Essentials Plus of SQL: Extracting and Updating Data

AUTHORS

Gove Allen, PhD

Brigham Young University

Gary Hansen, PhD

Brigham Young University

Robert Jackson, PhD

Brigham Young University

Introduction: Essentials of SQL Interactive Textbook

0.1Layout of the Interactive Book Pages

The *Essentials of SQL* interactive textbook is presented in the form of web pages and can be viewed from any device that is connected to the internet. The pages of the book are divided into three panels. Each of the two side panels can be expanded or collapsed by clicking on the three dots on the left margin or right margin of the center panel.

The left panel contains a table of contents for the chapter that is active. The section titles in the table of contents are hotlinks that transfer viewing to the chosen section.

The right panel provides hotlinks and controls for utilizing the textbook tools. These tools are as follows:

- Search box: Enter a search term, and a list of all references throughout the book is displayed. Each entry is a hotlink to the corresponding book section.
- Listen: Allows you to listen to the textbook using Read Speaker.
- Glossary: Opens a pop-up window that contains all the terms that have been defined throughout the book. The Go To button for each term transfers control to the place in the textbook where the term was defined. Hovering a mouse cursor over the term displays the definition.
- Flashcards: A learning practice tool. It presents glossary terms in a random order with the term on one side and the definition on the other side.
- Notebook: Opens up the notebook where you have recorded your notes. It works in conjunction with the highlighter.
- Highlighting: Highlighting allows you to highlight text using the mouse cursor and to add a note, which is added to the notebook.

- Advanced Options: Allows you to configure the book. See the explanation in the next section.
- Support: Opens up the Knowledge Base with many Frequently Asked Questions (FAQs). **It is recommended that you take a few minutes to review the FAQ page to be aware of how to use MyEducator in your particular environment.**

0.2Advanced Options

The textbook is designed to be able to configure itself according to environment in which it is used. The Advanced Options function is used to set the parameters for the appropriate configuration. There are two main sets of features that can be changed dynamically: the database management system or engine (DBMS) and the data modeling method that is used for data modeling diagrams. Figure 0.1 illustrates the Advanced Options pop-up window showing the configuration options.

The textbook and the underlying database access support four distinct database engines or DBMSs: Microsoft SQL Server, Oracle, MySQL, and PostgreSQL. Use the radio buttons to choose the appropriate DBMS.

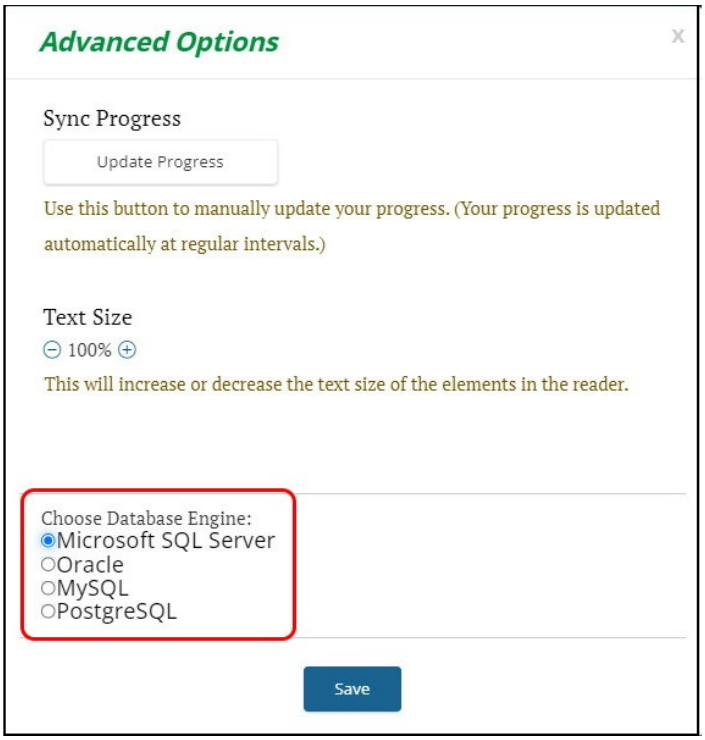

Figure 0.1: Advanced Options

Note that there are also separate sets of assessments for each chapter that are consistent with the distinct DBMS and the distinct modeling method that have been selected.

0.3Query Boxes and Executing SQL

Query Boxes within the Text

It is well known that practicing skills along with understanding the concepts is a powerful way to learn and to develop new skills. To facilitate this process, query boxes have been placed throughout the textbook to demonstrate SQL statement execution and so that you can practice writing SQL statements.

Figure 0.2 illustrates a typical query box. The SQL in the query box is a valid statement illustrating correct syntax and form. Clicking on the “Run Query”

button sends the query to the appropriate DBMS, which executes the query and returns the result. Note the DBMS that receives the query is the one that is assigned by the Advanced Options selection, which was explained earlier. The query box includes, behind the scenes, a connection string to connect to the DBMS and then to run the query and capture the result to format and display on the page.

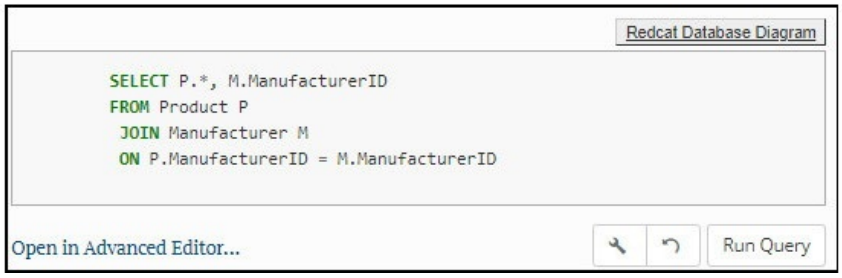

Figure 0.2: Query Box for entering SQL Statements

Right above the query box is a link to the database model that is connected to that query box. Clicking on that link opens a new tab that displays the database model. Displaying the database model facilitates writing the SQL statements.

By clicking anywhere within the query box, the query box goes into edit mode, which permits you to edit the SQL statement or write a new SQL statement. Multiple SQL statements may be written in the box but must be separated by semicolons. The semicolon is the standard termination character for SQL statements for all DBMSs.

The reset icon (half circle with arrow), removes the result display and resets the original SQL statement. The wrench icon at the bottom of the query box displays a settings pop-up window. Figure 0.3 displays the settings box. These are the settings that define how the results are displayed. The default parameters within the settings pop-up window work well, but you may change them as desired.

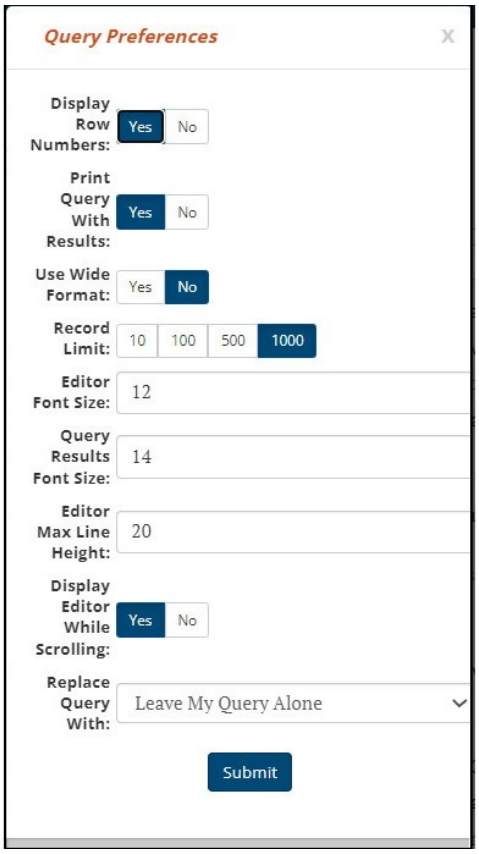

Figure 0.3: Results Settings for Queries

The Open in Advanced Editor option takes the SQL statement that is extant in the query box and opens up another page with the Advanced SQL Editor. The Advanced Editor is explained next.

Advanced Editor Page

The Advanced Editor page is similar to the query boxes and executes SQL queries in the same way. It does provide more options to control both the DBMS being used to run the queries and the database being accessed. Figure 0.4 illustrates the Advanced Editor with the two drop-down menus displayed.

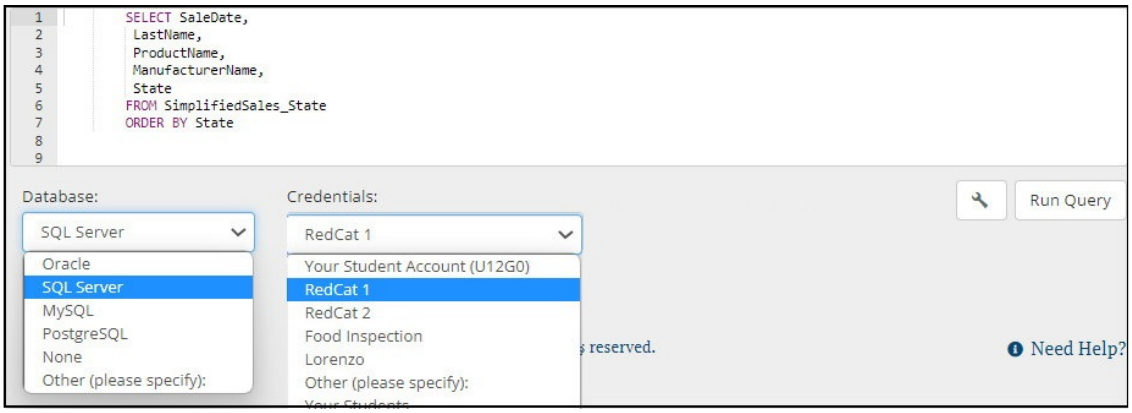

Figure 0.4: Advanced Query Page

The query box contains the original query from the query box in the textbook. It is in edit mode so that you can modify and enter your own SQL statements.

The Database drop-down menu is used to select which DBMS will be used to execute the query. The four supported DBMSs, Oracle, SQL Server, MySQL, and PostgreSQL, are listed first. The “Other (please specify)” option is designed to work with another DBMS on the MyEducator server. Selecting that option opens up another text box for the connection string. The “None” option is simply a placeholder.

The Credentials drop-down menu is used to select which database is to be used. The “Your Student Account (…)” option is the important selection for modifying the database. All of the other databases are “read-only” databases and are protected so that students always get correct values on valid queries. However, as a student you are given an empty sandbox where you can create your own database and insert, modify, and delete data. This option connects you to your sandbox database.

RedCat 1 is a denormalized and simplified version of the RedCat shoes database that contains the SimplifiedSales table. The RedCat 2 database is the full RedCat shoes database. The Lorenzo database is for Lorenzo Shipping. The Food

Inspection is the Chicago Food Inspection database. The data models for both the Lorenzo Shipping database and the Food Inspection database are given in the Appendix.

Chapter 1: Databases and SQL: Everywhere for Everybody

1.1Introduction: The Unseen Services

Learning Objectives

After reading this chapter, you will be able to:

- Identify several large public databases
- Describe the need for structure and organization in database design
- Describe the characteristics of information
- Explain the differences between OLAP and OLTP databases
- Explain the major components of a DBMS
- Describe the importance and use of a data warehouse
- Describe the need for database knowledge in business and technology careers

In today's modern societies, we depend on many services from organizations and governments that we take for granted. Often, we are not even aware of the underlying and invisible components that make these services available to us. For example, we live in houses, go to school and work in buildings, exercise in gymnasiums, and attend sports activities in stadiums. We observe and interact with the visible components of these edifices, but we seldom think about the unseen components that provide the comfort and stability that we enjoy on a daily basis.

Databases play the same role in our information-rich, connected, and mobile world. Of course, almost everything we do with an electronic device depends on some underlying databases. In most cases, we do not even think about the source and format of the information that we are retrieving.

Let's examine some examples.

- **Log in and use Facebook:** Your login and personal information is kept in

a database, as is the information for all of your friends, posts, and messages.

- **Make a phone call or send a text message:** The lists of phone numbers and locations in your cell phone is maintained by databases. All the information about your phone calls and text messages is maintained at a very detailed level in databases.
- **Read your email:** Not only is your login information kept in a database, but the history of all incoming and outgoing emails is also kept in very large databases.
- **Go shopping (groceries, clothes, gasoline):** First, the product, inventory levels, and price information is kept in a database so that the checkout register can identify it correctly. Then (if you use a credit or debit card), that information must be retrieved from another database.
- **Check a book out of the library:** The database is checked to make sure you are a valid patron of the library. Then, information about the book, its availability, and checkout status is all verified and updated for your library loan. The database is used to obtain both patron information and book information.
- **Withdraw money from an ATM:** First, your authorization information must be verified, and then the availability of sufficient funds is also verified. Finally, your account balances must be updated. All this information is stored in the bank's databases.
- **Have a physical exam with your doctor:** All of your patient information is maintained in your medical database. Your records will be updated from the results of your exam. The accounting, billing, and insurance records will also all be updated.

We seldom think about these databases, unless they become unavailable or there has been a security breach where private data has been accessed by unauthorized individuals or made public. When they don't work correctly, we get frustrated

and upset that the level of service that we expect is not being provided. The availability and integrity of databases is an important issue in the daily activities of each of our lives.

1.2Databases: The Ubiquitous Information Provider

We are all aware that we live in an information age. In fact, we could say that there are three major characteristics of today's society: pervasive connectivity, universal mobility, and abundant information. The pervasive connectivity and universal mobility also contribute to the abundance of information that is available to us. However, just as we saw in the previous section, the primary contributor to the provision of information is database technology. We also saw in the previous section how databases and database technology provide the foundation for many of our daily activities of business and commerce. In this section, let's investigate our need to obtain more in-depth information to see how databases provide support.

- **Searching to buy a specific item (e.g., an appliance, a service, clothing, a book):** We have all used an internet search provider. The information provided is usually a list of the particular item for which we are searching, as identified by a keyword. It is mind-boggling to think of the size and complexity of the databases that provide indexes to find what we want. The amount of data that must be stored, searched, and retrieved to answer a simple query for a product or service in a particular location is staggering.
- **Conducting in-depth research before purchasing a major item such as a car:** The information you need to buy a car includes items like the car's repair history, performance data, and safety record. You may also be interested in other people's opinions. This information can be obtained in such places as Consumer Reports, which is maintained in databases. You might find people's opinions in ratings, comments, or even in some blogs;

these are all provided by databases.

- **Planning a vacation or trip:** In this case, we want geographical and resort information as well as information about sites and activities to see and do. We also seek information about travel arrangements, including schedules, costs, and availability. Information systems may have to access multiple databases—airline databases, detailed resort or accommodations databases, site databases, activities databases, and even geological road map databases.
- **Learning about a particular hobby or sport that you enjoy, such as skiing, boating, or coin collecting:** Almost all hobbies and sports have organizations and clubs that generate related articles and conferences. The articles and information are largely stored in databases maintained by these organizations.
- **Researching a particular company before investing in it:** Many institutions maintain databases of financial records. There are databases that provide detailed histories of every publicly-traded company. Some of these databases are publicly available, but even more details are available through private databases maintained by individual financial service companies.
- **Researching a particular medical condition or medical problem:** There are also extensive databases with information about various illnesses, medications, and treatments. Many individuals also comment and write blogs about personal experiences with particular illnesses. Of course, not all of this information is valuable and correct. However, databases in almost every case provide the information repository.
- **Solving a technical problem or learning a new technical skill such as administering a Linux server:** One of the interesting phenomena in today's technical world is that for almost every problem that a developer or technical person encounters, someone else has already encountered it and there is a solution or additional training available. Most of these answers are on forums, blogs, or online documentation. All of these information

sources are maintained by databases.

- **Writing a research paper or researching for a school project:** Although many research articles and books are not stored in databases per se, keyword and topic search capability is always supported by search engine databases. In addition, depending on the research project, databases are available from newspapers, research organizations, and online libraries.
- **Browsing Wikipedia:** Who of us has not used Wikipedia from time to time to get a quick understanding of a particular subject? Wikipedia, as well as the keyword indexes, are maintained in a database.

It should be apparent from this section and the previous one that databases impact our lives in a multitude of ways in almost everything we learn and do. The ability to understand, design, build, maintain, and administer databases is an important and powerful skill to have.

1.3Database Skills for Knowledge-Based Careers

It should be evident from the previous examples that an understanding and knowledge of how to use, design, and build databases is essential for anyone desiring a career in information systems. The core of every type of information system is a database.

Every type of technical position requires extensive or substantial database skills. The database skills required range from being able to design and build a database to being able to monitor, tune, and optimize the database performance. However, database skills are not only required by technical people. Every type of knowledge-based career also requires substantial database knowledge and skill.

Table 1.1 identifies a few other types of knowledge-based careers. A knowledge-based career is one that requires an understanding of how to solve problems

using knowledge and analytical skills. A very large percentage of knowledge-based skills now require computer proficiency, along with the ability to understand and use databases to find and extract information. As can be seen from the table, knowledge workers must be able to understand the structure of the data in the database so that they can extract information from the raw data.

Table 1.1
Database Skills Required for Knowledge-based Careers

Job Title	Job Description	DB Knowledge Required	Skill Level Required
Financial Planner	Analyze the security market and recommend financial portfolios	Understand the financial information available in databases. Know how to extract information in various combinations	Moderate
Advertising Manager	Manage advertising campaigns and budgets	Understand sales and performance data. Be able to extract information in various ways.	Moderate
Human Resource Manager	Manage all the hiring, evaluating, and monitoring of the workforce.	Understand information about employees as well as all financial information.	Moderate
Sales Manager	Manage the salespeople and	Understand sales data and be able to extract information in	Moderate

	sales activities of an organization	various forms. Evaluate performance.	
Market Research Analyst	Analyze sales and economic data to predict future trends	Understand economic databases. Be able to use databases in novel ways to discover and extract information.	Substantial
Political Scientist	Understand and research demographic data and trends	Be able to analyze data and draw conclusions from demographic databases and questionnaire data.	Substantial
Management Consultant	Analyze organizational issues and recommend solutions	Understand the financial and operational data of an organization as found in various databases	Substantial
Accountant	Support all the accounting requirements of an organization	Understand all financial databases. Identify problems or potentially fraudulent activity through database analysis	Extensive
Economist	Analyze and predict economic trends based on	Understand economic databases, their structure, and data. Be able to extract	Extensive

	historical data	information in various forms using novel approaches.	
Sociologist	Analyze social trends and social problems in communities and nations	Understand demographic and social data from diverse and disparate databases. Be able to extract information using multiple techniques	Extensive
Public Administrator	Manage a public government unit	Be able to understand data and information provided in various and diverse databases from the governmental database.	Extensive

1.4The Database Approach

An information system that uses a **database management system (DBMS)** to manage its information has a particular structure, comprising three components: DBMS, data, and application software. This structure as described below is referred to as the database approach to information system development.

The central component of the database approach is the DBMS. This software is also referred to as the *database engine* or the *back end*. With regard to the data it manages, it has several responsibilities, including the following:

- **Data Definition:** providing a way to define and build the database
- **Data Manipulation:** providing a way to insert and update data in the database

- **Query Execution:** answering questions about the data in the database
- **Data Integrity:** ensuring that data stored is well formed
- **Data Security:** enforcing restrictions about who is able to access what data
- **Data Portability:** providing a means for backup and restore
- **Data Recovery:** protecting data from loss in the case of a catastrophic failure in hardware of software
- **Provenance:** logging capabilities to provide an audit trail for data changes
- **Performance:** providing a means to tune and optimize operation
- **Multiuser Concurrency:** supporting the activities of many users at the same time
- **Automatic Processing:** providing a way to define rules to execute business logic (e.g. stored procedures and triggers)

As can be seen from the above list, a DBMS is a complex software application. While all database management systems may not provide all of these features listed, these features are the general characteristics of today's DBMSs. Using a database requires considerable expertise and knowledge about the specific DBMS being used. Some of the more popular DBMS's today are MySQL, Microsoft SQL Server, Oracle, PostgreSQL, Microsoft Access, and IBM's DB2.

The second component in the database approach is the data. Although the physical location or manner in which the data are stored may be important for performance reasons, the location of the data does not determine whether or not a system is developed using the database approach. As long as the DBMS has access and is able to perform its responsibilities with respect to the data, the details of the data storage are not relevant. The data could be stored on a single local hard disk, an array of disks on a single computing device, the disks of multiple devices, or in any other configuration.

The final component of the database approach is the application, also called *front end* software. Application software interacts with the DBMS to provide information to a user. It may also provide a way for a user to invoke other functionalities of the DBMS. In fact, the DBMS software itself is non-visual, meaning that the user does not interact directly with the DBMS. Any software that provides an interface for the user to invoke procedures in the DBMS is defined as application software. Application software may provide complex functionality to database administrators or display the results of a fixed information request. Any program that sends a request to the DBMS and receives a response is application software.

