields panel into the appropriate level. SaleYear is the page break, SaleMonth and ModelType are the main Group breaks, and SaleTotal is the detail level.

the database and always click the button to test the connection. The data source can be saved as a shared data source (check box) so that it is available for multiple reports.

The most important aspect of a report is to build a query that retrieves the data needed for the report. The key to configuring the query is to retrieve the detail data needed for the report. Reports can have many levels with headings and subtotals; but ultimately, each report has a certain level of detail. The query needs to return the detail-level of data. The report itself will handle subtotals and formats. As an example, the goal is to create a report that displays total monthly sales of bicycles by model type. The total refers to value (of SalePrice) as opposed to the count of the number of bicycles. The goal is to create a report that lists one year on a page then the totals for each month with the detail level of sales by model type for that month. Consequently, the detail level needed in the query is the sum of SalePrice by month. As shown in Figure 2.42, the query also needs to return the Year value of the OrderDate to use as a page variable. The SQL statement is:

```
SELECT      YEAR(OrderDate) AS SaleYear,
      YEAR(OrderDate) * 100 + MONTH(OrderDate) AS
SaleMonth, ModelType,
      SUM(SalePrice) AS SaleTotal
FROM  Bicycle
GROUP BY YEAR(OrderDate), YEAR(OrderDate) * 100 +
MONTH(OrderDate), ModelType
ORDER BY SaleMonth, ModelType
```



(●) Stepped

( ) Block

[✓] Include subtotals

[✓] Enable drilldown

Figure 2.44

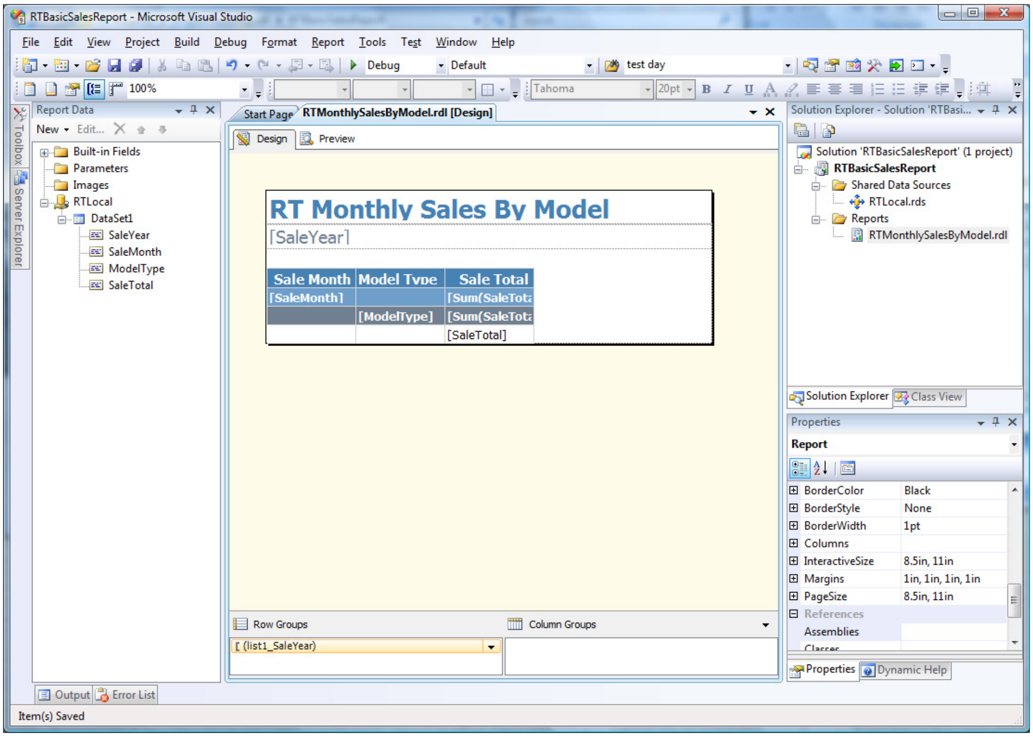
Report layout. Either Stepped or Block will work, but be sure to check both boxes to include subtotals and enable drilldown.

With the data selected, the next step is to define the report structure. Reports can have multiple levels or breaks. Each level can have a header and footer section. Headers are typically used to display titles or labels, and footers are used for subtotals. In the sales by month by model type example, the report will have groups for SaleMonth and ModelType. The page level break will be based on the Year.

Figure 2.43 shows the structure for the sample report using the Tabular layout. Place the SaleYear in the page level break. Place SaleMonth and Model Type—in that order—in the Group section. Move SaleTotal into the detail section.

For more complex cases, it is often best to place only the key or break items into the respective Group boxes. Avoid including additional data. For instance, if a grouping by Customer is desired, place CustomerID into the Group box, but do not include the customer name, address, phone number and so on. The report builder does not associate the additional columns with the key value—it will try to create new groupings based on each column in the group list. Leave related columns in the “available fields” box for now. They can be added to the report by hand after the main structure is defined.

Figure 2.44 shows the next step in the wizard—selecting the report layout. The stepped and block layouts are simply graphics design choices. Sometimes a report has to be tested both ways to see which version displays the data the best—particularly when a large number of columns are involved. The two checkboxes are more important—be sure to select both options to include subtotals and enable drilldown.



RT Monthly Sales By Model

[SaleYear]

Sale Month	Model Type	Sale Total
[SaleMonth]		[Sum(SaleTot
	[ModelType]	[Sum(SaleTot
		[SaleTotal]

BorderColor	Black
BorderStyle	None
BorderWidth	1pt
Columns	
InteractiveSize	8.5in, 11in
Margins	1in, 1in, 1in, 1in
PageSize	8.5in, 11in
References	
Assemblies	
Classes	

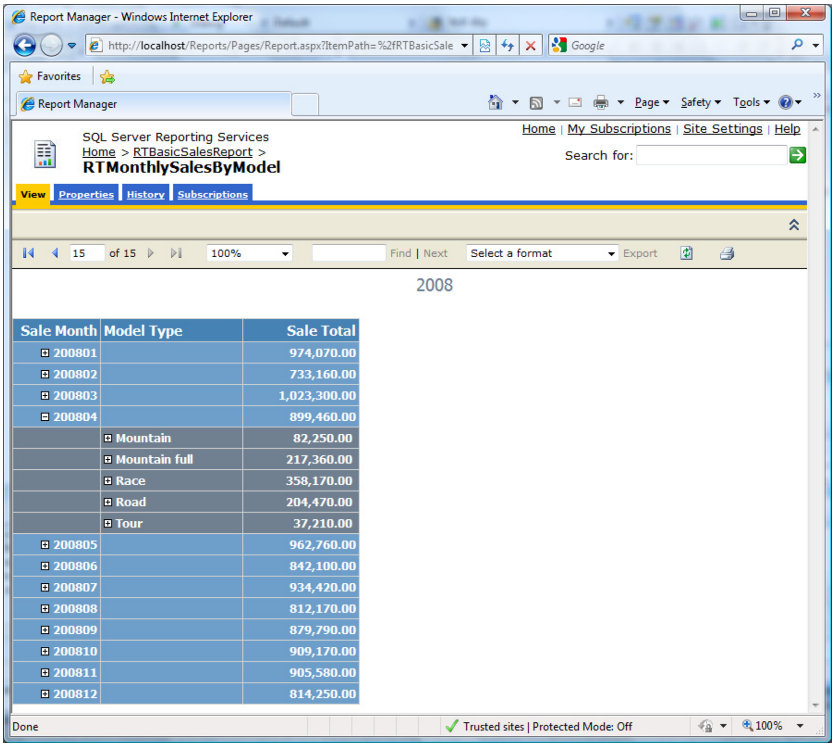
Figure 2.45

Initial Report design.

After this choice, the wizard provides options to set the overall color and style of the report. In a large project, all reports should follow a similar style. Some companies establish design guidelines that are used for all reports. It is also possible to add custom designs to the list, but those details are not covered in this book. Simply choose one of the existing designs. Follow the steps to finish the wizard and be sure to give the report a meaningful name that describes the report to users. However, the name should not include spaces. If necessary, use the underscore character (_) as a separator.

Figure 2.45 shows the initial report design and the Visual Studio editor. Because the model type names and sales totals might be wider than the allotted columns, it is useful to expand the column widths by dragging the dividing lines. Also, spaces can be added to the title. The report name itself should not contain spaces, but the display on the report can contain any desired characters. It is also possible to add logos and graphics.

The report can be previewed by clicking the Run button on the main toolbar (small green arrowhead). A useful improvement is to format the display of the totals. In the design mode, select the totals individually. Scroll the properties window near the bottom to find the Format line in the Number section. Enter #,##0.00 as the format to specify that the values should display with two decimal places and a separator for thousands. The report can be previewed with the Run button. When it is acceptable, right-click the main project name and choose the option to Deploy the project and reports to the reporting server.


Report Manager - Windows Internet Explorer
http://localhost/Reports/Pages/Report.aspx?ItemPath=%2fRTBasicSale
Favorites
Report Manager
Page
Safety
Tools
Home | My Subscriptions | Site Settings | Help
Search for:
View
Properties
History
Subscriptions
15 of 15
100%
Find | Next
Select a format
Export
Done
Trusted sites | Protected Mode: Off
100%