Once the application has determined what the user is trying to accomplish, it sends a request to the DBMS. The request may be an instruction to change data or a request for information such as the list of employees who were hired on a particular date. All relational databases use a standard language to receive and process requests. The standard language is called **Structured Query Language**.

The DBMS receives the request and determines if the operation requested is allowed for the authenticated user. If the operation is allowed, the DBMS completes the operation and sends a response to the application. The application then communicates the information to the user. If the operation is not authorized for the user or if there is an error in fulfilling the operation, the DBMS responds with an appropriate message. Again, it is up to the application to display that to the user. It is a critical feature of the database approach that the application never bypasses the DBMS to access stored data directly.

Figure 1.1 illustrates some of the primary components of a typical DBMS and how they are used in an information system.

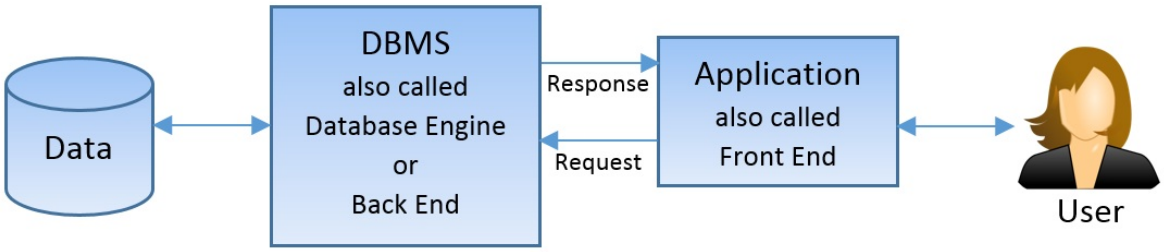

Figure 1.1: The Database Approach

1.5Information Structuring in Databases

Let's briefly look at some of the processes required to codify data and turn it into meaningful information. Interesting questions about information include, "What is information?" "How does it exist in the world?" and "How are databases used to help organize, structure, record, and provide information by identifying new relationships that were not seen previously?"

There are many different definitions and philosophical concepts about information. Our short discussion will barely scratch the surface of defining information. However, we can note that information is more than just data or facts. For our purposes, let us define information as a set of data facts that (1) can be labeled, (2) are identified within a larger context, and (3) have identified relationships with other facts. For example, if we have a piece of data such as "2875039275," we do not have much information. However, if I label it as a phone number, then it begins to contain meaningful information. If I add that this phone number belongs to John Appleby, then I have added both a context and a relationship, and now we have even more available information.

Additional information is added if I identify that the number is a cell phone number; now I know that I can send a text message using that number. We also gain additional information on phone numbers by adding more contextual and relationship information. This might include whether a particular phone number

is a business telephone or a residential telephone, and whether it is land line (with a physical location) or cellular (with a current cell tower location).

As you can see, meaningful information is more than just a set of facts. Databases are uniquely qualified to store a large set of data facts, attach labels to them, maintain data about their context, and establish important relationships.

Entities and Attributes

Every database will contain information about many different entities. An **entity** is a specific type of object or type of thing. It can also be considered a classification of the specific type of items. (Note: We will use the terms entity and entity set interchangeably to refer to the set of items that are the same type.) A database **schema** will describe both the details about each entity and the relationships between entities. Entities usually come from physical world items such as customers or vehicles. However, abstract items, such as purchases or sales, are also entities. Detailed information about the properties of each entity are called **attributes**. For example, attributes of Customers in a database will be things like Name, Address, and Phone Number. The database schema will identify all the entities to be included in the database and the attributes for each entity. The schema describes the structure of the data in the database.

Figure 1.2 is an example of a visual data model of some of the information that is maintained in a database as part of the schema. This diagram is a partial database model for a shoe retail store that maintains detailed information about its customers. Information about each sale is kept in the database along with the individual pairs of shoes that are sold (SaleItems) as part of the sale. Other information includes product information (shoes) as well as the manufacturers of the shoes. As you can see in the figure, there are six different entities as identified by the boxes.

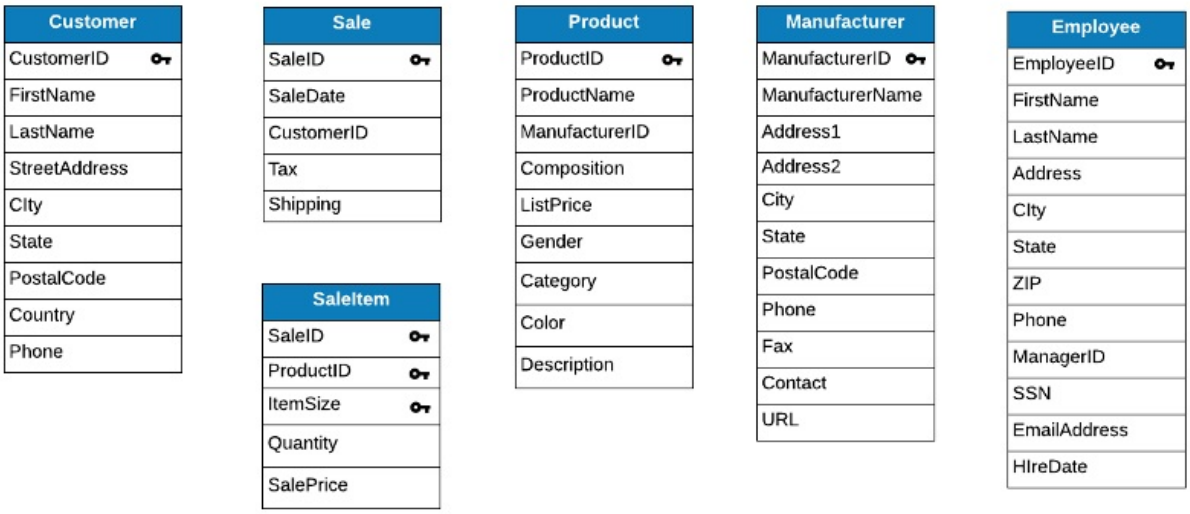

Figure 1.2: Data Entities for a Shoe Retail Store

Within each box is a list of the attributes for each entity. The attributes define the detailed information that is kept about each entity. The entities and their list of attributes are part of the database schema. In Figure 1.2, we see that attributes for Employee are EmployeeID, FirstName, LastName, Address, and so forth.

Each entity has one or more attributes that is defined as a **key** attribute. In the figure above, we have identified those attributes with a small key icon in the attribute rectangle. For example, for a Customer entity, the key attribute is CustomerID. A key is defined as an attribute whose value is unique or distinct for each record in the entity. Notice that a SaleItem requires three fields as the key—SaleID, ProductID, and ItemSize. It takes all three attributes to uniquely identify each SaleItem. Examples of key values are given below in the explanation of data.

Relationships

The database also needs to capture and describe information about the **relationships** between the entities. In this example, we want to know which

sales were done by which customers. Sale Items are associated with each sale. In other words, a sale may include several pairs of shoes. Each sale item, which is a pair of shoes, is described in detail by product information. Finally, each product is manufactured by a particular manufacturer. The visual data model in Figure 1.2 is not sufficient to provide this information. Figure 1.3 expands the model by defining the important relationships. These relationships are identified by lines connecting the boxes of related entities.

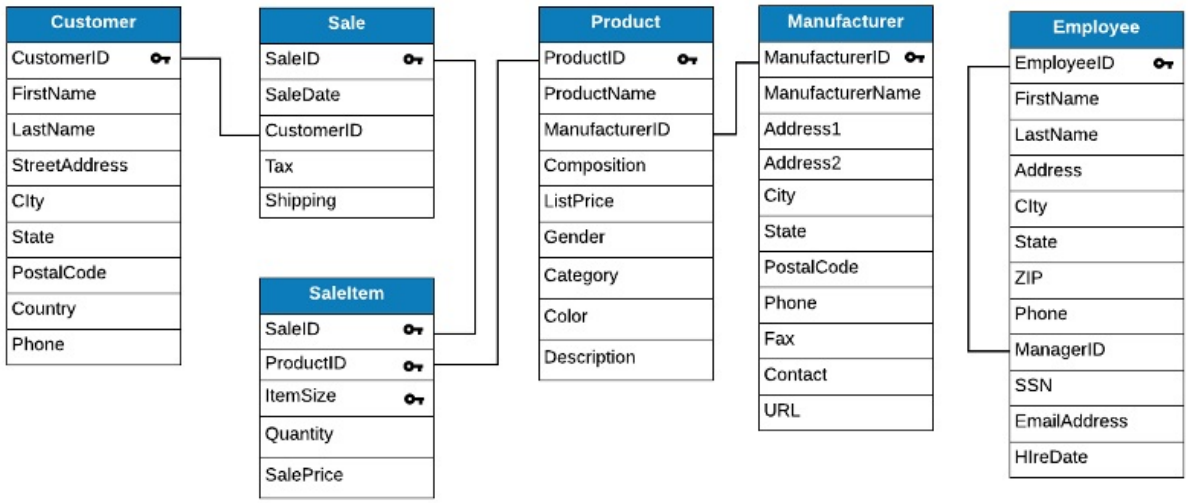

Figure 1.3: Data Entities with Relationships for a Shoe Retail Store

One good way to capture relationship information is by using an entity's key. Because keys uniquely identify a record, if we put the same key value in the data of a related record, then the two records are related to each other. In Figure 1.3 the Sale entity includes the CustomerID attribute, which is another entity that gives information about the customer. Thus, we know which customer purchased items on a particular sale. The Product entity includes the ManufacturerID attribute, which, again, is its own entity; we know who the manufacturer is for any given product. When a key attribute for one entity is placed in a different entity, it is called a **foreign key** in that different entity. In other words, a foreign key is a key attribute that belongs in one entity, but is listed in a different entity. In SaleItem, SaleID is a foreign key because it is the primary key for Sale.

SaleItem's primary key is made up of three, and exactly three, fields: SaleID, ProductID, and ItemSize.

There is also a relationship that connects an entity with itself. This is called a **recursive relationship**. In the Employee entity, the manager is also an employee, so the ManagerID value must refer to an EmployeeID.

The visual data model described in this section is an easy, and powerful, technique that is used to understand the schema for a given database. Obviously, the DBMS must contain this information within the database schema so that it can organize and maintain the correct data as shown in Figure 1.3.

Data

As indicated above, the database schema describes the structure of the database. The schema is not the same as the actual data. Each one of the entities, along with its attributes, can be thought of as a database **table**. A database table is like a spreadsheet with the name of the spreadsheet being the entity name, the attributes being the columns of the spreadsheet, and the data being the rows. In database terminology, the rows called rows or records. Figure 1.4 illustrates this concept for the Employee entity. (Note: Click on the three dots on either side of the page to expand the page and make the table bigger.)

Select * from Employee

EmployeeID	FirstName	LastName	Address	City	State	ZIP	Phone	ManagerID	SSN	EmailAddress	HireDate
5	Louis	McDougald	26 Ventura Drive	Scotts Valley	CA	95066	831-438-7850	5	557-77-0715	LouisHMcDougald@gustr.com	2001-10-25
17	Audrey	McLeod	2083 Jim Rosa Lane	San Francisco	CA	94107	415-762-4047	5	626-94-4419	AudreyMMcLeod@fleckens.hu	2002-02-28
35	Kyle	Sisk	2441 Cimmaron Road	Garden Grove	CA	92643	714-636-6046	5	624-27-0067	KyleTSisk@jourrapide.com	2000-06-20
45	Zachary	Cortez	289 Brown Bear Drive	Los Angeles	CA	90017	951-603-6990	5	569-79-8855	ZacharyECortez@fleckens.hu	2006-07-10
59	Thomas	Vaughan	3758 Armbrester Drive	Irvine	CA	92614	310-526-5232	5	620-15-4370	ThomasDVaughan@superrito.com	2006-11-05
61	Jerry	Scranton	1767 Reynolds Alley	Paramount	CA	90723	562-346-7291	17	622-18-9025	JerryEScranton@superrito.com	2011-06-26
68	Helen	Pauley	4332 Park Street	Livermore	CA	94550	925-424-4804	35	610-26-2945	HelenAPauley@dayrep.com	2009-10-17
74	Joseph	Butler	2817 Clarence Court	Los Angeles	CA	90017	909-998-7106	35	618-39-2983	JosephFButler@fleckens.hu	2010-03-13
82	Jessica	Botts	1706 Red Maple Drive	Los Angeles	CA	90031	323-441-6786	35	564-78-5083	JessicaABotts@dayrep.com	2005-04-19
99	Virginia	Hart	201 Diane Street	Oxnard	CA	93032	805-385-4096	35	572-20-9302	VirginiaDHart@gustr.com	2006-08-07
102	Kristen	Browner	1490 Pike Street	San Diego	CA	92126	858-566-5833	45	611-32-7709	KristenBBrowner@einrot.com	2002-08-16
115	Jerry	Myers	4094 Doctors Drive	Santa Monica	CA	90401	310-434-2891	45	616-43-7170	JerrySMyers@teleworm.us	2008-05-23
135	Ronald	Lynch	875 Pike Street	San Diego	CA	92123	858-587-5074	45	551-69-3204	RonaldJLynch@einrot.com	2012-01-09
155	Rickey	Dietz	4405 Beech Street	Antioch	CA	94509	925-756-5184	59	562-88-2667	RickeyCDietz@armyspy.com	2010-06-24
158	Dorothy	May	504 Black Oak Hollow Road	Sunnyvale	CA	94089	408-756-3631	59	606-38-3557	DorothySMay@armyspy.com	2009-05-29
160	Sergio	Silver	4622 Davis Avenue	San Francisco	CA	94107	707-759-6359	59	618-30-7224	SergioCSilver@armyspy.com	2008-02-13

Figure 1.4: Employee data

Remember, it is important not to confuse the database schema with the actual data. The schema defines the structure and the data is the actual information about each individual Employee or Establishment or Inspection.

Earlier, we introduced the idea of a key attribute. We can see from the data how the key attribute serves as the mechanism to uniquely identify each row or each record in the table. This is important because we could have two employees who have the same name or live in the same house. The key attribute solves this problem by guaranteeing that each record will have a unique value. We are familiar with this concept. Your bank account has a unique number. Your medical records are identified with a key, which may be your Social Security number. In the example, we have used a somewhat standard notation by naming the key with "ID". At times, key fields are also identified by a “_no” or “_id” as part of the name. The DBMS enforces this requirement so that no two records in the same table have the same key value.

1.6Knowledge Check

The purpose of this chapter was to introduce you to the importance of databases and database technology in today's world. Many of our daily activities require the use of databases to retrieve information. Simple things such as sending a text message, going grocery shopping, or even visiting our doctor are supported by database technology.

We also discovered that for almost all occasions where we need some information, we depend on a database. This is especially true when we use our electronic devices such as computers, tablets, or smartphones. We frequently look through catalogs for purchases, go to libraries for information, or look up websites for opinions and news.

Even though information is provided in many different forms and formats, the underlying storing and maintaining of data has many common elements. Database Management Systems (DBMS) are highly specialized computer systems that provide all of this underlying technology. A DBMS provides all the tools and functionality required to capture, store, maintain, configure, manipulate, and present raw data in many different forms and formats. The standard method of extracting data from a database by using the DBMS is to write queries using Structure Query Language (SQL).

The chapter also reviews various general business careers and their required database knowledge and skills. As is noted, almost any knowledge-based career in today's society requires substantial database knowledge.

This assessment can be taken online.

Chapter 2: Extracting Data: The Select Statement

2.1Introduction: The Need to Extract Data

Learning Objectives

After reading this chapter, you will be able to:

- Understand basic database structure
- Do simple queries on a single table
- Do complex queries on multiple tables
- Understand the clauses of the SQL Select statement
- Understand how to join multiple tables in a query
- Be familiar with various selection conditions in a query
- Know how to order a query's result for display

Suppose we are an online retailer named Red Cat Shoes that sells men's and women's shoes. We have information about products, sales, customers, manufacturers, and other items in our database. To increase sales and profitability we want our database to answer questions like these:

- Which products sell best? Which products sell worst? When are sales of these products best?
- Which manufacturers make the most popular products and which the least popular products?
- Who are our best customers? What kinds of products do they tend to buy? Which brands do they prefer? How often do they purchase: monthly, yearly, twice a year? Do they buy at the beginning of each month, in the middle, or at the end?

These are interesting questions, and getting accurate answers to them may help us significantly in achieving our goals. Fortunately, careful use of a database query language can give us answers to these and many other questions.

As we learned in Chapter 1, the standard query language for relational database processing is named **SQL**, also known as **Structured Query Language**, or "Sequel." We will cover SQL in detail in a later chapter, but for now, our aim will be to give you a brief introduction and overview to demonstrate SQL's power in answering practical questions of interest to businesses. We will start with simple questions—questions that are easier than those above—and step-by-step we will show you how these and more sophisticated questions can be answered using a database and SQL.

2.2Simple Queries: Extracting Data from a Single Table

In this section, you will learn about two important concepts—the database table and the SQL Select Statement. We teach these concepts primarily by illustrating examples of their use.

A database **table** consists of rows and columns much like a spreadsheet. The columns are the attributes of the entities in the table, and the rows are the individual entities. A **SQL Select Statement** is the basic SQL language statement that is used to extract data from a table or a set of tables.

We illustrate examples of tables and SQL select statements by showing them within a query box. The query boxes serve two functions. First, of course, you can view the format and syntax of the SQL select statement. As with any programming language, there is a precise syntax required for SQL Select statements. Second, and more importantly, you can actually execute the SQL Select statements real time as you proceed through the textbook. Each query box, or code box, is dynamically attached to the real live database. This feature of using query boxes that dynamically connect to the database is a unique capability only used with interactive smart textbooks.

How do the query boxes work?

One of the advantages of using an interactive online textbook is that you can actually execute code right here within the textbook. In the following sections, you will find code boxes that have SQL code in the box. When you click on the RunQuery button, the code in the box is executed dynamically. It sends the query in the code box to a remote database server that executes the code and returns the result.

Can I change the statement in the query box?
Yes! You, the reader, can also dynamically modify the code and execute your individualized code. Simply left-click within the code box to initiate edit mode. You can then edit the code in any way you desire. When you click on the RunQuery button, it executes the code that is showing in the box. As you see more examples and learn how to write SQL, you should practice within the useful capability of this online textbook. Don't worry about making mistakes, the worst that will happen is that you will get an error message. To reset the query to its original setting, click the Reset Query button. Enjoy!

We have created a database called Red Cat Shoes, which we will use throughout the remainder of the text. We begin by using a simplified version of the database with only one table, called the SimplifiedSales table. This table has a lot of information about sales, customers, products and so forth. We begin with this single table to teach the basic form of SQL statements. As the book progresses, we will expand the database by modifying the SimplifiedSales table and by adding more tables to show more complex types of queries. Eventually, we will transition to a more complete and correct database with several tables. When we finish, we will have a rather sophisticated database to support our little Red Cat Shoes company.

So let's begin by looking at the SimplifiedSales table in the Red Cat Shoes database. The table below shows the columns that are in this table. We use this table to teach the basics of SQL, however, you should note that this table is not

sufficient for the final Red Cat Shoes database. We will present the correct database tables as we progress through the text.

SimplifiedSales

SaleDate	FirstName	LastName	ProductName	Category	Co
-	-	-	-	-	-

If you click the "Run Query" button below, you will see a list of sales transaction records from this online retail shoe store. Don't worry about the syntax of the query, we'll discuss it in detail shortly. For queries executed from inside this book, results are normally set at 100 rows, but you can adjust the number of rows returned by using the Advanced Options settings.

```
SELECT * FROM SimplifiedSales
```

What is the Advanced Editor feature used for?
The query boxes also have an "Advanced Editor" feature. When you click on that link, a new tab will open in your browser, which allows the use of multiple database management systems and multiple login IDs or database credentials. The Redcat1 credential logs you onto a teaching version of the RedCat database. Redcat1 has all types of intermediate and non-standard tables that are used for instruction. The Redcat2 credential logs you onto the full version of the RedCat Shoes database. For both the Redcat1 and the Redcat2 databases, you only have permission to view the data, not to change it. The Student ID credential logs you onto your own database. At the beginning of the course, it is empty. Since it is your own database, you have permission to do anything you want—you can view or create tables, modify the data, and even destroy it. You will learn more about how to use the Advanced Editor in later sections.

Let's review this list. Each row gives the record for the sale of a single item. If you look at the column headings you will see the meaning of each field in each

row. For example, the first column gives the date of the sale, and the next two columns the name of the customer. In the fourth column you see the name of the product being purchased followed by its category, color, size, manufacturer, and price. We invite you to scroll through the data to see the different values in each column.