SQL Server Reporting Services
Home > RTBasicSalesReport >
RTMonthlySalesByModel

2008

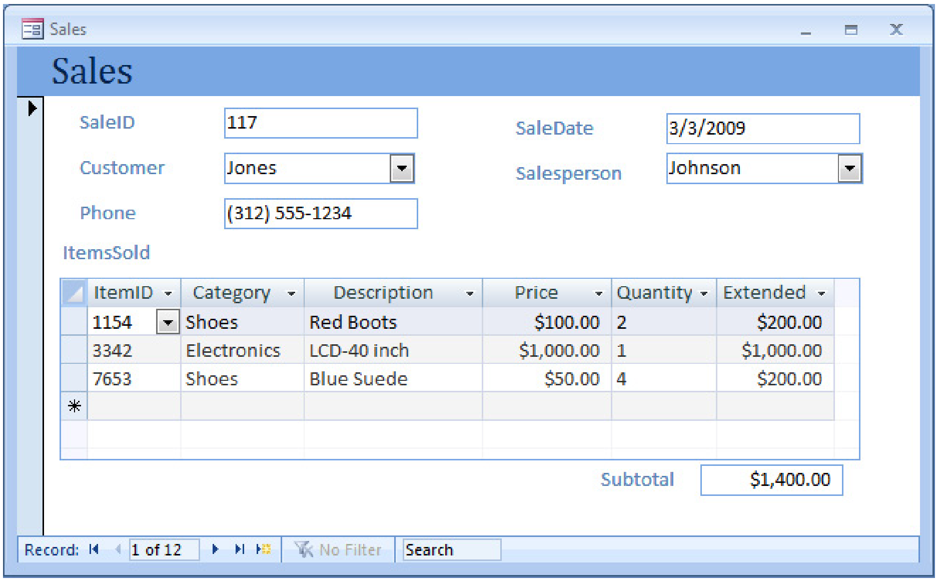
Sale Month	Model Type	Sale Total
⊞ 200801		974,070.00
⊞ 200802		733,160.00
⊞ 200803		1,023,300.00
⊟ 200804		899,460.00
	⊞ Mountain	82,250.00
	⊞ Mountain full	217,360.00
	⊞ Race	358,170.00
	⊞ Road	204,470.00
	⊞ Tour	37,210.00
⊞ 200805		962,760.00
⊞ 200806		842,100.00
⊞ 200807		934,420.00
⊞ 200808		812,170.00
⊞ 200809		879,790.00
⊞ 200810		909,170.00
⊞ 200811		905,580.00
⊞ 200812		814,250.00

Figure 2.46

Report displayed in browser. On a local machine, use http://localhost/reports to connect to the report server. Then navigate to the specific report.

Figure 2.46 shows the report generated from within a browser. The default location for the reporting services on the local computer is http://localhost/reports. After the browser connects to that site, navigate the links to the RT project and open the new report. Notice the scroll buttons at the top of the form to move to a new page, and remember that each page contains data for one year. Initially, the report displays the subtotals for each month. However, because the drilldown option was selected earlier, managers can click the + button in front of a given year and the month will be expanded to show the totals for each model type. This approach is useful for displaying multiple levels of data. Each item added as a Group entry creates a new level that can be expanded or contracted. Still, the approach is limited to the report layout specified in the design. More flexible options are available with the cube browser described in Chapter 3.

It is also possible to add selection boxes to the top of the page to set parameters that can be used in the query to control which data is displayed on the report. Additional calculations can be added to display percentages and other totals. All of these changes are made on the design of the report by adding labels, text boxes, and drop-down lists. Some of these tasks have little tricks (such as using Paramters.pname.Label to display a parameter's value) and the details are not covered in this book. Even managers with minimal background in development can learn to build relatively complex reports, but this book focuses more on the exploration of data.



Sales

SaleID	117	SaleDate	3/3/2009
Customer	Jones	Salesperson	Johnson
Phone	(312) 555-1234		

ItemsSold

ItemID	Category	Description	Price	Quantity	Extended
1154	Shoes	Red Boots	$100.00	2	$200.00
3342	Electronics	LCD-40 inch	$1,000.00	1	$1,000.00
7653	Shoes	Blue Suede	$50.00	4	$200.00

Subtotal $1,400.00

Record: 1 of 12 No Filter Search

Figure 2.47

The order form is used in almost any firm. We need to determine the best way to store the data that is collected by this form.

Database Design Concepts

How do you create a new database? This section is optional. It introduces the techniques used to design database tables. Most managers will not be required to create databases, and if they do, a database expert should be consulted to evaluate the design. In a data mining context, most tables already exist; however, it is sometimes useful to understand the foundations of designs to see why some columns are included in specific tables and others are not. Database management systems are powerful tools with the ability to present data in many ways. They are used by managers to answer many different types of questions. However, this flexibility is not automatic. Databases need to be carefully designed; otherwise, managers will not be able to get the information they need. Poor design also leads to unnecessary duplication of data. Duplication wastes space and requires workers to enter the same data several times. **Normalization** is an important technique to design databases.

To understand the process of normalization, consider the example of retail sales. You begin by thinking about who will be using the database and identifying what data they will need. Consider the situation of the salespeople. They first identify the customer then record each item being purchased. The computer should then calculate the amount of money due along with any taxes. Figure 2.47 shows a sample input screen that might be used.

The key design point is that you will need multiple tables to store the data. If you try to cram all of it into a single table, you will end up with unnecessary duplication of data and plenty of problems when you try to delete or insert new data. Each entity or object on the form will be represented by a separate table. For this

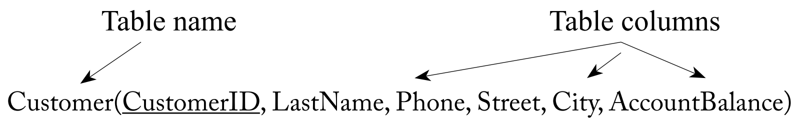


Table name
Table columns
Customer(CustomerID, LastName, Phone, Street, City, AccountBalance)

CID	Last Name	Phone	Street	City	Balance
12345	Jones	(312) 555-1234	125 Elm Street	Chicago	$197.54
28764	Adamz	(602) 999-2539	938 Main Street	Phoenix	$526.76
29587	Smitz	(206) 676-7763	523 Oak Street	Seattle	$353.76
33352	Sanchez	(303) 444-1352	999 Pine Street	Denver	$153.00
44453	Kolke	(303) 888-8876	909 West Avenue	Denver	$863.39
87535	James	(305) 777-2235	374 Main Street	Miami	$255.93

Figure 2.48

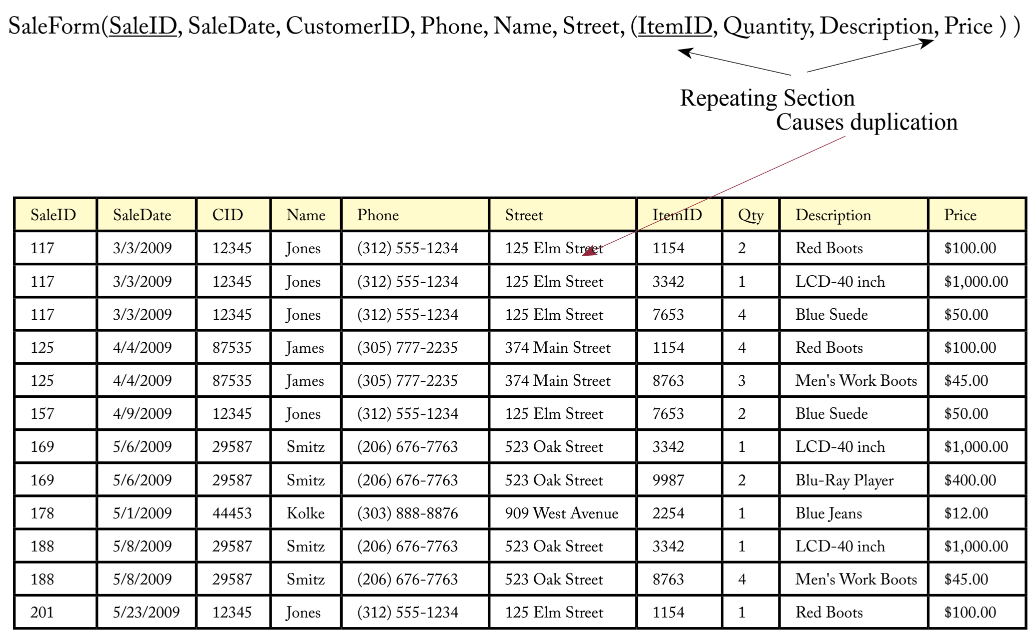
Notation for tables. Table definitions can often be written in one or two lines. Each table has a name and a list of columns. The column (or columns) that makes up the primary key is underlined.

example, there are five objects on the form: Customers, Salespeople, Items, Sale, and ItemsSold..

Before explaining how to derive the five tables from the form, you need to understand some basic concepts. First, remember that every table must have a primary key. A primary key is one or more columns that uniquely identify each row. For example, you anticipate problems with identifying customers, so each customer will be assigned a unique ID number. Similarly, each item is given a unique ID number. There is one drawback to assigning numbers to customers: you cannot expect customers to remember their number, so you will need a method to look it up. One possibility is to give everyone an ID card imprinted with the number—perhaps printed with a bar code that can be scanned. However, you still need a method to deal with customers who forget their cards. It is usually better to build a method to lookup customers by name.

The second aspect to understand when designing databases is the relationships between various entities. First, observe that there are two sections to the form: (1) the main sale that identifies the transaction, the customer, the salesperson, and the date, and (2) a repeating section that lists the items being purchased. Each customer can buy several different items at one time. There is a **one-to-many** relationship between the Sale and the ItemsSold sections. As you will see, identifying one-to-many relationships is crucial to proper database design.

In some respects, designing databases is straightforward: There are only three basic rules. However, database design is often interrelated with systems analysis. In most cases, you are attempting to understand the business at the same time the database is being designed. One common problem that arises is that it is not always easy to see which relationships are one-to-many and which are one-to-one or many-to-many.