Note that ProductName is very specific. It exactly identifies a product, while Category is more general. Thus, Category might read "boots" whereas ProductName reads "natalia high platform suede fringe boot." Each category has many different products so the Category column is used to group similar products.

Of course, this is a simplified table for use in this example. Normally many other pieces of information are included such as the quantity of items purchased and the total price. Also, there will often be multiple products purchased on a single sale. For simplicity in our explanations in this section, we have omitted some of this data. We will show how to expand the database for these items later.

To show how to get information from this table, let's start with four simple queries.

Query 1: List the sales of all green footwear.

To see all the sales of various items of footwear in green we could simply scroll through the table and visually identify those rows where Color is green. But when the quantity of data is very large, this is tedious and impractical. For example, suppose we had 10,000 sales records. Scrolling through the table would take a very long time, and we would probably miss some of the relevant sales since we are depending only on our eyes to spot them. Using the power of the computer and its capability of quickly sifting through data, we can instead create a simple query by stating the condition Color = 'Green'.

```
SELECT *
FROM SimplifiedSales
WHERE Color = 'Green'
```

Note that the number of rows in the result is smaller than the whole table and that all of them show sales where the color is green, as desired.

Now let's take it a step further.

Query 2: List sales of green sneakers.

To narrow down your search even more, add the selection condition Category = 'sneakers'.

```
SELECT *
FROM SimplifiedSales
WHERE Color = 'Green'
 AND Category = 'sneakers'
```

Query 3: List sales of green sneakers made by Adidas.

In this case, we would add the condition "ManufacturerName = 'Adidas'.

```
SELECT *
FROM SimplifiedSales
WHERE Color = 'Green'
 AND Category = 'sneakers'
 AND ManufacturerName = 'Adidas'
```

Suppose we shift gears now and decide that we want to instead list boots sold in April.

Query 4: List sales of all types of boots in April.

This time we don't care about the manufacturer or the color; we are only interested in the month of sale.

We state the query as follows:

```
SELECT *
FROM SimplifiedSales
WHERE Category = 'boots'
 AND MONTH(SaleDate) = 4
```

2.3Queries in SQL

Clauses of an SQL Query

These four queries are examples of the SQL Select statement. The Select statement consists of multiple **clauses**, each of which begins with an SQL reserved word which, depending on the clause, is followed by a sequence of column names, computations, literals, table names, or conditions. The four examples above each contains a **SELECT** clause, a **FROM** clause, and a **WHERE** clause. A Select statement can have as many as six clauses, with the other three being the **GROUP BY** clause, the **HAVING** clause, and the **ORDER BY** clause, all of which will be covered in this book. We will give some examples of the ORDER BY clause in this chapter, and we will cover the GROUP BY and HAVING clauses in Chapter 4.

Let's look more carefully at these queries to identify some basic characteristics. We will start by looking again at our first query:

```
SELECT *
FROM SimplifiedSales
```

This query—without a WHERE clause—gives us the entire SimplifiedSales table. It demonstrates that only two clauses are required in the Select statement—the SELECT clause and the FROM clause. Each of the other four clauses is optional. So now we can address some questions:

What is the meaning of the "*" in the Select clause?
The asterisk in the SELECT clause means to select all of the columns in the table (or tables), rather than just some of the columns.

When do I need to use a semicolon to end an SQL statement?
The standard termination punctuation for an SQL query statement is a semicolon. However, the query box automatically adds a semicolon after the statement before sending it to the database server if necessary. It is never wrong to add your own semicolon after your statement, and, in fact, is a good practice. As you become proficient with using the query boxes, you will discover that you can write multiple SQL statements in the same box. **In that case, you must always use the semicolon** as the ending punctuation for each separate SQL statement.

For example, suppose we formulated this query as:

```
SELECT Category
FROM SimplifiedSales
```

Or, we could list multiple columns:

```
SELECT Category,
 Color,
 ManufacturerName
FROM SimplifiedSales
```

Therefore, if we want *all* the columns to be listed, "SELECT *" is a shorthand way of saying it, rather than listing all of them. This gives us the obvious answer to several other questions.

What is the purpose of the SELECT clause?
The SELECT clause lists the columns that should show in the result of the query. They can be in whatever order you want.

What is the purpose of the FROM clause?

The FROM clause lists the table or tables in which the data resides. So far, all our queries access only one table to get their result, but later in the chapter we will see how we can access multiple tables to answer a query.

What is the purpose of the WHERE clause?
The WHERE clause contains the **selection conditions** the query uses to decide which data to include in the query result. Specifically, the query looks at the table in the FROM clause and then applies the selection condition from the WHERE clause to each row in the table. If a row satisfies the condition in the WHERE clause, then the row is selected for the result. If not, the row is simply omitted from the result.

In what order are the clauses of the SELECT statement executed?
Glad you asked! This is an important question, because if you understand the order of execution, you will have a better understanding of what result to expect. This will be particularly true as we get into more complex queries later in the book.

To simplify our discussion, suppose we're looking at this query:

```
SELECT Category,
 Color,
 ManufacturerName
FROM SimplifiedSales
WHERE Category = 'boots'
 AND Color = 'Black'
```

The FROM clause is always the first clause executed. In this case, its execution is very simple—it simply identifies the SimplifiedSales table. Next, the WHERE clause is executed. In this case, it means that each row in the SimplifiedSales table is examined, and if that row's Category field is boots and its Color field is black, then the row is placed in the result. Otherwise, the row is omitted from the result. (Note that the SimplifiedSales table itself is unchanged. The query simply builds a new, temporary table in which it stores its result.) After the FROM and

WHERE clauses, the SELECT clause is executed. This means in our example that the data from the Category, Color, and ManufacturerName columns of the selected rows is retained, while the data from all other columns is discarded from the query result.

How are literal values represented in the WHERE clause?
The queries shown above illustrate that string literal values, such as boots, must be shown with single quotes (') surrounding them, while numeric literals, such as 4, are shown without quotes.

Numeric fields, which use numeric literals, are normally fields that allow arithmetic. By convention, the table key fields, such as Customer_ID, are also numeric.

Character fields, which use string literals, are fields that include alphabetic characters, and some number-like fields that may include characters, such as postalcode and phone.

How do I handle date fields?
In SQL Server there are several date data types, with "date" and "datetime" the two most common. In order to access specific date information, you must first extract the appropriate portion of the date using a date function. For example, "month(aDateField)" will extract the month and return it as an integer. "Year(aDateField)" extracts the year as an integer, and "Day(aDateField)" extracts the day of the month as an integer. More detail about date functions is provided in Chapter 6.

Removing Duplicate Records

An obvious question raised by the queries we've seen so far is how to eliminate duplicate rows from the result. For example, if there are lots of sales of black

boots from a given manufacturer, then the set of 'boots', 'black', and the manufacturer shows up once for each such sale. If we want to eliminate these duplicates, we write the query as:

```
SELECT
DISTINCT Category,
 Color,
 ManufacturerName
FROM SimplifiedSales
WHERE Category = 'boots'
 AND Color = 'Black'
```

Note that the **DISTINCT** key word does not eliminate duplicate values from individual columns but rather duplicate rows from the query result.

Suppose we want to know who all the manufacturers of black boots are, and we don't want their names duplicated. Then:

```
SELECT
DISTINCT ManufacturerName
FROM SimplifiedSales
WHERE Category = 'boots'
 AND Color = 'Black'
```

How do I know if I need to use the keyword "distinct"?
Unless specifically asking for duplicate records, you should always remove duplicate records in your queries. In other words, do not allow duplicates unless the question specifically asks for it.

So the question remains, when do I need to use the "distinct" keyword? This can sometimes be a little difficult. Let's discuss two cases: a single table query and multiple table query.

Single table query: If the question asks for all the information (such as *) or for the primary information from a table (such as the ID, or possibly another identifier field), then normally the keyword "distinct" is NOT necessary. If, however, the question asks for a field whose values could be shared by multiple

records, then use "distinct" to remove duplicates.

For example, a query including ProductID or even ProductName from the Product table, would not need the keyword "distinct". However, a query asking for the ListPrices for Products would definitely need the keyword "distinct" because we should always expect there to be multiple products with the same ListPrice.

Multiple table query: Joining tables together almost always will produce records that have multiple copies of records from one of the tables. If the question asks for fields from the table with multiple copies, then you should include "distinct." You should always be asking, "Can there be multiple copies of the data to be returned?"

For example, a query asking for the names of customers who made purchases in January 2014 will require the joining of the Customer table and the Sale Table. However, there will probably be multiple copies of customer data because the same customer can have multiple purchases in January 2014. In that case, you will need to use the "distinct" keyword. Another example: If the question asks for categories of shoes from sales records, we would expect the same category of shoes to occur many times, and we will again need "distinct."

Finally, remember you can always test your queries with and without "distinct." If you get different answers, you should use "distinct." And inserting "distinct," even when not necessary, will usually yield the correct answer.

Performing Calculations

Suppose we want to do some calculations in the Select clause. For example, we want to calculate sales tax, shipping cost, and the price for each item to get a total sale price. Let's assume we have a 6% sales tax and an 8% shipping charge.

Then we can use this query:

```
SELECT SaleDate, FirstName,
        LastName,
        ProductName,
        Price,
        'Sales tax = ', .06*Price,
        'Shipping = ', .08*Price,
        'Total =', price + .06*price + .08*price
        FROM SimplifiedSales
```

Note how the Select clause contains string literals — ' Sales tax = ', ' Shipping = ', and 'Total =' —as well as the computations. This illustrates how we add explanatory information as well as computations to the Select clause. In this example, we didn't create specific column headings for the computed fields. We'll see how to do that below.

Notice that in the calculations we have used the asterisk as the multiply operator. This is a different use than we have seen before to return all columns of a table. In this case we cannot use the asterisk (*) to indicate all columns in SimplifiedSales. If we include literals or computations, then we have to list the other columns we want.

How do I use commas in the Select clause?
Commas are used to separate each column for the result. When columns are listed as part of the Select clause, each column name must be separated by a comma. Each literal, as shown above, is also not only enclosed in single quote marks but is also separated by commas. A calculation phrase must also be separated by commas, although it does not need single quotes because it is comprised of numbers and column names.

Renaming Columns with Alias Names

Instead of duplicating these literals in every row, we can use the (optional) AS

keyword to give a name to the computed column. If we want to use two words (with a blank space) as the column heading, we must put the two-word titles in double-quotes ("). This revision of the query shows how to do all this:

When we identify particular columns in the SELECT clause, those exact column names appear in the result table as the column names. However, sometimes we may want to rename the column to be more explicit or descriptive. This is also referred to as an alias name. Two examples come to mind.

The first example occurs when the database may use a column name that is not as descriptive as is required for the answer table. Attribute names—aka column names—in the database are always single words without spaces. In other words, no blank spaces are allowed in database table names or attribute names. But for the answer table, we may want the column heading to be more descriptive. For the displayed result, we may want to change an attribute name such as "FirstName" to be "First Name." In the RedCat database, the attribute names are quite descriptive. However, sometimes in other databases, the column names are very abbreviated or are quite cryptic and might be difficult for the user to understand. Examples could be "pdate" and "xdate" to mean "purchase date" and "expiry date." In those cases, it would be beneficial to rename the output column headings to display a more descriptive heading.

Another frequent example of the need to rename a column in the answer table is when an arithmetic operation occurs. For example, in the SimplifiedSales table, we might want to do some calculations for tax and shipping. The resultant columns might be named "Sales Tax", "Shipping" and "Total Amount".

The technique to rename columns in the result table is with the AS keyword. First, identify the column name or the arithmetic statement followed by the AS keyword and then provide the result table column name. If you included spaces in the name, you must enclose it within quotes. The following example shows

the syntax. Notice that each column name or calculation and its accompanying alias name is separated by a comma.

```
SELECT SaleDate,
        FirstName,
        LastName,
        ProductName,
        Price, .06*Price AS &amp;quot;Sales Tax&amp;quot;,
        .08*Price AS Shipping,
        Price + .06*Price + .08*Price AS &amp;quot;Total Amount&amp;quot;
FROM SimplifiedSales
```

Sorting the Result Records

Can we see query results in some order, for example alphabetical order? To do that, we add the Order By clause, which is always executed *after* the Select clause.

Here's a sample query:

```
SELECT
DISTINCT Category,
 Color,
 ManufacturerName
FROM SimplifiedSales
WHERE Category = 'sneakers'
ORDER BY ManufacturerName
```

As you can see, the query result is listed in alphabetical order by manufacturer. But after that, no order is maintained. That is, the colors need not be in order.

Suppose we want the order of the result to be alphabetical by color and after that in reverse alphabetical order by manufacturer.

This query will give us the right result:

```
SELECT
DISTINCT Category,
 Color,
```

```
 ManufacturerName
FROM SimplifiedSales
WHERE Category = 'sneakers'
ORDER BY Color,
 ManufacturerName DESC
```

In this query, the ManufacturerName column is a secondary sort key, and the **Desc** keyword causes that column to be sorted in reverse alphabetical order.

2.4Where Clause Operators

Combinations of AND, OR, and NOT

Can we do anything more with the WHERE clause?
We can do many things with the WHERE clause, but for now let's just demonstrate a few obvious choices. Suppose for example we want to see all the sales of boots as well as all the sales of Puma products. We would use the "OR" keyword as a connector in our WHERE clause.

Here is an example:

```
SELECT *
FROM SimplifiedSales
WHERE Category = 'sneakers'
 OR ManufacturerName = 'Puma'
```

Note that *either* the Category must be sneakers*or* the ManufacturerName must be Puma. It is not necessary that both of these be true, though they can be.

And if we want to see sales of sneakers as well as sales from manufacturers other than Puma:

```
SELECT *
FROM SimplifiedSales
WHERE Category = 'sneakers'
 OR NOT ManufacturerName = 'Puma'
```

Note in this case the combination of the "OR" keyword and the "NOT" keyword to give us the desired result, which is either sneakers or manufacturers other than Puma. If we wanted to only see sneakers from all manufacturers other than Puma, we would have said Category = 'sneakers' AND NOT ManufacturerName = 'Puma'. Try it.

How do you combine "not," "and," and "or" in a single Where clause? Combining conditions using the "not," "and," and "or," keywords follows the normal rules of precedence: "not" takes first precedence, followed by "and," followed by "or." If we need to override this precedence, then we use parentheses. In fact, to ensure that the correct result occurs, we recommend that you always use parentheses when combining conditions with these keywords.

Here are some examples:

Query: Find all sales of Nike products and all sales of black sneakers.

```
SELECT *
FROM SimplifiedSales
WHERE ManufacturerName = 'Nike'
 OR Category = 'sneakers'
 AND Color = 'Black'
```

Or, using parentheses:

```
SELECT *
FROM SimplifiedSales
WHERE ManufacturerName = 'Nike'
 OR (Category = 'sneakers'
 AND Color = 'Black')
```

Query: Find sales of Nike sneakers and boots.

```
SELECT *
FROM SimplifiedSales
WHERE (Category = 'sneakers'
 OR Category = 'boots')
 AND ManufacturerName = 'Nike'
```

In this case the parentheses are required, because the "OR" must take precedence over the "AND". You can see this by comparing the result of this query with the result of the following query.

```
SELECT *
FROM SimplifiedSales
WHERE Category = 'sneakers'
 OR Category = 'boots'
 AND ManufacturerName = 'Nike'
```

which is the same as

```
SELECT *
FROM SimplifiedSales
WHERE Category = 'sneakers'
 OR (Category = 'boots'
 AND ManufacturerName = 'Nike')
```

Query: Show sales of sneakers and boots not made by either Nike or Puma.

```
SELECT *
FROM SimplifiedSales
WHERE (Category = 'sneakers'
 OR Category = 'boots')
 AND NOT (ManufacturerName = 'Nike'
 OR ManufacturerName = 'Puma')
```

This example illustrates the use of all three operators as well as the necessary parentheses. If the parentheses were omitted, we would get the following:

```
SELECT *
FROM SimplifiedSales
WHERE Category = 'sneakers'
 OR Category = 'boots'
 AND NOT ManufacturerName = 'Nike'
 OR ManufacturerName = 'Puma'
```

As you can see, the results are very different from the query with the parentheses included.

There are many possible combinations of the three operators, and of course we

can't illustrate them all. But the appropriate use of the operators depends entirely on the query you want to answer. You combine them and use parentheses as needed according to the logical requirements of the question you need to answer.

Other WHERE Clause Options

A row is selected by the query if it satisfies the condition in the WHERE clause. As we showed in Chapter 2, this condition can be complex, consisting of multiple basic parts connected by AND, OR, and NOT operators. The basic parts determine whether a given comparison is true, and they can take any of the following forms:

1. Comparison of a value in a column or computation to some other value, using the six standard comparison operators (=, <>, <, <=, >, >=)
2. Determination of whether a value falls in a range, using the BETWEEN operator
3. Determination of whether a value is in a set of values, using the IN operator
4. Determination of whether a string value matches a pattern, using the LIKE operator and wild card characters
5. Determination of whether a given value is null

Using Comparison Operators

Query: Which sales have a price of over $20?

```
SELECT *
FROM SimplifiedSales
WHERE Price &gt;= 20
```

This query illustrates the use of one of the six comparison operators (=, <> [not

equals], <, >, <=, >=). These may be used to compare columns, computed values, and literals with other columns, computed values, and literals.

Additionally, the boolean connectives AND, OR, and NOT may be used to create compound conditions or to negate a condition. Parentheses may also be used to group conditions. Here is a more complex example:

Query: Show sneakers and boots not costing over $50.

```
SELECT *
FROM SimplifiedSales
WHERE (Category = 'sneakers'
 OR Category = 'boots')
 AND NOT Price &gt; 50
```

Here is a query that shows the use of computed values:

Query: List sales where the total tax and shipping cost is over $20.

```
SELECT *
FROM SimplifiedSales
WHERE .06*Price + .08*Price &gt; 20
```

Comparison operators can also be used with string values. For example:

Query: List sales in a category whose first letter is "f" or later in the alphabet.

```
SELECT *
FROM SimplifiedSales
WHERE Category &gt;= 'f'
```

Using BETWEEN

Consider this query:

Query: List sales with a price between $20 and $50, inclusive.

It can easily be written using two comparison operators, as follows:

```
SELECT *
FROM SimplifiedSales
WHERE Price &gt;= 20
 AND Price &lt;= 50
```

However, having to write the column name, Price, twice is annoying. Thus, we're tempted to write the Where clause as:

- Where Price >= 20 AND <= 50

or

- Where 20 <= Price <= 50

both of which are syntactically **incorrect**.

However, SQL provides the operator BETWEEN that fits this occasion. Using BETWEEN, the query is written as:

```
SELECT *
FROM SimplifiedSales
WHERE Price between 20
 AND 50
```

BETWEEN can be used in a comparison of a value with two other values (the first smaller than the second) if the value being compared is equal to either the smaller or the larger value or is equal to any value in between. In other words, BETWEEN includes the endpoints.

Using the IN Operator to Determine Set Membership

Query: List sales of sneakers, sandals, heels, and boots.

Again, we can already write this query using the '=' comparison operator, but it's

somewhat tedious:

- Select *
- From SimplifiedSales
- Where Category = 'sneakers' OR
  - Category = 'sandals' OR
  - Category = 'heels' OR
  - Category = 'boots'

Thus, we might be inclined to write the WHERE clause as

- Where Category = 'sneakers' OR 'sandals' OR 'heels' OR 'boots'

But you've probably guessed this is syntactically **invalid**. However, SQL provides a legal alternative that's even better, using the IN operator:

```
SELECT *
FROM SimplifiedSales
WHERE Category IN ('sneakers',
         'sandals',
         'heels',
         'boots')
```

This query introduces and illustrates the IN comparison operator. The Where clause evaluates to 'true' if the Category for the row is found *in the set* contained in parentheses —that is, if the Category is sneakers, sandals, heels, or boots. We will have more occasion to use the IN operator with subqueries. The advantage here is simply that it allows us to write the query in a way that is close to the way we think of saying it, whereas using the comparison operator multiple times, and connecting the comparisons with 'OR' operators, is awkward.

(Note that the English language query says "sneakers, sandals, heels, *and* boots" while the logical query is "sneakers, sandals, heels, *or* boots." It's interesting to

ponder why this is so. It shows the difference between normal English usage, and strict *logical* usage.)

Using the LIKE Operator in Pattern Matching

Suppose we're looking for a particular product but can't remember the exact spelling of its name. For example, we know that the word "high top" appears in the name, but we don't know the exact spelling—is it "hi top," "hightop," or "high top"? **Wild card characters**, special symbols that stand for unspecified strings of characters, in combination with the LIKE operator, make it easier to use inexact spelling in a query.