Repeating Section
Causes duplication

SaleForm(SaleID, SaleDate, CustomerID, Phone, Name, Street, (ItemID, Quantity, Description, Price))

SaleID	SaleDate	CID	Name	Phone	Street	ItemID	Qty	Description	Price
117	3/3/2009	12345	Jones	(312) 555-1234	125 Elm Street	1154	2	Red Boots	$100.00
117	3/3/2009	12345	Jones	(312) 555-1234	125 Elm Street	3342	1	LCD-40 inch	$1,000.00
117	3/3/2009	12345	Jones	(312) 555-1234	125 Elm Street	7653	4	Blue Suede	$50.00
125	4/4/2009	87535	James	(305) 777-2235	374 Main Street	1154	4	Red Boots	$100.00
125	4/4/2009	87535	James	(305) 777-2235	374 Main Street	8763	3	Men's Work Boots	$45.00
157	4/9/2009	12345	Jones	(312) 555-1234	125 Elm Street	7653	2	Blue Suede	$50.00
169	5/6/2009	29587	Smitz	(206) 676-7763	523 Oak Street	3342	1	LCD-40 inch	$1,000.00
169	5/6/2009	29587	Smitz	(206) 676-7763	523 Oak Street	9987	2	Blu-Ray Player	$400.00
178	5/1/2009	44453	Kolke	(303) 888-8876	909 West Avenue	2254	1	Blue Jeans	$12.00
188	5/8/2009	29587	Smitz	(206) 676-7763	523 Oak Street	3342	1	LCD-40 inch	$1,000.00
188	5/8/2009	29587	Smitz	(206) 676-7763	523 Oak Street	8763	4	Men's Work Boots	$45.00
201	5/23/2009	12345	Jones	(312) 555-1234	125 Elm Street	1154	1	Red Boots	$100.00

Figure 2.49

Converting to notation. The basic rental form can be written in notational form. Notice that repeating sections are indicated by the inner parentheses. If you actually try to store the data this way, notice the problem created by the repeating section: Each time a customer checks out a video we have to reenter the phone and address.

Notation

It would be cumbersome to draw pictures of every table that you use, so you usually write table definitions in a standard notation. The base customer table is shown in Figure 2.48, both in notational form and with sample data.

Figure 2.48 illustrates another feature of the notation. You denote one-to-many or repeating relationships by placing parentheses around them. Figure 2.49 represents all the data shown in the input screen from Figure 2.47. The description is created by starting at the top of the form and writing down each element that you encounter. If a section contains repeating data, place parentheses around it. Preliminary keys are identified at this step by underlining them. However, you might have to add or change them at later steps. You can already see some problems with trying to store data in this format. Notice that the same customer name, phone, and address would have to be entered several times.

Remember that some repeating sections are difficult to spot and might consist of only one column. For example, how many phone numbers can a customer have? Should the Phone column be repeating? In the case of the retail store, probably not, because you most likely want to keep only one number per customer. In other businesses, you might want to keep several phone numbers for each client. Data normalization is directly related to the business processes. The tables you design depend on the way the business is organized.

SaleID	SaleDate	CID	Name	Phone	Street	ItemID, Quantity, Description, Price
117	3/3/2009	12345	Jones	312-555-1234	125 Elm Street	1154, 2, Red Boots, $100.00 3342, 1, LCD-40 inch, $1,000.00 7653, 4, Blue Suede, $50.00
125	4/4/2009	87535	James	305-777-2235	374 Main Street	1154, 4, Red Boots, $100.00 8763, 3, Men's Work Boots, $45.00
157	4/9/2009	12345	Jones	312-555-1235	125 Elm Street	7653, 2, Blue Suede, $50.00
169	5/6/2009	29587	Smitz	206-676-7763	523 Oak Street	3342, 1, LCD-40 inch, $1,000.00 9987, 2, Blu-Ray Player, $400.00
178	5/1/2009	44453	Kolke	303-888-8876	909 West Ave.	2254, 1, Blue Jeans, $12.00
188	5/8/2009	29587	Smitz	206-676-7763	523 Oak Street	3342, 1, LCD-40 inch, $1,000.00 8763, 1, Men's Work Boots, $45.00
201	5/23/2009	12345	Jones	312-555-1234	125 Elm Street	1154, 1, Red Boots, $100.00

Figure 2.50

A table that contains repeating sections is not in first normal form. Each table cell can contain only basic data. Storing it in repeating form makes it difficult to search, insert, and delete data. This version is not in first normal form.

Figure 2.51

Splitting a table to solve problems. Problems with repeating sections are resolved by moving the repeating section into a new table. Be sure to include the old key in the new table so that you can connect the tables back together.

SaleForm(<u>SaleID</u>, SaleDate, CID, Phone, Name, Street, (<u>ItemID</u>, Quantity, Description, Price))

SaleForm2(<u>SaleID</u>, SaleDate, CustomerID, Phone, Name, Street)