Query: Find the sales of products with some form of "hi top" in their name.

```
SELECT *
FROM SimplifiedSales
WHERE ProductName like '%hi%top%'
```

SQL has two wildcard characters, % (percent) and _ (underscore). The percent stands for zero or more unspecified characters. The underscore stands for exactly one unspecified character. The LIKE operator is used to compare character variables and literals when wildcard characters are used. Other examples:

- LastName LIKE '%b_rg'
- FirstName LIKE 'J_dy'

The first example evaluates to true if a customer's last name ends with (for example) "berg," "barg", "birg", "borg", or "burg". In the second example the condition would identify all customers whose first name is "Judy" or "Jody", or any other combination of 4 letters starting with "J" and ending with "dy".

Checking for Null Values

Suppose we want to identify records for which basic information is missing – that is, certain fields are null. SQL helps us with that.

Query: List sales for which the color was not been entered.

```
SELECT *
FROM SimplifiedSales
WHERE Color is NULL
```

This is a simple test, but we're tempted to write the query as:

- Select *
- From SimplifiedSales
- Where ItemSize = NULL

But this structure is not permitted. If we want to check for nulls, we have to write "IS NULL". (We can use lower-case if we wish.)

What does Null value really mean?
A NULL value has a special meaning in database technology. A value of NULL indicates that the value is unknown or not applicable. A NULL value can apply to any type of data, including strings, integers and dates. NULL values are not equal, so fields that have NULL values do not compare as equals.

It should be noted that a NULL value is not the same as a zero length string or a field containing blanks or spaces. A difficulty arises because on an output result a NULL field and a blank field display the same way. As indicated earlier, to test if a field is NULL, you must use the IS NULL test. Testing for a field equal to " " does not give the same result.

2.5Knowledge Check

The purpose of this chapter was to introduce you to the SQL Select statement

and give you experience in learning its basic functions so you could formulate and execute basic queries. A Select statement can have as many as six clauses, the **Select** clause, the **From** clause, the **Where** clause, the **Group By** clause, the **Having** clause, and the **Order By** clause, all of which will be covered in this book.

Since this was an introductory chapter, we thoroughly explored two of the most important clauses in the Select statement—the Select clause and the Where clause. We showed how data can be displayed as whole records, selected and arranged columns, and with literals and computations using the Select clause. The Where clause is particularly valuable because it allows us to choose those records that meet specified criteria, such as having certain values in columns. We also illustrated the Order By clause which makes the sorting of query results possible.

Throughout the chapter, we used a single table for all query examples—SimplifiedSales. This table contains all the data needed to demonstrate the different types of queries that can be performed on a single table using SQL. As we move to the next chapter you will see how SQL can be used to query multiple tables to get more complete information from a database.

This assessment can be taken online.

Chapter 3: Data across Tables: The Join Statement

3.1Introduction: Data Combinations

Learning Objectives

After reading this chapter, you will be able to:

- Understand how the Join operation works
- Do more complex queries on multiple tables
- Join two or more tables with a single Select statement
- Understand why you would want to join a table to itself, and use aliases to do so

The examples given in the previous chapter have probably given you insight into the possibilities for obtaining valuable information from a database. However, the queries we've looked at so far are fairly simple and straightforward. Suppose we want more sophisticated information that would answer questions like those below.

- Which categories are being purchased by customers living in New England?
- What are the zip codes of customers who buy products costing over $50?
- Which customers purchase from manufacturers located in California?

As you look at each of these questions you have undoubtedly noticed that we don't have enough information in the SimplifiedSales table to answer them. That's because the table used in Chapter 2 was "simplified" so that your introduction to SQL would be easier. However, if we expanded SimplifiedSales with data to answer these and similar questions, you can readily see that the table itself would soon become unwieldy. Therefore, the information needed to answer any of these questions is, of necessity, located in multiple tables. We need a way to combine the data of multiple tables while still keeping these tables distinct from each other. We will show you how to do that in this chapter.

3.2Extracting Data from Multiple Tables

The Basic Join

Remember in Chapter 1 we defined entities and attributes. We also discussed that each entity becomes a table in a database system, and the attributes become columns in the table. When a table is returned from a query, the rows of the table are the individual data items. In Chapter 2 we used only one table, the SimplifiedSales table. In this chapter and all the subsequent chapters, we will use a more complete database structure, which has several tables representing the several entities of the Red Cat Shoes database.

The following figure is a visual model of the more complete Red Cat Shoes database. Each box represents a table, and the shelves in the boxes are the columns of the table. The connecting lines between the boxes are the relationships between tables. It is often easier to understand the database schema by using a visual model, as show below.

You may wonder how these tables were chosen and defined. That is a topic for a database management and design course. Suffice it to say that the following set of tables is a **normalized** database schema. Normalized simply means that the schema is structered in the simplest and most efficient form.

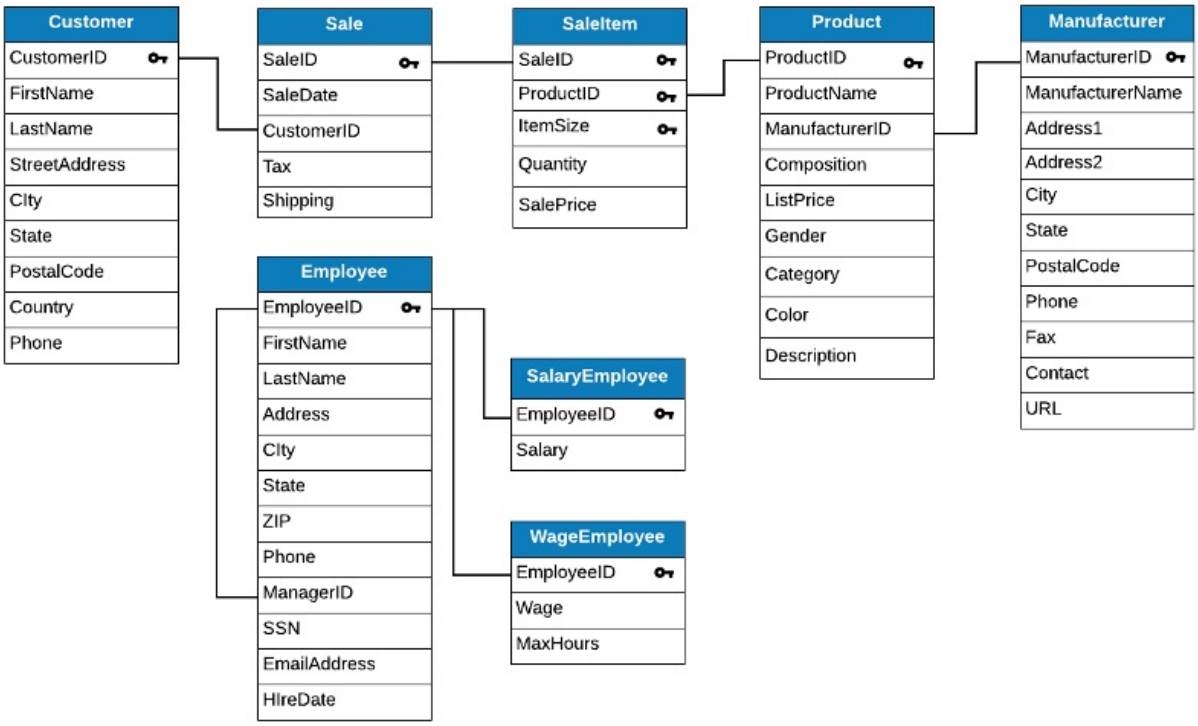

Figure 3.1: Visual model of the Red Cat database tables

Let's review a few important concepts. Notice that the first column in seven of the tables is an identifying column. This column's value identifies the row since this value is unique in its table. Because of this unique property, the column is called a key. This is indicated by the presence of a key symbol next to the column name.

In the SaleItem table, the key consists of three columns: SaleID, ProductID and ItemSize. The SaleID and ProductID connect the SaleItem with the Sale and the Product. The ItemSize key field allows for the sale of the same product in different sizes on the same Sale. The combination of SaleID, ProductID and ItemSize uniquely identifies a sale item in the SaleItem table.

Now let's talk about the connecting lines. Look at the line that connects the Sale table with the Customer table. Notice that the column name at each end of this line is the same—CustomerID. This means that if we look at a sale record in the

Sale table, we can find the information about the customer who made that sale in the Customer table. For example, if we see a sale record with CustomerID 17583, then we can go to the Customer table, find the record whose key is 17583, and that will give us full information about the customer to whom the sale was made.Thus, the column CustomerID in the Sale Table is referred to as a **foreign key**—a set of attributes in one table that is a key in another table. In other words, foreign keys link data in one table to data in another table. This means that CustomerID in the *Sale table* has a key value in *Customer*.

The other connection lines in the diagram have similar functions, and we will explain them through queries as we go along.

We start with a query that connects two tables:

Query: What are the products manufactured by Fila?

The data needed to solve this query is found in two tables: Product and Manufacturer. The SQL solution requires listing *both* of these tables in the FROM clause, together with a particular type of FROM clause condition:

- SELECT *
- FROM Product P Join Manufacturer M On P. ManufacturerID = M. ManufacturerID
- WHERE ManufacturerName = 'Fila'

This query introduces several new features we haven't seen in the FROM clause before. First, it lists more than one table, in this case Product and Manufacturer, and it uses aliases—P for Product and M for Manufacturer—to simplify the query. Aliases are alternate names for tables and are used to simplify the writing of a query. An alias is defined by writing it after the name of the table in the FROM clause. The table name and the alias name are separated by one or

more blank spaces. We will use them throughout the chapter. Second, it combines the tables — Product and Manufacturer — by "joining" them. It does this through the key words JOIN and ON, and by providing a **join** condition.

How do I give a table an alias name?
In Chapter 1 you learned about giving an alias name to a column. Tables listed in the From clause also can be given an Alias name. In both instances, we can use the keyword AS to identify the alias name. However, the AS keyword is optional in both situations. It is a normal convention to use the AS keyword for column aliases but do not use it for table aliases.

We use table alias names to provide a shorthand notation to refer to the table, which requires less typing throughout the SQL statement. It is a normal convention to use one or two-letter alias names for tables.

To understand what is happening, we need to break the query into two parts. First, we look at a simpler query:

- SELECT *
- FROM Product P JOIN Manufacturer M ON P. ManufacturerID = M. ManufacturerID

The process the system follows for this query is best understood if we simplify it even further. Suppose then that Product and Manufacturer have only two columns each, as follows:

Product

ProductID	ManufacturerID
--	--

Manufacturer

ManufacturerID	ManufacturerName
--	--

We want to join a row from Product with a row from Manufacturer if the ManufacturerID of the Product row is equal to the ManufacturerID of the Manufacturer row. This means we have to compare *every* row in Product with *every* row in Manufacturer to see if the ManufacturerIDs in the two rows are equal. To illustrate, suppose each table has three rows:

Product

ProductID	ManufacturerID
1	10
2	20
3	10

Manufacturer

ManufacturerID	ManufacturerName
10	Fila
20	yyy
30	zzz

Then the result of the query

- SELECT *
- FROM Product P JOIN Manufacturer M ON P. ManufacturerID = M. ManufacturerID

is

Product-Manufacturer

ProductID	P.ManufacturerID	M.ManufacturerID	Manufacturer
1	10	10	Fila
2	20	20	yyy
3	10	10	Fila

What is dot notation?

Column names are associated with a particular table. In situations where the same column name is used in multiple tables, we use **dot notation** to distinguish which table the column belongs to. The syntax is table name (or alias) dot column name— P.ManufacturerID

Side note: In these examples, we use the same column name for the key field and the foreign key field. However, the column names do NOT have to be the same.

There are several points to notice here. First, note that the result of the query has 4 columns, consisting of the two from Product and the two from Manufacturer. Since ManufacturerID appears in both tables, it appears twice in the result. We distinguish these two by prefixing the alias name to the column name just as we did in the FROM clause.

Second, note that in each row the value of ManufacturerID is the same in both copies of the column. This is because the JOIN ON clause said "Join the two rows if the ManufacturerID values are *equal*."

Third, notice also that the row from Manufacturer for ManufacturerID 10 appears twice in the result. This is because there were two Products for this Manufacturer.

Finally, note that the Manufacturer row with ManufacturerID equal to 30 does not appear in the result. This is because there is no row in Product having a ManufacturerID of 30.

This query's FROM clause lists two tables: Product and Manufacturer. It is therefore a multi-table query. In processing the query, SQL looks at all possible combinations of a row from Product with a row from Manufacturer. In the ON clause the expression

P.ManufacturerID = M.ManufacturerID

means that in each row-pair combination from Product and Manufacturer, if the ManufacturerID in the Product row has the same value as the ManufacturerID in the Manufacturer row, then the row-pair is saved for the result. Otherwise, the row-pair is discarded. Our query result shows only those row-pairs that have been saved.

This process is called **joining** the two tables, and the ON clause condition

"P.ManufacturerID = M.ManufacturerID"

is the **join condition**. If we join more than two tables, we need multiple join conditions. We will see examples of such joins in the queries below.

Now let's look at the join query using the full Product table and one column from the Manufacturer table:.

Redcat Database Diagram

```
SELECT P.*, M.ManufacturerID
FROM Product P
 JOIN Manufacturer M
 ON P.ManufacturerID = M.ManufacturerID
```

Notice again that this query result has two ManufacturerID columns—one from Product and one from Manufacturer. You can see that although there are more columns than in our stripped-down example, the critical columns are still just the ManufacturerID columns in the two tables. These are the columns on which the

join depends. Moreover, even though there are many more rows in this result (because our database is much larger than 3 rows), each row still simply came about because the ManufacturerID in a Product row was equal to the ManufacturerID in a Manufacturer row.

To continue our present example, recall that the original query had a condition in the WHERE clause:

ManufacturerName = 'Fila'

The system completes the query by applying this condition to all the rows in the joined table. After all row-pair combinations are processed in this way, the SELECT clause is executed. Because these two tables have so many columns, only selected ones are displayed in the result.

Redcat Database Diagram

```
SELECT ProductID, ProductName, ListPrice, Gender, Category, ManufacturerNar
FROM Product P
 JOIN Manufacturer M
 ON P.ManufacturerID = M.ManufacturerID
WHERE ManufacturerName = 'Fila'
```

You may be wondering why sometimes we prefix the column name with a table or alias name and sometimes we don't. It is always permissible to prefix the column name with a table identifier, but it is not necessary if the column name exists in only one of the tables listed in the FROM clause. Thus, since ManufacturerName appears only in the Manufacturer table and not in the Product table, we simply write

ManufacturerName = 'Fila'

in the query above, rather than

M.ManufacturerName = 'Fila'

as we could have done.

Now suppose the foreign key column ManufacturerID in the Product table had a different name. Suppose, for example, it had been named "Mfr" instead. That is, suppose these two tables have this structure:

Product

ProductID	ProductName	**Mfr**	Composition	ListPrice	Gender	Category	Co

Manufacturer

ManufacturerID	ManufacturerName	Address1	Address2	City	State	PostalC

Then the join condition in the query above could be written as

Mfr = ManufacturerID

and the query

SELECT *

FROM Product JOIN Manufacturer ON Mfr = ManufacturerID

You see that under this redefinition of the Product table, since Mfr appears only in the Product table and ManufacturerID appears only in the Manufacturer table, no table or alias name is needed as a prefix to either column name. This is legal, simpler, unambiguous, and still accomplishes the join as before.

3.3Expanding the Join

Joining More than 2 Tables

As we saw in Chapter 2, more than two tables can be joined at once in SQL.

With our new database we will illustrate this with a query from Chapter 2. The new, more realistic database structure we're using in this chapter, however, requires some changes in Chapter 2 query solutions. Specifically, the new database allows for multiple products on a single sale by placing the product identifier in the new SaleItem table, which has a many-one relationship with the Sale table. That is, for each sale there are many sale items and each sale item contains a single product and is associated with only one sale. Therefore, we have to change our SQL queries to conform to this structure.

Query: List sales of black sneakers.

Solution 1: With join conditions in the FROM clause

Redcat Database Diagram

```
SELECT S.SaleDate,
 P.ProductName,
 P.Category,
 P.Color,
 SI.Quantity,
 SI.SalePrice
FROM Sale S
 JOIN SaleItem SI
 ON S.SaleID = SI.SaleID
 JOIN Product P
 ON SI.ProductID = P.ProductID
WHERE Category = 'sneakers'
 AND Color = 'Black'
```

In this case, the join conditions are in the FROM clause and they follow each other sequentially. First we join Sale and SaleItem with the join condition attached to the first join, then we join Product to the result of the first join with the new join condition attached at the end. If we had more tables we would follow with more joins.

Solution 2: With join conditions in the WHERE clause

The version of joining tables with the "Join... On" first became a standard in 1992. In a prior version of SQL (1987 Standard), the join condition was placed within the WHERE clause. As a database-knowledgeable technician, you will probably find the earlier version of SQL in existing systems that you work on. In this section, we illustrate the 1987 version of joining tables. The syntax is slightly different, but the concepts are the same. The syntax is as follows:

- The tables to be joined are simply listed in the FROM clause separated by commas.
- The joining conditions (i.e. the ON clause) are listed in the WHERE clause.

Other than that, the joins work exactly the same. In the earlier standard, SQL would identify the tables to join by identifying the equality of columns in different tables. The following example illustrates this earlier syntax.

Redcat Database Diagram

```
SELECT S.SaleDate,
 P.ProductName,
 P.Category,
 P.Color,
 SI.Quantity,
 SI.SalePrice
FROM Sale S, SaleItem SI, ProductBrief P
WHERE S.SaleID = SI.SaleID
 AND SI.ProductID = P.ProductID
 AND Category = 'sneakers'
 AND Color = 'Black'
```

With respect to joining multiple tables in a query, here is the basic rule: "If you have more tables, simply list them and add more join conditions."

Joining a Table to Itself

We now show how a relation can be joined to itself in SQL. In this case, **aliases** for the table name are required.

Query: List employees with the names of their managers.

SELECT E.LastName, M.LastName

FROM Employee E JOIN Employee M ON E.ManagerID = M.EmployeeID

The FROM clause in this example defines two "copies" of the Employee relation, given the alias names E and M. The E and M copies of Employee are joined by setting the EmployeeID in M equal to the ManagerID in E. Thus, each row in E has attached to it the row from M containing the information about the E row's manager. By selecting the Employee names from each row, we obtain the required list of employees paired with their managers. To make the list easier to understand, we've added first names and column headings distinguishing employees from their managers.

Redcat Database Diagram

```
SELECT E.FirstName AS EmpFirstName,
 E.LastName AS EmpLastName,
 M.FirstName AS MgrFirstName,
 M.LastName AS MgrLastName
FROM Employee E
 JOIN Employee M
 ON E.ManagerID= M.EmployeeID
```

Notice that some employees manage themselves, as indicated by the fact that ManagerID = EmployeeID for those employees.

3.4Knowledge Check

This chapter covered the crucial Join operation, which is vital in connecting data in one table with data from other tables. Without Join, the power of database systems would be significantly reduced. Using it, however, we're able to answer much more complicated and sophisticated queries connecting data from tables that seem widely separated in the database. The basic idea is to connect two

records if the value in one column of one record is the same as the value of a column in the other record. Based on that simple principle, we can navigate from one table to the next and see connections that would otherwise be difficult to make.

We showed basic information as to what the Join means and how it acts on multiple tables to connect them. We started with a relatively simple two-table join and then expanded to multiple tables. We also showed how a table can be joined to itself so that records in one part of the table are connected to records in other parts of the table. This is important for showing hierarchies between records in a table, such as showing an organizational structure chart.

This assessment can be taken online.

Chapter 4: Compounding Data: The Aggregating Functions

4.1Introduction: Built-in Functions and Aggregating Data

Learning Objectives

After reading this chapter, you will be able to:

- Understand some of the functions built into SQL
- Understand the use of the aggregate functions SUM, AVG, MAX, MIN, and COUNT
- See how to group records for statistical analysis

There are a number of functions built into SQL which help the query process. In this chapter we will look at several of these. These functions operate on dates, strings, and sets of records. We will also see how we can group records and use certain types of functions on them.

4.2Built-in Functions

There are a number of functions built into SQL that help the query process. In this chapter, we will look at several of these.

What exactly is a function?
A function is a concept from the field of mathematics. Basically a function receives an input through a parameter and replaces itself with an output value. A function always returns a value.

For example, one function that converts all the letters in a character string to upper case is called UPPER(x). So if an SQL query statement includes the phrase "UPPER(George)", that phrase is replaced in the SQL query statement by the string "GEORGE". The input to the function is "George" and the output, or replacement value, is "GEORGE". So when the SQL query is evaluated, it only

receives "GEORGE."

Another example might be the length function, named LEN(x). The length function takes as input a character string, but replaces it with an integer. For example, LEN('Bill Jones') is replaced by the integer 10. So LEN is the function, 'Bill Jones' is the input parameter and 10 is the output value that is returned. Again, a function always returns a value which replaces the function in the SQL query where it was used.

Functions operate on entire columns in a set of rows or on an individual column from a single row to return a new value. For example, the Sale table has a column named SaleDate. An individual Sale row contains the date of the sale in the SaleDate column. If we're interested in knowing the sales that took place in a particular month, we apply the appropriate built-in function (which you will recognize from Chapter 2) to the SaleDate column as follows. This expression will show only those rows where the month of the SaleDate is 10 (October). In other words, the date function to return the month takes a date as input and returns an integer value for the month value from that date. The SQL query only sees the integer month value in its evaluation.

The following statement uses the MONTH function to return an integer value which represents the month of the SaleDate and then compares that value to the number 10. So the result of the following statement is either TRUE (It is equal to 10) or FALSE (It is not equal to 10).

- Month(SaleDate) = 10

Month is a **built-in function**, so-called because it is "built-in" to the SQL language. Other built-in functions that work on date-time values are Day and Year, which return the day of the date and the year of the date, respectively. Built-in functions also exist to return the various parts (hour, minute, second) of

the time in a date-time value. To return the current date and time from the server's system clock, we use the "GetDate()" built-in function. Month, Day, and Year operate on a value from a single row and return values for that row. Although GetDate() requires no input, it too returns a value for each row that conforms to the query's WHERE clause. Built-in functions that behave this way are called **scalar functions**.

Other built-in functions work on columns in a set of rows. These are sum, average, max, and min that compute the sum, average, maximum, or minimum of the values in the column of a specified set of rows. Count is another function that simply counts how many values there are in a column or set of rows. These built-in functions are sometimes called "aggregate functions," "set functions," or "column functions," but we will call them **aggregate functions** in this book. They are covered in the next section.

Date Functions

You have already been using simple date functions. Here are several more examples of the application of date functions.

Query: List sales of sneakers sold in April.

Redcat Database Diagram

```
SELECT Sale.*
FROM Sale
 JOIN SaleItem
 ON Sale.SaleId = SaleItem.SaleId
 JOIN ProductBrief
 ON SaleItem.ProductID = ProductBrief.ProductID
WHERE Category = 'sneakers'
 AND Month(SaleDate) = 4
```

This is a query from Chapter 2. We have updated it for this chapter's database structure and used join terminology in the FROM clause.

Query: List sales of sneakers sold during the last 3 months of 2015.

Redcat Database Diagram

```
SELECT Sale.*
FROM Sale
 JOIN SaleItem
 ON Sale.SaleId = SaleItem.SaleId
 JOIN ProductBrief
 ON SaleItem.ProductID = ProductBrief.ProductID
WHERE Category = 'sneakers'
 AND MONTH(SaleDate) BETWEEN 10 AND 12
 AND YEAR(SaleDate) = 2015
```

Do all Database Management Systems have the same functions?
No. Different Database Management Systems use different names for various built in functions. For example, Microsoft SQL Server uses GetDate() to return the date and time from the system clock while Oracle uses SysDate. MySQL uses SysDate(), and PostgreSQL uses Current_Date. Although many are specified in the SQL standard, it is up to the architects of each DBMS to decide the specifics of the implementation.

This textbook configures itself—both the explanations and coded examples—according to the DBMS that you are using for the course.

In our examples, we've shown the date functions appearing only in the WHERE clause, but they can appear in the SELECT clause as well.

How do I format literals as dates?
In addition to the date functions described above, at times it is necessary to compare the entire date field to a literal (i.e., a number). Each DBMS uses a slightly different technique for recognizing a literal as a date and converting it to the correct internal format to affect the comparison.

Based on the type of the compared field, SQL Server attempts to convert the compared literal to a date format. The two primary formats recognized for date literals are 'mm/dd/yyyy' and 'yyyy-mm-dd'. For example '06/28/2017' and

'2017-06-28' are both valid dates for June 28, 2017. The literals must be enclosed in quote marks.

Operations and Scalar functions

Scalar functions can also be used with an SQL statement. Listed below are three types of scalar functions: Column functions and operations, Arithmetic functions, and String functions.

Column operations permit operations on multiple columns in the result set. Arithmetic operations also can work on multiple columns or literals and columns, as you learned earlier. Both Arithmetic functions and String functions are functions that operate on the value in a single field or column. Arithmetic functions work on numeric values and String functions work on string or character data. Normally the Arithemetic and Scalar functions are used in the SELECT clause, but can also be used as part of a condition in the WHERE clause.

Column operations

Column operations work on field names (i.e. column names) and go inline within an SQL statement and operate as normal mathematical operators.

- Arithmetic operations
  - '+' (Add), '-' (Subtract), '*' (multiply), '/' (divide)
  - Example: SELECT SaleID, (Shipping + Tax) AS AddonCost from Sale
- Concatenation operation
  - The arithmetic operation of + also serves as a concatenation operation for string columns.

- Example: Select LastName + ', ' + FirstName as CustomerName FROM Customer == "Jones, Bill"

Arithmetic Functions

The following lists a few of the arithmetic functions that are useful for operating on numeric fields. This is only a partial list.

- ABS(x)
  - Returns the absolute value of a number.
  - Example: SELECT ABS(DegreesInTurn) FROM FlightAngle == ABS(-120) returns 120
- SQRT (x)
  - Returns the square root of a number.
  - Example: SELECT SQRT(variance) FROM Analysis == SQRT(9) returns 3
- ROUND (x,d)
  - Rounds the numeric value to d decimal places.
  - Example: SELECT ROUND(Rate,2) FROM RateTable == ROUND(345.9284, 2) returns 345.93

String Functions

String functions perform many varied and elaborate functions on the character strings. The following list contains a few of the more useful string functions.

- LEN(x)
  - Returns the length of the character string. It works on literal strings or string values in a column.

    - Example: LEN ('My house is white.') returns 18.

- UPPER(x)
    - Returns all characters as upper case.
    - Example: UPPER('George') returns 'GEORGE'
- LOWER(x)
    - Returns all characters as lower case.
    - Example: LOWER('McDougal') returns 'mcdougal'
- CHARINDEX('value', x)
    - Returns the first location of value in x.
    - Example: CHARINDEX('hi', 'My white house') returns 5, which is the 5th character in the string.

- LTRIM (x)
    - Removes leading spaces.
    - Example: LTRIM(' Bill') returns 'Bill' without the leading spaces.
- RTRIM (x)
    - Removes trailing spaces.
    - Example: RTRIM(' Leather Shoes ') returns ' Leather Shoes' without the trailing spaces.
- CONCAT(x,y,[z]...)
    - Returns the strings concatenated together. It works just like the concatenation operator above.
    - Example: Concat(Lastname, ', ',firstname) == Jones, Bill

- SUBSTRING(x, s, n)
    - Returns the n characters from x beginning with character number s.
    - Example: SUBSTRING ('My White House',4,5) returns 'White' which is

5 characters long and beginning with the 4th character.

- REPLACE (x, f, t)
  - Replaces every occurrence of f with t in the original string, x.
  - Example: REPLACE('Good shoes at the shoe store', 'shoe','shirt') returns 'Good shirts at the shirt store'

In the following example we want to limit the number of characters returned for the Composition and the Description of products to the first 15 characters for Composition and the first 20 characters for Description. We also show how many characters are used for each product description.

Redcat Database Diagram

```
Select ProductID, ProductName,
SUBSTRING(Composition,1,15) AS &quot;Brief Composition&quot;,
LEN(Description) AS &quot;Desc Length&quot;,
SUBSTRING(Description,1,20) AS &quot;Brief Description&quot;
From Product
```

The following example shows the use of a scalar function in the WHERE clause. In this example we want all products that have leather in their composition. The function returns a value of 1 or more if the word 'Leather' is found. If it is not found in Composition, then the function returns 0. Earlier you learned how to search for a character string using the LIKE operator. This is another technique to find a substring within a longer character string.

Redcat Database Diagram

```
Select ProductID, ProductName, Composition
FROM Product
WHERE CHARINDEX ('Leather', Composition) &gt;= 1
```

Nesting Functions

Functions can be nested inside of other functions. The key to nesting functions is to remember that a function replaces itself with its value. So you need to be sure that the returned value's type is correct for the location where it is nested. In other words, if the outside function is expecting a number, then the nested function should return a number as its return value. Nested functions are evaluated from inside out, with the innermost functions evaluated first.

As an example, suppose we want to determine the length of the address in each address column. However, we know that the data is a little bit dirty because it has leading and trailing spaces that we don't want to count. So, let's nest the LTRIM and RTRIM inside of the length function.

LEN(LTRIM(RTRIM(Address)))
In this example the inner most function, RTRIM, is evaluated first. It expects a string as input and returns a string as output. The next innermost function, LTRIM, is evaluated next. It also takes strings as input and output. The outermost function, the length function, is evaluated last. It also accepts a string as input, but returns an integer as output.

Let's do one more example. Let's find the longest Address for the employees in the Redcat database. (You will learn about the MAX function in the next section.)

Redcat Database Diagram

```
SELECT MAX(LEN(Address)) AS LongestAddress FROM Employee
```

4.3Aggregate Functions

Consider these questions, which require aggregate functions.

What are the highest and lowest priced products? For sales in May of last year,

what was the average shipping cost? What is the total quantity of sneakers we've sold to customers in Oklahoma? How many different categories of shoes do we have?

These questions require statistical functions that examine a set of rows in a table and produce a single value. SQL provides five such aggregate functions. The five are **Sum, Avg, Count, Max,** and **Min**.

MAX and MIN

MAX(column name) and MIN(column name) look at a particular column across all of the rows that are eligible based on the WHERE clause and finds the maximum or minimum value. The input parameter may be a column name, or may be a calculation between columns.

Query: What are the highest and lowest prices for products?

Redcat Database Diagram

```
SELECT MAX(ListPrice) AS HighPrice,
 MIN(ListPrice) AS LowPrice
FROM Product
```

The Max and Min functions operate on a single column in a relation. They select the largest or the smallest value, respectively, to be found in that column. The solution for this query does not include a WHERE clause. For most queries, this need not be the case as our next example shows.

AVG

The AVG(column name) function calculates the average amount for the specified column. It adds up all the values for all of the rows and divides by the

number of rows. The input parameter may be a column name or a calculation between columns.

Query: For sales in May of 2014, what was the average shipping cost?

Redcat Database Diagram

```
SELECT AVG(Shipping) AS AvgShip
FROM Sale
WHERE MONTH(SaleDate) = 5
 AND YEAR(SaleDate) = 2014
```

Query: What is the average shipping and tax for December sales?

Redcat Database Diagram

```
SELECT AVG(Shipping + Tax) AS AvgShipTax
FROM Sale
WHERE MONTH(SaleDate) = 12
```

Notice the number of decimal places on the result. Although this answer is correct, it is much more readable if we conform to currency notation and only show two decimal places. We do this using the ROUND function that was introduced earlier.

Redcat Database Diagram

```
SELECT ROUND(AVG(Shipping),2) AS AvgShip
FROM Sale
WHERE MONTH(SaleDate) = 5
 AND YEAR(SaleDate) = 2014
```

Another option is to use the FORMAT function. The syntax for formatting numeric fields is FORMAT(fieldname, 'N2'). The 'N2' is a string literal and must be inside single quotes. The 'N' means it is numeric, and the '2' indicates with two decimal places. You should try it in the above query. Replace the ROUND syntax, with the FORMAT syntax.

To calculate the average, only rows in sales for May of the previous year are considered. As is normally the case in SQL, the WHERE clause restricts consideration to these rows.

Order of execution is important here:

1. The FROM clause identifies tables.
2. The WHERE clause selects the rows to be used in the aggregate function calculation.
3. The aggregate functions are calculated.

SUM

The SUM(Column name) function adds up all of the values for the specified column across all eligible rows based on the WHERE clause. The SUM may be for a particular column or it many be a calculated field between columns.

Query: What is the total quantity of sneakers we've sold to customers in Oklahoma?

Redcat Database Diagram

```
SELECT SUM(Quantity) AS TotalQty
FROM Customer C
 JOIN Sale S
 ON C.CustomerID = S.CustomerID
 JOIN SaleItem SI
 ON S.SaleID = SI.SaleID
 JOIN Product P
 ON P.ProductID = SI.ProductID
WHERE State = 'OK'
 AND Category = 'sneakers'
```

This solution required the joining of four relations—Customer, Sale, SaleItem, Product. This was necessary since Customer contains the state of residence (Oklahoma), SaleItem contains Quantity, Product contains the Category

(sneakers), and Sale provides the logical connection between them. After the joins are executed, we have one large relation and the Sum function can be executed on the Quantity column of that relation.

Additionally, SUM and AVG must be used with columns that are numeric. The other functions can be used either with numeric or with character string data. All the functions except Count may be used with computed expressions.

COUNT

The COUNT(item) counts the number of times the "item" appears in the eligible rows based on the WHERE clause. Normally the "item" is a column name, but it can also be the '*' which counts the rows returned. Note, when doing a join, the number of rows includes the total rows in the joined tables, again as constrained by the WHERE clause.

Query: How many different categories of shoes do we have?

Redcat Database Diagram

```
SELECT COUNT(DISTINCT Category) AS CatCount
FROM Product
```

Since the same category is repeated in several different rows, it is necessary to use the keyword "DISTINCT" in this query, so that the system will not count the same category more than once. DISTINCT may be used with any of the aggregate functions, although naturally, it is a redundant operator with the MAX and MIN functions, and it may not make sense with the SUM and AVG functions. Additionally, SUM and AVGvg must be used with columns that are numeric. The other functions can be used either with numeric or character string data. All the functions except COUNT may be used with computed expressions:

Count may refer to entire rows rather than just a single column:

Query: How many customers live in Kentucky?

Redcat Database Diagram

```
SELECT COUNT(*) AS CountKY
FROM Customer
WHERE STATE = 'KY'
```

As all these examples show, if an aggregate function appears in the SELECT clause, then nothing but aggregate functions may appear in that SELECT clause. The only exception to this occurs in conjunction with the GROUP BY clause which we examine below. Note that it does not make sense to try to use aggregate functions in the WHERE clause because an aggregate function does not apply to a single record.

Beware of Pitfalls!

Coming up with the correct solution for answering an English-language query can sometimes be tricky. We have to carefully analyze the meaning of the query (in English) and then make sure that the SQL solution we develop actually answers it. Look at this query:

Query: What is the average list price of products sold last month?

Proposed Solution:

- SELECT Avg(ListPrice) as AvgPrice
- FROM Sale S join SaleItem SI on S.SaleID = SI.SaleID join Product P on P.ProductID = SI.ProductID
- WHERE Month(SaleDate) = Month(GetDate()) – 1

Unfortunately, this solution won't work. Presumably, the user submitting the query is interested in identifying those products which sold last month and getting their average list price *regardless of how many times they were sold*. But the proposed query solution *does* use the number of times they were sold. An example using a shortened database clarifies this. Suppose, for example, there were only two sales last month, the first with two sale items and the second with 1. After taking natural joins as indicated in the FROM clause of the query, we get the following structure for our data:

Sale 1	– Sale item – Product 10 with list price $20.00
	– Sale item – Product 20 with list price $11.00
Sale 2	– Sale item – Product 10 with list price $20.00

The average price the solution will calculate is $17.00 = (20 + 11 + 20) ÷ 3 = 51 ÷ 3. But the average price actually desired by the user is $15.50 = (20 + 11) ÷ 2 = 31 ÷ 2. So what went wrong?

The problem is that when we join a table (like Product) to another table (like SaleItem), some rows in Product could appear more times in the joined tables than other rows. If we then calculate the average from the joined tables, we will probably come up with an erroneous result. This is easy to see with our small example, but if we're dealing with a database of thousands of products and even more sales, we could make this mistake and never realize it. Notice that this could also happen with the SUM and COUNT functions—although not with MAX and MIN. Notice also that using the Distinct keyword (applied to ListPrice) wouldn't help either. It would give us the correct solution in this simple example but not in a case where there are many products and several of them have the same list price.

So what is the solution? How can we answer queries like these? We will see when we study subqueries below. Before we do that, however, we need to explore the other two clauses of the SQL Select statement—GROUP BY and HAVING.

4.4Group by and Having Clauses

GROUP BY Clause

Management is often interested in knowing statistical information as it applies to each group in a set of groups. Consider the following query.

Query: For each product category, what is the highest list price for a product in that category?

To solve this query, we use the **GROUP BY** clause. This clause divides the products into groups, one group for each category. Then, in the SELECT clause we determine the maximum list price in each group. We do this in SQL in this manner:

Redcat Database Diagram

```
SELECT Category,
MAX(ListPrice) AS MaxListPrice
FROM Product
GROUP BY Category
```

In processing this query, the system proceeds by first organizing the rows of Product into groups, using the following rule. Rows are placed in the same group if and only if they have the same Category. Now the SELECT clause is applied to each group. Since a given group can have only one value for Category, there is no ambiguity as to the value of Category for that group. The SELECT clause

calls for the Category to be displayed and for the Max(ListPrice) to be calculated and displayed *for each group*. The result is as shown. Only column names appearing in a GROUP BY clause may appear in a SELECT clause with an aggregate function. Note that Category can appear in the SELECT clause, since it appeared in the GROUP BY clause.

Let's look at another example where we desire to have more columns in the SELECT clause. In this case, we want the total amount spent by a customer in a given year. To do this we need three tables, Customer, Sale, and SaleItem. Note that the amount spent on a sale is in the SaleItem table and is the product of SalePrice and Quantity. For the query, we will join these three tables together, and sum the dollar amounts and group by CustomerID. However, we also want to include the Customer's name. Thus all the fields in the SELECT clause, other than the Sum function need to be included in the GROUP BY clause.

Query: What is the total amount spent by each customer in 2015? Include CustomerID, FirstName and LastName.

To solve this query, we do this in SQL in this manner:

Redcat Database Diagram

```
SELECT C.CustomerID,
        Firstname,
        LastName,
SUM(SalePrice * Quantity) AS TotalAmt
FROM Customer C
        JOIN Sale S
        ON C.CustomerID = S.CustomerID
        JOIN SaleItem SI
                ON S.SaleID = SI.SaleID
                WHERE YEAR(SaleDate) = '2015'
                GROUP BY C.CustomerID,
                        FirstName,
                        LastName
```

The GROUP BY clause changes the nature of the query. Without a GROUP BY clause, the query is simply a query on rows—those rows identified by the FROM

and the WHERE clauses. However, once the GROUP BY clause is introduced, the query *changes from a query on rows to a query on groups*, and the SELECT clause applies to the groups identified by the columns named in the GROUP BY clause.

Let's make this a little more interesting. What if we also wanted to know how many pairs of shoes each customer bought? In the join we have each SaleItem record with the quantity field. If we just add up all the quantities for each customer, we should know how many pairs of shoes each customer bought. Let's add that clause to the query. As you learned earlier, it is possible to use multiple aggregate functions simultaneously—even with the GROUP BY clause.

Redcat Database Diagram

```
SELECT C.CustomerID, Firstname, LastName,
 SUM(SalePrice * Quantity) AS TotalAmt,
 SUM(Quantity) AS NumPairs
FROM Customer C
         JOIN Sale S ON C.CustomerID = S.CustomerID
         JOIN SaleItem SI ON S.SaleID = SI.SaleID
        WHERE YEAR(SaleDate) = 2015
        GROUP BY C.CustomerID, FirstName, LastName
```

We can ask other kinds of interesting questions about this query. How many different times did each customer buy shoes? How many times did each customer buy more than one pair of the same shoes in the same size? What if we wanted to know the maximum amount a customer spent on any given sale? That would change the SELECT clause and the GROUP BY clause to include SaleID. You might try some of these queries.

It is permissible to use the WHERE clause with GROUP BY:

Query: For each product category, what is the average list price of black shoes?

Redcat Database Diagram

```
SELECT Category,
AVG(ListPrice)
FROM Product
WHERE Color = 'Black'
GROUP BY Category
```

The WHERE clause is executed before the GROUP BY clause. Thus, no group may contain a row with a color other than black. The rows with color black are grouped by Category and the SELECT clause is applied to each group.

HAVING Clause

We can also apply a selection condition to the groups formed by the GROUP BY clause. This is done with the **HAVING** clause. Suppose, for example, we wish to make one of the previous queries more specific.

Query: For each category containing more than 50 products, what is the maximum list price in that category?