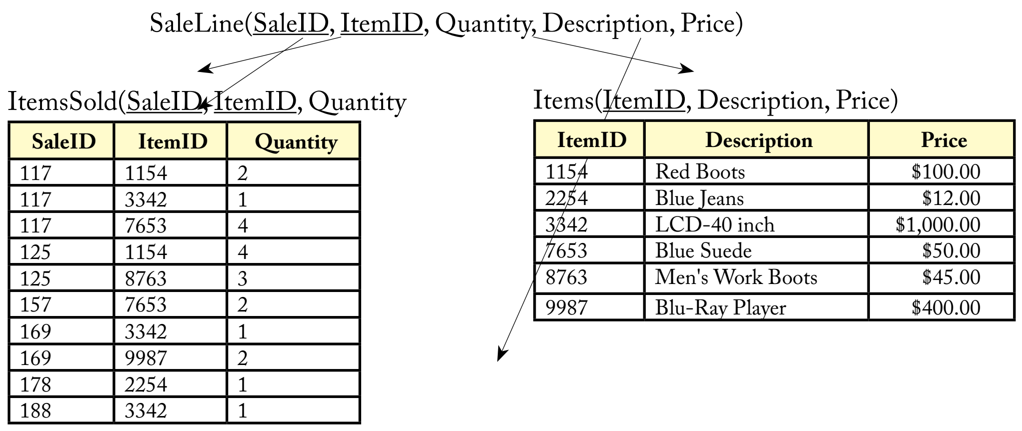
Note replication

SaleID	SaleDate	CID	Name	Phone	Street
117	3/3/2009	12345	Jones	(312) 555-1234	125 Elm Street
117	3/3/2009	12345	Jones	(312) 555-1234	125 Elm Street
117	3/3/2009	12345	Jones	(312) 555-1234	125 Elm Street
125	4/4/2009	87535	James	(305) 777-2235	374 Main Street
125	4/4/2009	87535	James	(305) 777-2235	374 Main Street

SaleLine(<u>SaleID</u>, <u>ItemID</u>, Quantity, Description, Price)

Note replication

SaleID	ItemID	Quantity	Description	Price
117	1154	2	Red Boots	$100.00
117	3342	1	LCD-40 inch	$1,000.00
117	7653	4	Blue Suede	$50.00
125	1154	4	Red Boots	$100.00
125	8763	3	Men's Work Boots	$45.00
157	7653	2	Blue Suede	$50.00
169	3342	1	LCD-40 inch	$1,000.00
169	9987	2	Blu-Ray Player	$400.00
178	2254	1	Blue Jeans	$12.00
188	3342	1	LCD-40 inch	$1,000.00
188	8763	4	Men's Work Boots	$45.00
201	1154	1	Red Boots	$100.00



SaleLine(SaleID, ItemID, Quantity, Description, Price)

ItemsSold(SaleID, ItemID, Quantity

SaleID	ItemID	Quantity
117	1154	2
117	3342	1
117	7653	4
125	1154	4
125	8763	3
157	7653	2
169	3342	1
169	9987	2
178	2254	1
188	3342	1

Items(ItemID, Description, Price)

ItemID	Description	Price
1154	Red Boots	$100.00
2254	Blue Jeans	$12.00
3342	LCD-40 inch	$1,000.00
7653	Blue Suede	$50.00
8763	Men's Work Boots	$45.00
9987	Blu-Ray Player	$400.00

Figure 2.52

Second normal form. Even though the repeating sections are gone, we have another problem. Every time the ItemID is entered, the description has to be reentered, which wastes a lot of space. There is a more serious problem: if no one has purchased a specific item yet, there is no way to find its description or price since it is not yet stored in the database. Again, the solution is to split the table. In second normal form, all nonkey columns depend on the whole key (not just part of it).

First Normal Form

Now that you have a way of writing down the assumptions, it is relatively straightforward to separate the data into tables. The first step is to split out all repeating sections. Think about the problems that might arise if you try to store the repeating data within individual cells. You will have to decide how many rows to set aside for storage, and you will have to write a separate search routine to evaluate data within each cell, and complications will arise when inserting and deleting data. Figure 2.50 illustrates the problem.

The answer to this problem is to pull out the repeating section and form a new table. Then, each item purchased by a customer will fill a new row. Figure 2.51 uses the notation to show how the table will split. Notice that whenever you split a table this way, you have to bring along the key from the prior section. Hence, the new table will include the SaleID key as well as the ItemID key. When a table contains no repeating sections, you say that it is in first normal form.

Second Normal Form

Even if a table is in first normal form, there can be additional problems. Consider the SaleLine table in Figure 2.51. Notice there are two components to the key: SaleID and ItemID. The nonkey items consist of the Quantity, Description, and Price of the item. If you leave the table in this form, consider the situation of selling a new item. Every time an item is sold it will be necessary to reenter the Description and list Price. It means that you will be storing the description every time an item is sold. Popular items might be sold thousands of times. Do you really want to store the description (and other data) each time?

The reason you have this problem is that when the SaleID changes, the item description stays the same. The description depends only on the ItemID. If the Price

SaleForm2(SaleID, SaleDate, CustomerID, Phone, Name, Address)

Sales(SaleID, SaleDate, CustomerID, SalespersonID)

SaleID	SaleDate	CID	SPID
117	3/3/2009	12345	887
125	4/4/2009	87535	663
157	4/9/2009	12345	554
169	5/6/2009	29587	255
178	5/1/2009	44453	663
188	5/8/2009	29587	554

Customers(CustomerID, Phone, Name, Address, City, State, ZIPCode, AccountBalance)

CID	Name	Phone	Street	City	Balance
12345	Jones	(312) 555-1234	125 Elm Street	Chicago	$197.54
28764	Adamz	(602) 999-2539	938 Main Street	Phoenix	$526.76
29587	Smitz	(206) 676-7763	523 Oak Street	Seattle	$353.76
33352	Sanchez	(303) 444-1352	999 Pine Street	Denver	$153.00
44453	Kolke	(303) 888-8876	909 West Avenue	Denver	$863.39
87535	James	(305) 777-2235	374 Main Street	Miami	$255.93

Figure 2.53

Third normal form. There is another problem with this definition. The customer name does not depend on the key (SalesID) at all. Instead, it depends on the CustomerID. Because the name and address do not change for each different SalesID, the customer data must be in a separate table. The Sales table now contains only the CustomerID, which is used to link to the Customers table and collect the rest of the data. The same rule applies to Salespeople.

represents the list price of the item, the same dependency holds. However, what if the store offers discounts on certain days or to specific customers? If the price can vary with each transaction, the price would have to be stored with the SaleID. The final choice depends on the business rules and assumptions. Most companies resolve the problem by creating a list price that is stored with the item and a sale price that is stored with the transaction. However, to simplify the problem, stick with just the list price for now.

When the nonkey items depend on only part of the key, you need to split them into their own table. Figure 2.52 shows the new tables. When each nonkey column in a table depends on the entire key, the table is in second normal form.

Third Normal Form

Examine the SaleForm2 table in Figure 2.51. Notice that because the primary key consists of only one column (SaleID), the table must already be in second normal form. However, a different problem arises here. Again, consider what happens when you begin to collect data. Each time a customer comes to the store and buys something there will be a new transaction. In each case, you would have to record the customer name, address, phone, city, and so on. Each entry in the transaction table for a customer would duplicate this data. In addition to the wasted space, imagine the problems that arise when a customer changes a phone number. You might have to update it in hundreds of rows.

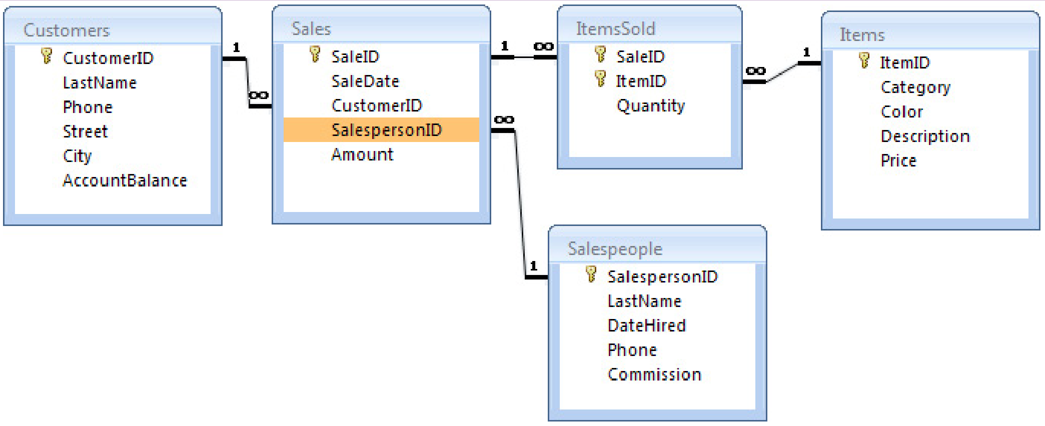


Figure 2.54

Third normal form tables. There are no repeating sections and each nonkey column depends on the whole key and nothing but the key. This figure also shows the relationships between the tables that will be enforced by the DBMS. When referential integrity is properly defined, the DBMS will ensure that rentals can be made only to customers who are defined in the Customers table.

The problem in this case is that the customer data does not depend on the primary key (SalesID) at all. Instead, it depends only on the CustomerID column. Again, the solution is to place this data into its own table. Figure 2.53 shows the split. Splitting the table solves the problem. Customer data is now stored only one time for each customer. It is referenced back to the Rentals table through the CustomerID. The same rule applies to Salespeople, resulting in the fifth table.

The five tables you created are listed in Figure 2.54. Each table is now in third normal form. It is easy to remember the conditions required for third normal form. First: There are no repeating groups in the tables. Second and third: Each nonkey column depends on the whole key and nothing but the key.

Note in that if the Customers table contains complete address data, including ZIP Code, you could technically split the Customers table one more time. Because ZIP codes are uniquely assigned by the post office, the city and state could be determined directly from the ZIP code (they do not depend on the CustomerID). In fact, most mail order companies today keep a separate ZipCode table for that very reason. For our small retail firm, it might be more of a nuisance to split the table. Although you can purchase a complete ZIP code directory in computer form, it is a large database table and must be updated annually. For small cases, it is often easier to leave the three items in the Customer table.

Summary

Databases, particularly relational database systems, hold much of the data used in business. Most business applications, including enterprise systems and Web sites, store data in relational databases. SQL Server Analysis Services is closely tied to the SQL Server database system. For these reasons, you need to know how to create queries to retrieve data. Queries can be built with the design editor but they are actually built in SQL. The four main steps in building a query are: (1) Identify the output needed, (2) Specify the constraints, (3) Select the tables holding the data, and (4) Indicate how the tables are joined.