We could indicate this by the proper use of the HAVING clause:

Redcat Database Diagram

```
SELECT Category,
MAX(ListPrice) AS MaxListPrice
FROM Product
GROUP BY Category
HAVING COUNT(*) &gt; 50
```

The difference between the WHERE clause and the HAVING clause is that the WHERE clause is applied to *rows* while the HAVING clause is applied to *groups*. In this example, Count(*) > 50 means that only groups having more than 50 rows are to be considered in the result of the query.

A query may contain both a WHERE clause and a HAVING clause. In that case, the WHERE clause is applied first since it applies before the groups are formed.

For example, look at this revision to a query given above:

Query: For each product category, what is the average list price of black shoes? Consider only those categories having no product priced over $100.

Redcat Database Diagram

```
SELECT Category,
AVG(ListPrice)
FROM Product
WHERE Color = 'Black'
GROUP BY Category
HAVING MAX(ListPrice) &lt;= 100
```

Observe that, starting with the FROM clause, the clauses in SQL statements are applied in order, and then the SELECT and ORDER BY clauses are applied last. Thus, the WHERE clause is applied to the Product relation, and all rows having a color other than black are eliminated. The rows remaining are grouped by category, with all rows of the same category being in the same group. This creates a number of groups—one for each value of category. The HAVING clause is then applied to each of the groups, and those having a maximum list price above $100.00 are eliminated. Finally, the SELECT clause is applied to the groups that remain.

Finally, we can also sort the results with the ORDER BY clause. Let's take the last query and sort the data by the category. Obviously we can only sort the data on fields that appear in the SELECT clause. Let's sort the data descending on Category. In this example, there is only one column in the SELECT clause, but if there are multiple columns, then the data could be sorted on a single column or with multiple columns.

Redcat Database Diagram

```
SELECT Category,
 AVG(ListPrice)
FROM Product
WHERE Color = 'Black'
```

```
GROUP BY Category
HAVING MAX(ListPrice) &lt;= 100
ORDER BY Category DESC
```

How do I know which to use, WHERE clause or HAVING clause?
The WHERE clause tests values in individual rows and is applied before the HAVING clause. So if you are testing individual values, use WHERE. The HAVING clause test values or counts on the groups that are formed due to an aggregate function. HAVING is applied after the groups are formed. So use HAVING if you want to qualify an entire group. You can also use an aggregate function inside the HAVING clause.

4.5Knowledge Check

In this chapter we saw how SQL contains a number of built-in functions, which operate on individual fields—such as date and string functions—or on groups of records, such as the aggregate functions of SUM, AVG, MAX, MIN, and COUNT.

These functions can be used for their own sake to give us information for display through the SELECT clause, or they can also be used to give new selection criteria for use in the WHERE clause.

GROUP BY and HAVING clauses open up a whole new realm of possibility for SQL queries because the presence of a GROUP BY clause changes the nature of a query. Instead of being a query on rows, when a query has a GROUP BY clause, it is now a query on groups of rows. The HAVING clause is a selection clause for groups. The WHERE clause selects out rows for consideration, but the HAVING clause selects out groups for consideration.

This assessment can be taken online.

Chapter 5: Combining Queries: Queries with Subqueries

5.1Introduction: Subqueries

Learning Objectives

After reading this chapter, you will be able to:

- Understand subqueries and their use
- Create subqueries within main queries
- Use both correlated and non-correlated subqueries
- Use aggregate functions in subqueries
- Be familiar with subqueries in the SELECT and FROM clauses
- Learn about other database concepts beyond the scope of this book

A **subquery** is a query that is written inside another query. A synonym for a subquery is a **nested query**. A subquery can be used inside of SELECT, INSERT, UPDATE, or DELETE SQL statement. A subquery's syntax follows the same rules as a regular query. It must be enclosed in parentheses and will contain the following:

- A regular SELECT clause
- A regular FROM clause
- Options WHERE, GROUP By, or HAVING clauses

Subqueries can be nested. A subquery can be used within the clauses of the outside query, namely within:

- The SELECT clause
- The FROM clause
- The WHERE clause
- The HAVING clause

5.2Subqueries in the WHERE clause

Subqueries are most frequently found in the WHERE clause. As such, a subquery in the WHERE clause is evaluated first and the result is used to evaluate the outside query. A subquery in a WHERE clause may return either a set of values or a single value, depending on the context and need of the outer query.

Query: What are the categories of products made by manufacturers located in New Jersey?

- SELECT Category
- FROM Product
- WHERE ManufacturerID IN
  - (SELECT ManufacturerID
  - FROM Manufacturer
  - WHERE State = 'NJ')

Subqueries are always shown within parentheses. The subquery in this example is

- (SELECT ManufacturerID
- FROM Manufacturer
- WHERE State = 'NJ')

The query that contains this subquery is called the **outer query** or the **main query**. The subquery causes the following set of ManufacturerIDs to be generated:

Redcat Database Diagram

```
SELECT ManufacturerID
    FROM Manufacturer
    WHERE State = 'NJ'
```

This set of IDs then takes the place of the subquery in the outer query. At this point, the outer query is executed using the set generated by the subquery. The outer query causes each row in Product to be evaluated with respect to the WHERE clause. If the row's ManufacturerID is *in* the set generated by the subquery, then the Category of the row is selected and displayed as part of the result of the query:

Redcat Database Diagram

```
SELECT Category
FROM Product
WHERE ManufacturerID IN
         (SELECT ManufacturerID
         FROM Manufacturer
         WHERE State = 'NJ')
```

It is very important that the SELECT clause of the subquery contain ManufacturerID and *only* ManufacturerID. Otherwise, the WHERE clause of the outer query, which states that ManufacturerID is *in* a set of ManufacturerIDs, would not make sense.

Note that the subquery may logically be executed before *any* row is examined by the main query. In a sense, the subquery is independent of the main query. It could be executed as a query in its own right. We say that this kind of subquery is **non-correlated** or not correlated to the main query. As we will see shortly, subqueries may also be **correlated**.

Here is an example of a subquery within a subquery.

Query: List sales of black sneakers.

Redcat Database Diagram

```
SELECT *
FROM Sale
WHERE SaleID IN
         (SELECT SaleID
         FROM SaleItem
         WHERE ProductID IN
        (SELECT ProductID
                 FROM Product
                 WHERE Category = 'sneakers' AND Color = 'Black'))
```

Observe that we did not have to prefix any column names with relation names. This is because each subquery deals with only one relation, so no ambiguity can result. The evaluation of this query proceeds from the inside out. Thus, the innermost (or "bottom-most") subquery is evaluated first, then the subquery that contains it, followed by the outer query.

This query is identical to a query we used to illustrate the join in Section 3.3. Therefore, subqueries offer a partial alternative to the join.

Which is preferable to use: a join or a subquery?
It depends. There are two issues to consider: First, which method is easier to understand, and second, which is more efficient. You should become familiar with using both JOIN and subqueries. Use whichever is easiest to understand a solution. Efficiency also varies. The simple correlated subquery above is probably more efficient than a join, but complex subqueries are often resource-intensive. Focus primarily on which method is easier for you to solve the problem. Because you learned to join first, it may seem more natural, but a subquery is often easier to understand and provides a more straightforward solution.

5.3Correlated Subqueries

The subqueries we have studied so far are independent of the main queries that use them. By this we mean the subqueries could exist as queries in their own right. We will now look at a class of subqueries whose value upon execution

depends on the row being examined by the main query. Such subqueries are called **correlated subqueries**.

Query: List employees who were hired before their manager.

The pivotal word in this query is "their." That is, the manager row to be examined in the subquery depends directly upon the employee row being examined. This query can be solved by using a correlated subquery. For each employee, we find the manager record and then compare the hire dates. Let's only display their EmployeeIDs, Names, and Hiredates.

Redcat Database Diagram

```
SELECT EmployeeID,
       FirstName,
       LastName,
       HireDate
FROM   Employee E
WHERE  E.Hiredate &lt;
( SELECT M.Hiredate
FROM Employee M
WHERE M.EmployeeID = E.ManagerID)
```

The logical steps involved in executing this query:

1. The system makes two copies of the Employee relation: copy E (for Employee) and copy M (for Manager).
2. The system then examines each row of E. A given row (called "E") is selected if it satisfies the condition in the WHERE clause. This condition states that the row will be selected if its Hiredate is less than the Hiredate generated by the subquery.
3. The subquery selects the Hiredate from the row of M whose EmployeeID is equal to that of the ManagerID of the row of E currently being examined by the main query. This is the Hiredate of E's manager.

Since E.Hiredate can only be compared to a single value, the subquery must, of

necessity, generate only one value. This value *changes depending on the row of E being examined*. Thus, the subquery is "correlated" to the main query. We will see more applications for correlated subqueries next as we study aggregate functions with subqueries.

Let's do another example.

In the next section of the text, subqueries with aggregate functions are explained. However, as an introduction, and to help explain correlated subqueries, the next example illustrates a correlated subquery that uses an aggregate function.

Query: For each manufacturer, show its higher-priced products. We define higher-priced products to be those whose list price is higher than the average list price for that manufacturer.

Notice that the operative phrase here is that we want the average price of products for a particular manufacturer. In other words, we do not want the average price of all products. For each product from a given manufacturer, we want the average price of that manufacturer's products.

Here is a preliminary query that shows each manufacturer and the average price of its products. We have joined Manufacturer to Product in order to show the Manufacturer's name, but that is not necessary for the query to work.

Redcat Database Diagram

```
SELECT   M.ManufacturerID,
         ManufacturerName,
         ROUND(AVG(listPrice),2) as AveragePrice
FROM     Product P
   JOIN  Manufacturer M on M.ManufacturerID = P.ManufacturerID
    GROUP BY M.ManufacturerID, ManufacturerName
    ORDER BY M.ManufacturerID
```

Notice that each manufacturer has a different value for the average price. To answer the query in our example, the system needs to look at a product,

determine its manufacturer, calculate the average for that manufacturer, and then compare the product's list price to see if it qualifies as being higher than the average for the product's manufacturer. Figure 5.1 illustrates how this occurs.

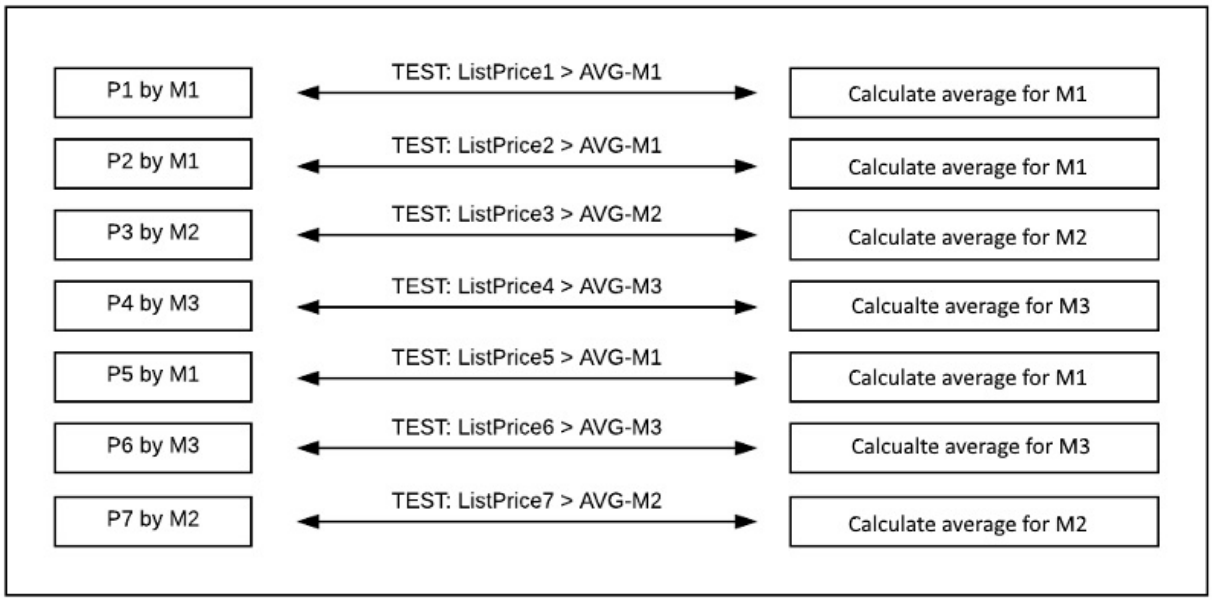

Figure 5.1: Finding above-average values

Here is the code that accomplishes that purpose. Notice that in the subquery, the WHERE clause contains a reference to the ManufacturerID of the main query in order to find out which Manufacturer to use to calculate the average. In a correlated subquery, there is always a clause, usually a WHERE clause, that links the subquery to the main query. In this case, the linking field is ManufacturerID. For this query, we dropped the join to Manufacturer, since it is not necessary for the calculations. You can compare the list prices of the result in this query against the average list price for a manufacturer from the previous, preliminary query to verify that the answer is correct.

Redcat Database Diagram

```
SELECT  ManufacturerID,
        ProductID,
        ProductName,
        ListPrice
```

```
FROM    Product P1
WHERE   ListPrice &gt;
    ( SELECT AVG(ListPrice)
        FROM   Product P2
        WHERE  P2.ManufacturerID = P1.ManufacturerID)
    ORDER BY ManufacturerID
```

5.4Aggregate Functions with Subqueries

Subqueries in the WHERE clause

An aggregate function may appear only in a SELECT clause or a HAVING clause; however, a SELECT clause containing an aggregate function may be part of a subquery.

Here is an example of such a subquery:

Query: Which employees receive a higher-than-average salary?

Redcat Database Diagram

```
SELECT FirstName, LastName, Salary
FROM Employee E JOIN SalaryEmployee SE ON E.EmployeeID = SE.EmployeeID
WHERE Salary &gt;
 (SELECT AVG(Salary)
 FROM SalaryEmployee)
```

Note that this subquery is a non-correlated subquery that produces precisely one value: the average salary. The main query selects an employee only if the salary for that employee is above this calculated average.

Correlated subqueries may also be used with aggregate functions:

Query: Which employees receive a salary higher than the average for employees reporting to the employee's manager?

In this case, instead of calculating a single average for all employees, we must

calculate an average for each group of employees reporting to the same manager. Moreover, this calculation must be performed anew for each employee being examined by the main query.

Redcat Database Diagram

```
SELECT FirstName, LastName, Salary
FROM Employee E1 JOIN SalaryEmployee SE1 ON E1.EmployeeID = SE1.Employe
WHERE Salary &gt;
 (SELECT AVG(Salary)
 FROM Employee E2 JOIN SalaryEmployee SE2 ON E2.EmployeeID = SE2.Emplo
 WHERE E2.ManagerID = E1.ManagerID)
```

The WHERE clause of the subquery contains the crucial correlation condition. This condition guarantees that the average will only be calculated for those employees having the same supervisor as the employee being examined by the main query.

Subqueries in the HAVING clause

Query: For each product category, what is the average list price of black shoes? Consider only those categories whose average list price is below the average list price for all black shoes.

Redcat Database Diagram

```
SELECT Category,
 AVG(ListPrice)
FROM Product
WHERE Color = 'Black'
GROUP BY Category
HAVING AVG(ListPrice) &lt;
   (SELECT AVG(ListPrice)
    FROM Product
    WHERE Color = 'Black')
```

Avoiding the Pitfalls

When we previously discussed aggregate functions, we warned that care must be taken in some queries to avoid the pitfalls of miscalculation:

Query: What is the average list price of products sold this month?

Proposed Solution:

- SELECT Avg(ListPrice)
- FROM Sale S join SaleItem SI on S.SaleID = SI.SaleID join Product P on SI.ProductID = P.ProductID
- WHERE Month(SaleDate) = Month(GETDATE())

As shown earlier, the problem with this solution is that in joining several tables the average list price from the Product table would be calculated incorrectly because different products would appear at various numbers of times in the joined table. To avoid this problem, we use a subquery:

Correct Solution:

Redcat Database Diagram

```
SELECT AVG(ListPrice) AS AveragePrice
FROM ProductBrief
WHERE ProductID IN
 (SELECT ProductID
 FROM SaleItem
 WHERE SaleID IN
 (SELECT SaleID
 FROM Sale
 WHERE MONTH(SaleDate) = MONTH(GETDATE())
                    )
            )
```

The main query in this solution looks only at rows in the Product table and takes the average list price from those rows that are selected. Of course, each selected row is represented just once in the calculation of the average so that the earlier error is not repeated.

The selection criterion for rows in the Product table is found in the subquery, which is part of the main query's WHERE clause. This WHERE clause says to select a Product row if its ID is found in some SaleItem for a Sale that took place this month. It doesn't matter if the product appeared in one SaleItem or in a hundred—the product is selected once and only once if it appears in at least one SaleItem of this month's sales.

5.5Subqueries in the SELECT clause

The SELECT clause of a query determines which columns are returned from a query. As we learned earlier, a regular, non-correlated subquery simply returns a value that is used for all rows of the main query. Hence, using a regular, non-correlated subquery in the SELECT clause will produce a constant value across all rows.

A more useful reason to include a subquery in a SELECT clause is to use a correlated subquery so that each row of the main query will include a distinct value from the subquery. As mentioned earlier, the correlated subquery must return only one value for each row of the main query. A frequent use of a subquery in the SELECT clause is to use an aggregate function as the value returned by the subquery. For example, the main query might want to show results such as Customer name, and a total (using the aggregate SUM function) amount spent across a group of Sales to that customer.

For our example, let's assume we want to show the customer ID, the customer name, and the total amount of sales made to that customer. We can use a subquery in the SELECT clause with an aggregate SUM function to return the total value.

In the following example, there are four columns to be returned: CustomerID, FirstName, LastName, and a calculated column called TotalSold. The TotalSold

column is the sum of all of the SalePrice values for a particular customer. The subquery in the SELECT clause is rather sophisticated in that it includes a join of the Sale and SaleItem tables. The SalePrice values are in the SaleItem table, but the CustomerID is in the Sale table, so a join is required. The WHERE clause within the subquery correlates the calculated values with a particular CustomerID from the outside query.

This query could be made more elaborate by including a WHERE clause on the outside query to return such things as particular customers in a state or a city.

Redcat Database Diagram

```
SELECT CustomerID,
 FirstName,
 Lastname,
 (SELECT SUM(SalePrice * Quantity)
 FROM SaleItem SI
         JOIN Sale S
 ON SI.SaleID = S.SaleID
 WHERE S.CustomerID = C.CustomerID) AS TotalSold
FROM Customer C
```

Notice that the question for this query is similar to the question that was posed in the section on GROUP BY in Chapter 4 Section 4.4. The solution will be slightly different for this query because this query returns all the customers, even those without data in the sale table. This query is also a little more flexible because additional columns, and even columns from joined tables, can be included in the result set without having to include them in the GROUP BY clause.

5.6Subqueries in the FROM clause

Joining a Table with a Subquery Table

The purpose of the FROM clause in a query is to identify the table or tables that

are used in the query. Normally this is just the name of a permanent database table, which is often called a base table.

Sometimes the table to be used in the FROM clause needs to only be temporary. Thus, a subquery is appropriate because a subquery in a FROM clause is a temporary table. However, it must be given a name as an identifier for the primary query.

In the previous section, we found the total amount sold to a customer with a subquery in the SELECT clause. We can solve this same request by using a subquery in the FROM clause. However, this solution is more complex than the previous one. The subquery essentially does all the work here. Note that it is not a correlated subquery. It creates a full table that is then used as any other database table. It creates a record for each CustomerID value with the TotalSales amount. The GROUP BY clause is required to group the data together. Then, the SumQuery table is joined to the Customer table to get customer names.

As can be seen, there are frequently several different approaches to write SQL to solve the same problem. As we saw in Section 4.4, another solution might be to simply join all the tables together and use the GROUP BY function on the primary query to get the total value.

Redcat Database Diagram

```
SELECT C.CustomerID,
         FirstName,
         Lastname,
         TotalSold
FROM Customer C
 JOIN (SELECT CustomerID,
 SUM(Quantity * SalePrice) AS TotalSold
 FROM Sale S
        JOIN SaleItem SI
         ON S.SaleID = SI.SaleID
         GROUP BY CustomerID) SumQuery
         ON C.CustomerID = SumQuery.CustomerID
```

The above subquery is a non-correlated subquery. At first glance, it may appear

as though it is a correlated subquery due to the inclusion of the CustomerID in both the main query and the subquery. However, on closer inspection, it is evident that the CustomerID in the subquery is only in the SELECT clause of the subquery, and therefore the subquery is NOT executed for each row of the main query. The subquery does create an entire table consisting of all the CustomerIDs and the TotalSold. That table is then joined with the Customer table.

When do I use the keyword "AS"?
When we use a subquery in the SELECT clause, we are identifying a new column. It is best to use the "AS" keyword to name that new column. For example, we might say "Select FieldOne, FieldTwo, (select max(FieldName) from someTable) AS MaxAmount)..." If we don't name the query, SQL usually just provides some dummy field name.

When we use a subquery in the FROM clause, we are identifying a new table, and it MUST have a name. In this case, however, we do NOT use the AS clause. For example, we might say "SELECT * From table1, (SELECT FieldOne, FieldTwo from anotherTable) FromQuery ..." Notice that we named the table "FromQuery" without using the AS clause.

Some DBMSs do not explicitly adhere to these rules (such as SQL Server). Other DBMSs, however, do enforce this syntax precisely (such as Oracle).

Standalone Subquery

In this example, we will use a subquery as the only table in the FROM clause.

In the SaleItem table, there are three fields that together serve as the key for the table. Remember that the key, either a single column or a combination of columns, must yield a unique value for each row of the table. One quick way to verify that the key that was chosen uniquely identifies each record in the

database is to check that the number of unique key values is the same as the number of rows in the table.

In each of the queries below, the main query simply counts all the values in the table produced by the FROM clause. With the use of a subquery in the FROM clause in the last two queries, we are able to build tables that can be counted.

Let's check out the SaleItem table. To do this, we will create three short queries:

- **Query 1:** Count all the rows in the table.
- **Query 2:** Count the unique values of the three-column key.
- **Query 3:** Count the unique values of a two-column key.

We expect the calculated values in the first two queries to be equal, but the third query to show fewer unique values.

Query 1: Count all the rows in the table.

Redcat Database Diagram

```
SELECT COUNT(*) AS TotalRows
FROM SaleItem
```

Query 2: Count the unique values of the three-column key.

Redcat Database Diagram

```
SELECT COUNT(*) AS RecordCount
FROM (SELECT
        DISTINCT SaleId,
        ProductID,
                ItemSize
        FROM SaleItem) QueryAlias
```

Let's check to see if SaleID and ProductID are sufficient as a key.

Query 3: Count the unique values of a two-column key.

Redcat Database Diagram

```
SELECT COUNT(*) AS RecordCount
FROM (SELECT
        DISTINCT SaleId,
                        ProductID
        FROM SaleItem) QueryAlias
```

5.7Knowledge Check

This chapter introduced the concept of subqueries, or queries within queries. Subqueries give a different approach to solving queries, such as those requiring joins, while also providing a way to do things that cannot otherwise be done using the SQL capabilities discussed in earlier chapters.

Subqueries can be used in many of the clauses of the SELECT statement, and are also found in the SQL INSERT, UPDATE, and DELETE statements. Subqueries generate query results which may be a single value, such as the results from an aggregate function, or an entire table whose existence is just for the duration of the query. This query result can then be used by the main query in a way that completes the need of the main query.

Subqueries can be correlated or non-correlated. Non-correlated subqueries can stand as queries on their own, while correlated subqueries depend on the outer or main query for the complete information needed for their result. Aggregate functions can also be used in subqueries in both the SELECT and HAVING clauses.

Although the use of subqueries is most easily seen and used as part of the WHERE clause in a main query, it can also be used in the SELECT and FROM clauses, as was noted and discussed in this chapter.

Closing the chapter, we briefly described more advanced topics not covered in

this book. Databases provide much informational power, but as a consequence, require a great deal of effort in design, population, and maintenance. We invite you to explore these additional topics in *Database Management and Design*, also available from MyEducator.

This assessment can be taken online.

Chapter 6: Updating the Database: Create, Insert, Update, Delete

6.1Introduction: Define the schema and update data

Learning Objectives

After reading this chapter, you will be able to:

- Create database tables including name, column definitions, keys and contraints
- Understand data types used for field definitions
- Define key fields including primary keys and foreign keys
- Modify database schema by changing table structure
- Use Insert, Update and Delete statements to update database data
- Copy data from one table directly into another table
- Access other databases including a Food Inspection database and a Trucking database

In order for us to query a database, there must be data in the database. Also, sometimes existing data needs to be changed because of changes in the real world or because of data entry errors. Finally, it's sometimes necessary to delete data. In this chapter, we show you how to define, populate, and maintain a database. The next section reviews the database schema definition, while Section 6.3 discusses the database Insert, Update, and Delete statements.

First, we must actually define a relational database schema in SQL. To do this we have to define tables with their column, key, and foreign key definitions and thus create a database schema. This is done with SQL's Data Definition Language (DDL) that we use in this section to define the Red Cat Shoes database schema.

However, before the tables in a database can be defined, the database itself must be specified. But defining a database or database schema in SQL is DBMS dependent since each commercial DBMS takes its own approach. Therefore, we will not look specifically at schema or database definitions and will assume that

the Red Cat Shoes database has already been defined, although it does not yet have any tables in it. Therefore, our task will be to define the tables.

6.2Schema Definition

There are various activities that are involved in defining the database schema. The first activity is to define the tables. Within this step, the columns of the tables are defined and the data type of each column is identified. The primary keys and foreign keys, along with any other constraints, must also be defined within the schema. The following paragraphs explain how this is done.

Defining Tables

Tables are defined in three steps:

1. The name of the table is given.
2. Each column is defined, possibly including column constraints.
3. Table constraints are defined—these are key and foreign key definitions.

We start by looking at the definition we gave for some of the tables in the relational Red Cat Shoes database we gave above. We are primarily interested in the Customer and Sale tables.

- Customer (CustomerID, FirstName, LastName, StreetAddress, City, State, PostalCode, Country, Phone)
- Sale (SaleID, SaleDate, Tax, Shipping, CustomerID)
  - Foreign Key: CustomerID references Customer
- SaleItem (SaleID, ProductID, ItemSize, Quantity, SalePrice)
  - Foreign Key: SaleID references Sale
  - Foreign Key: ProductID references Product

These tables already contain most of the elements we need for an SQL table definition: We have the table names (Customer and Sale), column names, primary key identifications, and a foreign key definition. The only things missing are data type definitions and constraints for the columns and a delete specification for the foreign key (see below). So if we directly convert the relational definitions, but add data types and constraints to the columns and tables, the SQL definitions will be as shown in the Create Table statements below.

Here is the data diagram of the Redcat entities that you first saw as Figure 1.3 in Chapter 1. Note the relationships between the Customer, the Sale and the SaleItem. They will be referenced in the following discussion.

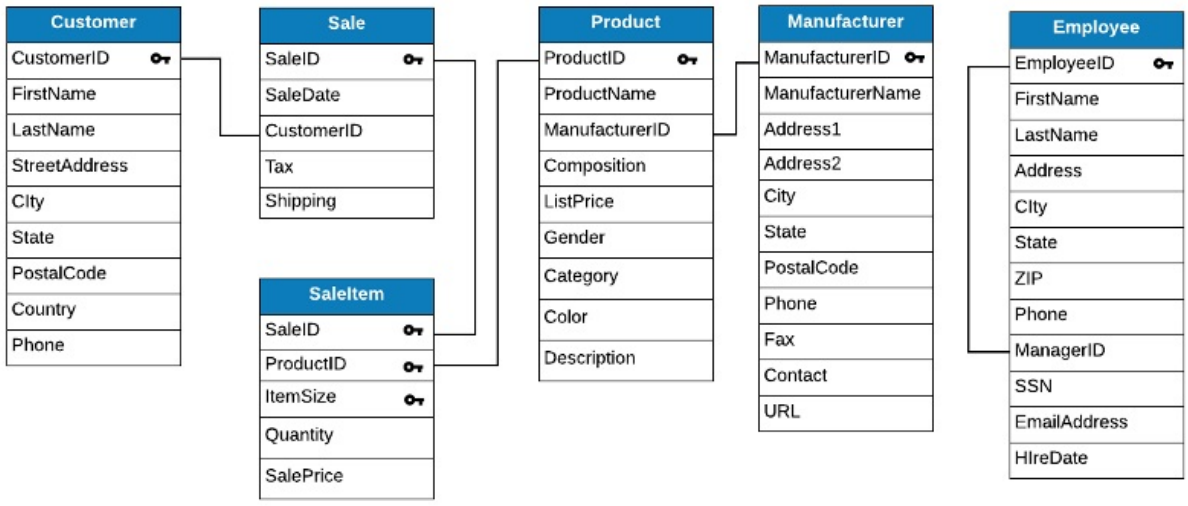

Figure 6.1: Redcat Data Model with Relationships

There is one SQL statement that will be useful for you. If you execute the statement to create a new table but decide that you want to delete the newly created table, you can use the Drop statement. The DROP TABLE statement is used to delete a table. Obviously you should be very very careful when using this statement. A dropped table is gone forever. The syntax is simply:

- DROP TABLE table-name

The DROP TABLE statement gives an error message if there is no such table in the database. That is usually not a problem. However, some DBMS systems have extended the DROP TABLE statement to elminate that error. The extended statement first tests to see if a table exists, and if so it then deletes it. This statement always executes successfully. Of course, there is no way to distinguish between a whether a table actually existed and needed to be dropped, or whether there was no table. The syntax for this extended version is:

- DROP TABLE IF EXISTS table-name

How do I make a correction to a table I already created?
To make changes to a newly created table, it is best to DROP the table and rerun a corrected CREATE TABLE statement. If you want to rerun a CREATE TABLE statement, you must first remove the table that was created in the previous execution. If you do not, it will cause an error because the table already exists. To allow a specific CREATE TABLE statement to execute multiple times, it is preceded by a statement to drop that table. However, the non-extended Drop Table statement may cause an error the first time you execute it because the table it is dropping does not yet exist in your table schema. That is not a problem.

Because tables reference other tables with foreign keys, and those foreign keys cannot be null, sometimes we must drop multiple tables. The order in which we drop tables is also important. For example we cannot drop the Customer table if there are Customer records that are referenced in the Sale table. Also, we cannot drop the Sale table if there are SaleItems referencing Sales. Thus, we always start at the lowest level, dropping SaleItem first, then Sale, then Customer.

So, in the first set of statements below, if the Sale table or the SaleItem table already exists, we must drop them *before* dropping the Customer table because

the SaleItem table has a foreign key that references the Sale table, and the Sale table has a foreign key that references the Customer table. This relationship would prevent us from successfully dropping the Customer table. After these tables are dropped, we drop the Customer table so we can create it again.

Redcat Database Diagram

```
DROP TABLE IF EXISTS SaleItem;
DROP TABLE IF EXISTS Sale;
DROP TABLE IF EXISTS Customer;
CREATE TABLE Customer (
         CustomerID INT NOT NULL,
         FirstName CHAR(30) NOT NULL,
         LastName CHAR(30) NOT NULL,
         StreetAddress VARCHAR(50) NOT NULL,
         City VARCHAR(50) NOT NULL,
         State CHAR(2) NOT NULL,
         PostalCode CHAR(6) NOT NULL,
         Country VARCHAR (50),
         Phone CHAR (15),
 PRIMARY KEY (CustomerID)
               )
```

Redcat Database Diagram

```
DROP TABLE IF EXISTS SaleItem;
DROP TABLE IF EXISTS Sale;
CREATE TABLE Sale (
         SaleID INT NOT NULL,
         SaleDate DATE NOT NULL,
         Tax NUMERIC(8,2) NOT NULL,
         Shipping NUMERIC(8,2) NOT NULL,
         CustomerID INT,
 PRIMARY KEY (SaleID),
 FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) ON DELETE SET NUl
                        )
```

What are NULL values and what does NOT NULL mean?
A **NULL** value in a database column is merely a statement that we don't know what the value should be, or that no value applies to that particular column for this particular record. Null values are not 0 nor are they blank. They're "NULL." "NULL" is a special database term, and a null value is simply a flag that tells us "unknown" or "not applicable."

NOT NULL on a column name means that that column must always have a

value. A NULL value is not allowed for that field in any row of data. For example, the CustomerID is a the primary key for the rows in the table. How could the table function if there were rows that did not have values in that field. Other mandatory columns are also specified as NOT NULL.

Explanation: Data Types

The data types we have given in the definition above are standard SQL data types. Commercial DBMSs, such as Access, have their own versions of these. Our discussion compares some SQL data types with their versions in Access.

The table IDs (CustomerID and SaleID for these two tables) are typically integers for which uniqueness within the table must be guaranteed so that these IDs can be used as primary keys for their tables. In Access, the data type is often AutoNumber, which means that the field is an integer data type, but is automatically generated by Access for each new row added to the table. Typically, its value is the next integer in sequence. So, for example, the first customer record entered has CustomerID 1, the second has CustomerID 2, and so on. If a record is deleted, the old value is not used again. Access simply increments the last ID value generated by 1 for the next record entered into the table.

Single-columned primary keys for tables not subsets of other tables commonly, though not always, have a data type of AutoNumber. Those that do not are often also integers, although they originated in some way that prevented them from having the AutoNumber data type.

The Char and VarChar data types are used for text or string-type data. String data types in Access are specifically ShortText and LongText fields, but they do not correspond exactly to the Char and VarChar data types of SQL. In SQL, a type designation like Char(30) means alphabetic, numeric, and some special

characters can be included in the data value with each value being 30 bytes long. Thus, if the actual data is less than 30 characters, the field's value will be padded to the right with spaces. In VarChar(50) on the other hand, the field's value is *at most* 50 characters long. If the actual value is only 37 characters, then only 37 characters are stored together with the actual length of the data field. Fields of these types can also be numbers, like Social Security numbers and telephone numbers, but these numbers are treated as character strings on which arithmetic computations are not performed.

A DateTime field can contain both a date and a time, or it can be simplified to being only a date or a time.

A Numeric field, naturally, contains numeric data and can be used in computations. Since Integer is available as a data type, Numeric is used when the values include digits to the right of a decimal point. Thus, Numeric(8,2) means that the number can be 8 digits in length, but 2 of them are to the right of the decimal. Currency or Money are numeric data types used to represent money amounts. Decimal is also available to represent values with fractional portions. Which one is chosen for a given field depends on the field and its use. Numeric fields are often floating point numbers for which storage and calculation is more efficient.

Null Values and Empty Fields

There is an ongoing discussion among database experts on the use of Null values (NULL) versus empty strings (two singles quotes ''). Unfortunately, there is not an agreement among these experts as to which is best to use or if both can be used in the same database. You are invited to research this topic on the internet.

For purposes of this class and the homework assignments, we will use NULL values when a field is empty. We will not use the empty string notation (''). You should insert NULL values for empty fields to ensure your homework is graded correctly.

You should note some characteristics of NULL fields:

- A NULL value can be applied to any data type.
- NULL values cannot be tested with the equal (=) sign. They must be tested by "____ IS NULL".
- Null fields do not return equivalence. "Where A = B" returns false if both fields are NULL.
- Fields that are NULL are not counted in the count (fieldname) function.

A list of ANSI standard data types for SQL is given at W3Resource.com. This site also lists data types from various commercial SQL products.

More generally, since there is a large variety of data types in the commercial SQL products, a description of all of them is beyond the scope of this book. However, you may find data type descriptions for Access, MySQL, Oracle, PostgreSQL, and SQL Server at:

Access

MySQL

Oracle

PostgreSQL

SQL Server

Explanation: Defining the Primary Key

Primary key definition for these two tables was straightforward. We simply included these statements:

Primary Key (CustomerID)

Primary Key (SaleID)

However, since the primary key in both cases was a single column, we could

have defined it when we listed the column. Thus, for example, we could have defined the primary key of Customer as:

Create Table Customer (

CustomerID Integer Not Null Primary Key, …

If the table's key has multiple columns—for example, as the SaleItem table has—then we *must* define the primary key separately from the column definitions. In that case, as you will see when we give the full schema definition below, the primary key is defined *after* the column definitions, and all columns that are part of the key are listed.

Explanation: Foreign Key Definition

In our example we have one foreign key that is defined as

Foreign Key (CustomerID) references Customer(CustomerID)

On Delete Set Null)

The first part of the definition is nearly identical to the foreign key definition of the original relational model. The definition says the CustomerID column in Sale references the key of the Customer table. But it also includes the name of the key column in Customer (CustomerID). The actual name of the key column is required in MySQL (as shown), but is optional in SQL Server, Oracle, and PostgreSQL. For clarity, we include it in all our foreign key definitions.

Moreover, in addition to identifying the foreign key column in Sale and the primary key column in Customer, this definition includes

"On Delete Set Null"

What can this mean?

Suppose a sale is made to customer 101—that is, to a customer with CustomerID 101. Thus, 101 will be the value of the CustomerID column in the record for that sale. Now suppose we delete the *Customer* record for customer 101. What happens to all the *Sale* records that have 101 as their CustomerID? If customer 101's record is deleted, then all those sale records have a foreign key that points to a record that doesn't exist. This is a violation of referential integrity that, as you will recall from Chapter 3, requires that every foreign key is either null or contains an actual key value in the referenced table.

The solution, in this case, is to set the CustomerID value in those sale records to null—or at least that is the instruction we are giving the system when we say:

"On Delete Set Null'

There are other possibilities for action if the customer record is deleted:

- Restrict
- Cascade
- Set Default

Restrict means the deletion of the Customer record is not allowed if there are any Sale records that reference that customer.

Cascade means that if the Customer record is deleted, then *all* Sale records referencing that Customer record will also be deleted. Cascade is a rather serious option since if there are records referencing the Sale records, which there would be, then those may have to be deleted as well. Therefore, the deletion of a single Customer record could result in the deletion of a considerable portion of the database itself.

Set Default means that if the Customer record is deleted, then the CustomerID in all referencing Sale records will be set to a default value—that is, they will be set to a value that references a particular Customer record, probably a "dummy" record, that does not stand for a real customer but rather is a catch-all set up to keep track of Sale records for customers no longer in the database.

Red Cat's SQL Schema

The following is the complete schema definition for the Red Cat Shoes database. For consistency, we have defined all primary keys as table constraints. Many fields are also defined as Not Null—meaning that null values are not allowed. In other words, data is required for those fields. Certain foreign keys are allowed to be null when their reference records are deleted.

Executing the Create Table Statements

In order to execute these Create Table Statements, you will need to use your own database namespace. These example queries have been configured to execute in your individual schema. Once you have created one of these tables, you will not be able to execute the "CREATE TABLE" statement again until you drop the table. However, you may not be able to just drop the table because other tables may have foreign key constraints that require the table to exist.

The first SQL block below shows a set of statements that will drop all of the tables that are created in subsequent code samples. If you execute it before you have created the tables, each "DROP TABLE" statement will fail. That's okay. It won't hurt anything. The error just means that there is not a table to drop.