SQL can perform some basic computations. Arithmetic is handled on a row-by-row basis and new columns can be defined using data on a single row at a time. SQL Server support several functions to perform additional computations, such as standard mathematical functions, string manipulation, and date calculations and formatting. Aggregations (Sum, Count, Avg, and so on) are performed across rows of data. Subtotals are computed using the GROUP BY statement. Common business questions involve subtotals for each customer, employee, state or region, and item category. Questions involving complex nested subtotals are better handled with hyper cube browsers covered in the next chapter.

SQL has many powerful features useful for formatting and cleaning data sets. The UPDATE, INSERT, and DELETE commands are useful for transferring data. This chapter touched on the basic capabilities of SQL. People who need to focus on building data warehouse systems should study the advanced SQL options in more detail—in a database textbook.

Key Words

aggregation
BETWEEN
Boolean algebra
columns
cross join
data definition
data manipulation
data type
database
database management system (DBMS)
DESC
enterprise resource planning (ERP)
FROM
GROUP BY
HAVING
identity
JOIN
normalization
NOT
one-to-many
ORDER BY
primary key
query system
row-by-row calculations
SELECT
SQL
SQL Server Business Intelligence (BI)
table
view
WHERE

Review Questions

1. What is a table and why is it important in a relational database?
2. What four questions must be answered to create a query and what are the corresponding parts of the SQL statement?
3. What is the command word for matching portions of a string in SQL?
4. Why is it critical to store dates using a datetime format?
5. How are aggregation functions such as Sum different from arithmetic calculations (+/-)?
6. What is the purpose of the GROUP BY function?
7. In common problems involving subtotals created in the design editor, such as sales by customer, why is it often necessary to specify a WHERE option when a date condition is included?
8. What is the syntax for joining two tables in SQL (for example Bicycle and Customer)?
9. What is the typical role of database reports?

Exercises

For the SQL exercises, just submit the SQL statements, not the query results.

Book

1. List the tables in the Rolling Thunder Bicycles database.
2. SQL. Which race bicycles were ordered in 2012 with a framesize greater than 60 cm?
3. SQL. Which mountain bicycles in 2011 had a list price over 7000?
4. SQL. What is the total value of all bikes sold in November 2012?
5. SQL. What is the total number of bicycles sold of each model type in March 2012?
6. SQL. Who was the best sales person by value in 2011?
7. SQL. Which customer from California spent the most on bikes from 2010 through 2012?
8. SQL. Which model types were not sold in December 2012? (Hard: See database book)

Rolling Thunder Database

9. SQL. Who are the top customers who bought the most bikes?

10. SQL. In which state were the most bicycles sold in 2010?

11. SQL. List the manufacturers and the total value of cranks purchased from them in 2012.

12. SQL. What was the most popular paint color for mountain bicycles (including full suspension) in 2011?

13. SQL. Create a query to list all Customers, Bicycles, and Components installed on bicycles in 2010

14. SQL. Create a query to list bicycle sales by month and model type by count and value, where month is displayed in the form YYYYMM such as 199401.

Diner

15. What are the columns in the Diners table?

16. SQL. How does the average bill compare for Lunch, Dinner, and Evening?

17. SQL. What are the average sales by day of week?

18. SQL. What are the average sales by gender of the group?

19. SQL. How many times did a Mixed group order dessert? How many times did they not order dessert? Hint: Both can be answered with one query.

Corner Med

20. List the tables in the Corner Med database.

21. SQL. Create a view to compute the number of patients and total amount billed by day.

22. SQL. What is the average number of patients seen per day and the standard deviation? Hint: Use the view from #21.

23. SQL. What is the average amount of money billed per day and the standard deviation? Hint: Use the view from #21.

24. SQL. Which physician has treated the most patients (based on visits)?

25. SQL. What is the most common diagnosis based on a first three letters of the diagnosis code? Hint: Use the Substring function.

Basketball

26. List the tables in the basketball database.

27. SQL. Which player scored the most points in the regular season 2010-2011 (not playoff)?

28. SQL. Which player had the most total steals plus blocks plus defensive rebounds in the regular season 2010-2011?

29. SQL. Which player with at least 20 free throw attempts had the worst free throw percentage in 2009-2010? Hint: Cast(FT as real).

30. SQL. List the total number of wins by each team in the regular season 2010-2011 (not playoff), ordered by the number of wins.

Bakery

31. List the tables in the Bakery database.

32. SQL. What is the total value of sales by year?

33. SQL. What is the total value of sales by category in 2009?

34. SQL. In 2010, what was the best-selling item in the Muffin category (by count)?

35. SQL. What is the average number of items and average value of a Sale (basket) in 2012? Hint: First create a view that computes values by sale.

Cars

36. What are the columns in the Cars database?

37. SQL. What is the average MPG and average price by category?

38. SQL. What is the average weight and average price by Make (company)?

39. SQL. List the vehicles in descending order of Power/Weight ratio. Hint: Use Cast(HP as real).

Teamwork

40. Each person in the group should choose one of the databases and write a business question (not in the existing exercises). Share each question with the other team members and write the query to answer all of the questions. Compare the answers by each team member.

Additional Reading

Post, Gerald, 2011, *Database Management Systems*: http://www.JerryPost.com/Books/DBBook. [Textbook on standard database management systems, covering design, queries, applications, and management.]