Redcat Database Diagram

```
/*These statements only need be executed
 if you want to run the CREATE TABLE
 statements below more than once*/
DROP TABLE SaleItem;
DROP TABLE SalaryEmployee;
DROP TABLE WageEmployee;
DROP TABLE Employee;
DROP TABLE Product;
DROP TABLE Manufacturer;
DROP TABLE Sale;
```

```
DROP TABLE Customer;
```

Redcat Database Diagram

```
CREATE TABLE Customer (
         CustomerID INT NOT NULL,
         FirstName CHAR(30) NOT NULL,
         LastName CHAR(30) NOT NULL,
         StreetAddress VARCHAR(50) NOT NULL,
         City VARCHAR(50) NOT NULL,
         State CHAR(2) NOT NULL,
         PostalCode CHAR(6) NOT NULL,
         Country VARCHAR(50),
         Phone CHAR(15),
 PRIMARY KEY (CustomerID)
                        )
```

Redcat Database Diagram

```
CREATE TABLE Sale (
         SaleID INT NOT NULL,
         SaleDate DATE NOT NULL,
         Tax NUMERIC(8,2) NOT NULL,
         Shipping NUMERIC(8,2) NOT NULL,
         CustomerID INT,
PRIMARY KEY (SaleID),
FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID) ON DELETE SET NULl
                        )
```

Redcat Database Diagram

```
CREATE TABLE Manufacturer (
    ManufacturerID INT NOT NULL,
        ManufacturerName VARCHAR(255) NOT NULL,
        Address1 VARCHAR(255) NOT NULL,
        Address2 VARCHAR(255) NOT NULL,
        City VARCHAR(50) NOT NULL,
        State CHAR(20) NOT NULL,
        PostalCode CHAR(10) NOT NULL,
        Phone CHAR(15) NOT NULL,
        Fax CHAR(15) NOT NULL,
        Contact VARCHAR(255),
        URL VARCHAR(255),
 PRIMARY KEY (ManufacturerID)
                )
```

Redcat Database Diagram

```
CREATE TABLE Product (
         ProductID INT NOT NULL,
         ProductName VARCHAR(255) NOT NULL,
         Color VARCHAR(255) NOT NULL,
         ListPrice NUMERIC(8,2) NOT NULL,
```

```
        Gender CHAR(6) NOT NULL,
        Category VARCHAR(255) NOT NULL,
        ManufacturerID INT,
        Composition VARCHAR(255) NOT NULL,
        Description VARCHAR(255) NOT NULL,
PRIMARY KEY (ProductID),
FOREIGN KEY (ManufacturerID) REFERENCES Manufacturer(ManufacturerID) ON DEl
                        )
```

Redcat Database Diagram

```
CREATE TABLE Employee (
        EmployeeID INT NOT NULL,
        FirstName VARCHAR(255) NOT NULL,
        LastName VARCHAR(255) NOT NULL,
        Address VARCHAR(255) NOT NULL,
        City CHAR(25) NOT NULL,
        State CHAR(20) NOT NULL,
        PostalCode CHAR(15) NOT NULL,
        SSN CHAR(11) NOT NULL,
        Phone CHAR(15) NOT NULL,
        HireDate DATE NOT NULL,
        EmailAddress VARCHAR(50),
        ManagerID INT,
PRIMARY KEY (EmployeeID),
FOREIGN KEY (ManagerID) REFERENCES Employee(EmployeeID)
                        )
```

Redcat Database Diagram

```
CREATE TABLE WageEmployee (
        EmployeeID INT NOT NULL,
        Wage NUMERIC(8,2) NOT NULL,
        MaxHours NUMERIC(8,2) NOT NULL,
PRIMARY KEY (EmployeeID),
FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ON DELETE CASCADE
                        )
```

Redcat Database Diagram

```
CREATE TABLE SalaryEmployee (
        EmployeeID INT NOT NULL,
        Salary NUMERIC(8,2) NOT NULL,
PRIMARY KEY (EmployeeID),
FOREIGN KEY (EmployeeID) REFERENCES Employee(EmployeeID) ON DELETE CASCADE
                        )
```

Redcat Database Diagram

```
CREATE TABLE SaleItem (
        SaleID INT NOT NULL,
        ProductID INT NOT NULL,
        ItemSize NUMERIC(3,1) NOT NULL,
```

```
        Quantity INT NOT NULL,
        SalePrice NUMERIC(8,2) NOT NULL,
PRIMARY KEY (SaleID, ProductID, ItemSize),
FOREIGN KEY (SaleID) REFERENCES Sale(SaleID) ON DELETE CASCADE
                    )
```

You will note several of these foreign key's definitions include "On Delete Cascade." In every case, this is because the foreign key is itself part of the key of its table. In a relational database, no part of a key can be null, so if the referenced record is deleted, we cannot set the referencing foreign key to null. We could use Restrict or Set Default in these cases, but we've decided that if the user wishes to delete the referenced record, then the other records don't really make sense, so we've chosen to delete them (via Cascade) as well.

Other Schema Manipulation Statements

Besides the Create Table statement, which defines a new table, SQL provides other statements for changing the definitions of tables such as Alter Table or Drop Table. Alter Table can be used to add a column to a table, change the definition of an existing column, or drop a column from a table. Drop Table will delete all rows currently in the named table and will remove the entire definition of the table from the schema. Obviously, this operation should be used with great care and you will learn more about this in the next section.

Example for Alter Table:

Suppose we want to add CellPhone to the Employee table. We execute:

- Alter Table Employee
  - Add CellPhone VarChar(25);

We cannot indicate "Not Null" for this column, because the CellPhone for existing records will automatically be null. But we could set a default value if we chose.

We can also, among other operations:

- Drop a column
- Change the definition of a column
- Add or drop a primary key
- Add or drop a foreign key

For example, if we wanted to create a table, but forgot to give it a primary key, we could add a primary key with the following:

- Alter Table Customer
  - Add Primary key (CustomerID);

Or if we wanted to add a foreign key constraint to a table, we would use the following:

- Alter Table Product
  - Add Foreign Key (ManufacturerID) references Manufacturer (ManufacturerID)
  - On Delete Cascade;

6.3Insert, Update, and Delete Statements

Once the schema is completely defined, we need to populate the database and maintain values as circumstances change. We need to be able to enter new rows into tables, change values in existing rows, and delete rows that we no longer require. SQL provides three operations—Insert, Update, Delete—for adding, changing, and deleting data in the database. Each of these operations can affect only one table at a time. There are no updates to joins of tables. Bear this in mind as we discuss each of these three operations.

Copy to New Table

In addition to the standard Insert and Update discussed below, one powerful technique to load data into a database is by extracting the data from an existing table, creating a new table, and loading the data into the new table. The SQL language includes a statement that combines the following three actions into one statement:

1. A new table can be created based on the chosen columns from an existing table.
2. Data is extracted from the existing table.
3. The rows of data extracted from the existing table are inserted into the newly created table.

Normally when we use the SELECT statement, we just want to view the data on the screen, or sometimes to print it out to a report. However, this technique can be used to make a complete copy of a table, or to create a new table with only certain columns and populate it with selected data. The syntax of this statement is:

- **SELECT [* or (list of column names)] INTO new-table-name FROM table-name [(WHERE clause)]**

This statement works just like the previous SELECT statements you have already learned. It can select from a single table or from joined tables. The brackets indicate optional clauses. The only difference is that it inserts the results into a new table. As noted by the syntax, you can either select all the columns from the existing table, or you can select only a few columns. The data type information is also pulled over from the selected columns. You will use this statement to create some test data in your own private database. Once you have

some data in your own database, you can try the Insert, Update, and Delete statements that are taught next.

What are the dangers with change statements?
When using SELECT statements, since you are only viewing the data, you may run and rerun the queries as many times as desired. And if you mess it up, you only get an error. However, when working with change statements, you have to be careful because you can *really* mess up the database. And there is no UNDO statement in SQL. You could end up with undesired modifications and no way to recover the lost or modified data. You should always double-check your work, and consider ways to recover should the update query not do what you expect.

Another caveat is that you should not rerun your update queries. Sometimes rerunning your update queries might give you an error message. (Such as when you try to add a record with the same unique key that was already added.) Or you might make the change multiple times. (Such as when you increase a price by 10%. Running it multiple times would give you a compounded rate increase.)

Insert

The Insert operation has two versions. The first is used to enter a single row into a table. The second version inserts data from a separate table into the first table.

Insert: Listed Values Version

- ***INSERT INTO table-name [(list of column-names)] VALUES (list of values)***

The square brackets ([and]) are not part of the statement; they merely indicate optional entries. That is, the list of column-names is optional, but if it's included, it must be enclosed in parentheses. Likewise, the list of values is enclosed in

parentheses. If there is no list of columns, then the system will enter the values into the new row from left to right, starting with the first column. Here are some examples:

- Insert into Customer (CustomerID, FirstName, LastName, State)
    - Values (500, 'John', 'Smith', 'NM')

This statement will add a new row to the Customer table. Its CustomerID will be 500, and the first and last names will be John Smith, respectively. The State will be New Mexico, shown in the abbreviated form. All other column values, being omitted, will be null.

How do I specify particular fields to be empty?
In the last section you learned about NULL values and need for specific columns to never be empty. However sometimes there are columns that can be empty. A NULL value is assigned to a field when there is no value, in other words, when that field is empty.

Examples of this are in the Manufacturer table where we may have an Address 2 column. Most manufacturers don't have a value for this column, so it doesn't apply. Therefore, this column is null for those manufacturers. In the case of the customer, John Smith, whom we just added to the Customer table above, the street address, city, and other data that we didn't enter probably exist, but we don't yet know what they are. Because we don't know, we don't enter them, and the database places null values in these columns.

We could also write this Insert statement as follows, and the result would be the same:

- Insert into Customer (CustomerID, FirstName, LastName, StreetAddress, City, State, PostalCode, Country, Phone) Values (500, 'John', 'Smith', null, null, 'NM', null, null, null)

We can also omit the column list. But if we do, then we *must* state the null values in the value list:

- Insert into Customer Values (500, 'John', 'Smith', null, null, 'NM', null, 'USA', null)

Why isn't the keyword NULL in quotes? That is, why don't you write 'null', instead of null?
If we put null in quotes ('null'), then we would be entering a string of length four, and this would be considered an actual value instead of a null.

Let's practice using the Insert statement. We will do this in two steps. First, we will extract some data from the Redcatd Customer table and put it into your own private database. Next, we will execute the insert statement to add a new customer. We will look at the results after each step.

Redcat Database Diagram

```
DROP TABLE MyCustomer;
SELECT *
INTO MyCustomer
FROM redcatd.Customer
WHERE CustomerID &lt; 100;

SELECT *
FROM MyCustomer;
```

Notice that in this query box we have three SQL statements, and each statement is terminated with a semi-colon. The DROP statement will allow you to run this query multiple times. **The first time you execute it, you will get an error message, because there is no table to drop.** But each subsequent time you run it, it will first drop the existing MyCustomer table, and then recreate it. We also only extracted a few records so we can play with this table and see the results.

Redcat Database Diagram

```
INSERT Into MyCustomer(
```

```
        CustomerID, FirstName, LastName, StreetAddress, City, Sta
        Values (500, 'John', 'Smith', null, null, 'NM', null, nul

SELECT *
FROM MyCustomer;
```

We see the newly added record. Again, if you run these statements multiple times you will continue to add John Smith multiple times. You will learn in later lessons how to control this by setting up unique key constraints.

As a learning exercise for yourself, left click in the above code box and modify the code. Remove the list of column names and run the query using only the Values clause. Then rerun the query. You should see an additional John Smith record, or you can change the data and insert your own record.

Insert: Subquery Version

The second form of the Insert statement includes a query. We explain with an example. Suppose we want to archive our sale data by creating an archive table named SaleArchive. It would have identical structure to the Sale table, but would be used to store records of sales from the more-or-less distant past. To accomplish this, we would write an Insert statement as follows:

- Insert into SaleArchive
  - Select *
  - From Sale

    Where SaleDate < '2014-01-01'
  -

This statement would first perform the query indicated. It would then take its result set (consisting of every Sale record whose date was prior to the year 2014) and insert it into the SaleArchive table. The SaleArchive table must already exist, and in fact, may already have data in it. The SaleArchive table will now

consist of all the Sale records previously inserted as well as the ones now inserted.

The general form for this SQL statement is:

- ***Insert into table-name (select statement)***

Note that to use this type of insert statement that the data must be consistent in both tables. In other words, the number of columns, the order of the columns, and the data types of the columns must be the same for both tables.

What is the difference between SELECT...INTO and INSERT...INTO?
SELECT . . . INTO creates a new table from the schema of the FROM table.
INSERT . . . INTO requires that the target table already exist.

Update

The SQL Update statement is used to change the value of one or more columns in one or more records of a specified table. The generalized syntax is:

- **UPDATE table-name**
- **SET column1=value1,column2=value2,...**
- WHERE some_column=some_value;

For simplicity, suppose customer Raquel Lopez got married and changed her last name to West. Then this statement would make the necessary change:

- UPDATE Customer
- SET LastName = 'West'
- WHERE FirstName = 'Raquel' and LastName = 'Lopez'

What if more than one customer is named Raquel Lopez?

Oops! You're exactly right. If more than one customer is named Raquel Lopez, then all of them would now be named Raquel West in our Customer table. Obviously, we don't want that. So we have to be more precise. Instead of using her first and last name as the identifying characteristic, we use her CustomerID. Suppose Raquel's CustomerID is 30. Then this statement will work because CustomerID uniquely identifies a customer record.

- UPDATE Customer
- SET LastName = 'West'
- WHERE CustomerID = 30

Let's try it.

Redcat Database Diagram

```
UPDATE MyCustomer
SET LastName = 'West'
WHERE CustomerId = 30;

SELECT *
FROM MyCustomer;
```

As a practice exercise, why don't you update John Smith's record—the one you just added. Update his record and add a StreetAddress, a City, a PostalCode, a Country, and a Phone Number. Try it. You can do it with multiple statements, or all in one statement.

However, our mistake with Raquel points out that you can change the value of more than one record in the table with a single Update statement. So, suppose we want to reduce the ListPrice of all black shoes by 5%. Then we could do so with:

- UPDATE Product
  - Set ListPrice = ListPrice * .95
- WHERE Color = 'Black'

If we reran the SQL statement to update Raquel's last name, it would work just fine. It would just keep changing her last name to West. But what would happen if you reran the update to the Product table? That's a different story. Every time you reran it, the prices would be reduced by 5%. We repeat our emphasis on the potential harm that change statements can cause if not executed carefully.

Delete

We remove records with the Delete statement. Note the "from" keyword which is enclosed in brackets is optional.

- **DELETE [from] table-name**
- WHERE some_column = some_value

After we archive our Sale records with the multi-row INSERT statement we showed you above, we may want to remove the newly archived records from the Sale table since they're already backed up in another table, and we don't want the Sale table to have an unwieldy size.

- DELETE From Sale

  WHERE SaleDate < '2014-01-01'
-

Notice that we want the WHERE clause in this statement to be identical to the WHERE clause in the query of the Insert statement we used for archiving the Sale records. That way we delete exactly those records we had previously archived.

Obviously, we can delete an individual record as well by simply using the record's key in the WHERE clause. We would do this, for example, if we wanted to remove from the database a Manufacturer which has gone out of business.

- DELETE From Manufacturer
- WHERE ManufacturerID = 105

Let's practice the delete statement by deleting John Smith's record that we inserted earlier. In fact, if we reran that insert multiple times, this statement will delete them all.

Redcat Database Diagram

```
DELETE
FROM MyCustomer
WHERE CustomerID = 500;

SELECT *
FROM MyCustomer;
```

Caution!!!

We should note that if the WHERE clause is *omitted* from an UPDATE or DELETE statement, then every row in the named table will be updated or deleted respectively. Therefore, it is absolutely essential that you include a WHERE clause in these statements unless you intend to update or delete all the records.

CAUTION — DANGER
We emphasize this point. There is no UNDO function with database updates. When you update or delete data in a database, it is changed and the previous information is lost. You should always include a WHERE clause in an UPDATE or DELETE statement unless you are absolutely sure you want to affect every row. Of course, you should also be sure that the WHERE criteria are absolutely correct! Normally, before an actual update, the SQL statement should be verified to ensure that it affects the correct records. This can usually be done by testing it out as part of an SQL SELECT query.

6.4Knowledge Check

This chapter introduces the essential concepts and SQL language for defining database tables and populating and maintaining the data in those tables.

We began this chapter with the definition of tables within the database using Create Table statements, in which columns, keys, and foreign keys are defined. The columns in the tables are given their respective data types, and may also be defined as allowing or not allowing null values. These ideas are concretely illustrated through the complete definition of all database tables referenced in previous chapters. For each of the four DBMS platforms we support—SQL Server, Oracle, MySQL, and PostgreSQL—we provide a url link, through which the reader can connect directly to the description of the data types supported by the respective DBMS.

The Insert, Update, and Delete statements are also defined in Section 6.3 and illustrated with specific examples. Each of these statements work on only a single table at a time—you cannot update a join. The Insert statement can add a single row, with specific column values given, to its target table, or it can include a subquery on other database tables, which makes it possible to add multiple rows at once. The Update and Delete statements typically require a subquery on the target table, since all rows satisfying the subquery will be updated or deleted by the statement. For example, if there is no subquery with the Delete statement, then all rows will be deleted from the table. Since these are powerful statements, it's essential for the user to apply them with care. There is no "undo" statement in an SQL database!

Other Advanced Topics

This book's focus has been on the SQL SELECT statement which is the principal statement used in SQL for querying databases. We have covered in depth the features of the SELECT statement, and have developed the use of all six of its clauses. In this chapter you have also learned about statements of data manipulation, including how to create tables and modify the data in the tables.

Of course there are, of necessity, other SQL statements to maintain and administer databases. These additional statements provide additional capability to define the database schema, to modify data, to grant permissions, and to create and invoke triggers and stored procedures. The many database administration tasks having to do with backup and recovery, the security of data, increasing database performance, and so on, require a complete understanding of the using, maintaining, and protecting of databases. Finally, the vital topic of database design provides the basis for the proper development and use of well-functioning databases.

These topics are beyond the scope of this *Essentials Plus of SQL* book. We invite you to further explore these ideas and develop your own more thorough understanding of database concepts in our textbook *Database Management and Design*, also available from MyEducator.

This assessment can be taken online.

Appendix

7.1Connecting Directly to this Book's Databases

MyEducator hosts a Microsoft SQL Server database, MySQL database, PostgreSQL databse, and an Oracle database to support this book. You can connect to them directly as specified below. Be aware that you will need to install the appropriate drivers on your local computer. If you are using a Windows operating system, you already have the driver for SQL Server installed. If you want to connect to Oracle, you will likely need to install the oracle client.

Microsoft SQL Server

Host: myeducator.database.windows.net

Port: 1433

Database: userdata

Username: public_user

Password: Keep_on_truckin

MySQL

Host: oracle-sqlgrading.myed-eng.net

Port: 3306

Database: public

Username: public_user

Password: Keep_on_truckin

PostgreSQL

Host: oracle-sqlgrading.myed-eng.net

Port: 5432

Database: userdata

Username: public_user

Password: Keep_on_truckin

Oracle

Host: oracle-sqlgrading.myed-eng.net

SID: XE

Port: 1521

Username: public_user

Password: Keep_on_truckin

If you have installed Oracle's "SQL Plus" with the oracle client, the following command should get you connected:

sqlplus public_user/Keep_on_truckin@(DESCRIPTION=(ADDRESS_LIST=(ADDRESS=(PROTOCOL=TCP)(HOST=oracle-sqlgrading.myed-eng.net)(PORT=1521))(CONNECT_DATA=(SID=XE)(SERVER=DEDICATED))))

7.2Data Set: Chicago Food Establishment Inspections

This data set is provided as a resource for instructors to use for in-class demonstrations as well as for students to explore using SQL. These data report details of the inspections of food establishments and have been extracted from the city of Chicago's open data portal. The city of Chicago does not publish which employee conducted each inspection, so fictitious values have been generated for that data to provide a richer querying experience. Data about fines charged for violations are not published either so information about fines is also fictitious. However, the information about food establishments, inspections, and violations are real data.

Shown below is the data model for the Chicago Food Establishment Inspection data.

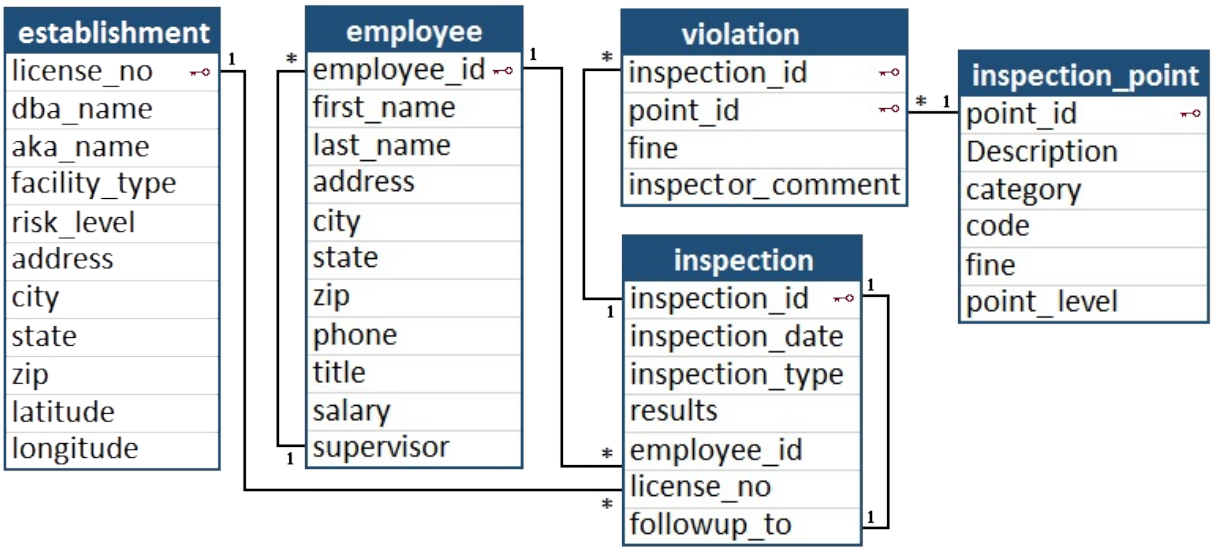

Figure 7.1: Chicago Food Establishment Inspection Data

Use the query box below to explore the data set or use the advanced query editor, using the "Food Inspection" credentials.

```
SELECT * FROM inspection_point;
SELECT * FROM employee;
```

7.3Data Set: Lorenzo Transit Corporation

This data set is provided as a resource for instructors to use for in-class demonstrations as well as for students to explore using SQL. These data are of a fictional transportation company, Lorenzo Transit Corporation (LTC). Customers hire LTC to move goods from their location to a specified city. LTC uses 18-wheel tractor-trailers and specializes in less-than-full loads. On each shipment, a particular driver uses a truck to make a delivery to the city specified by the customer.

Shown below is the database diagram for LTC.

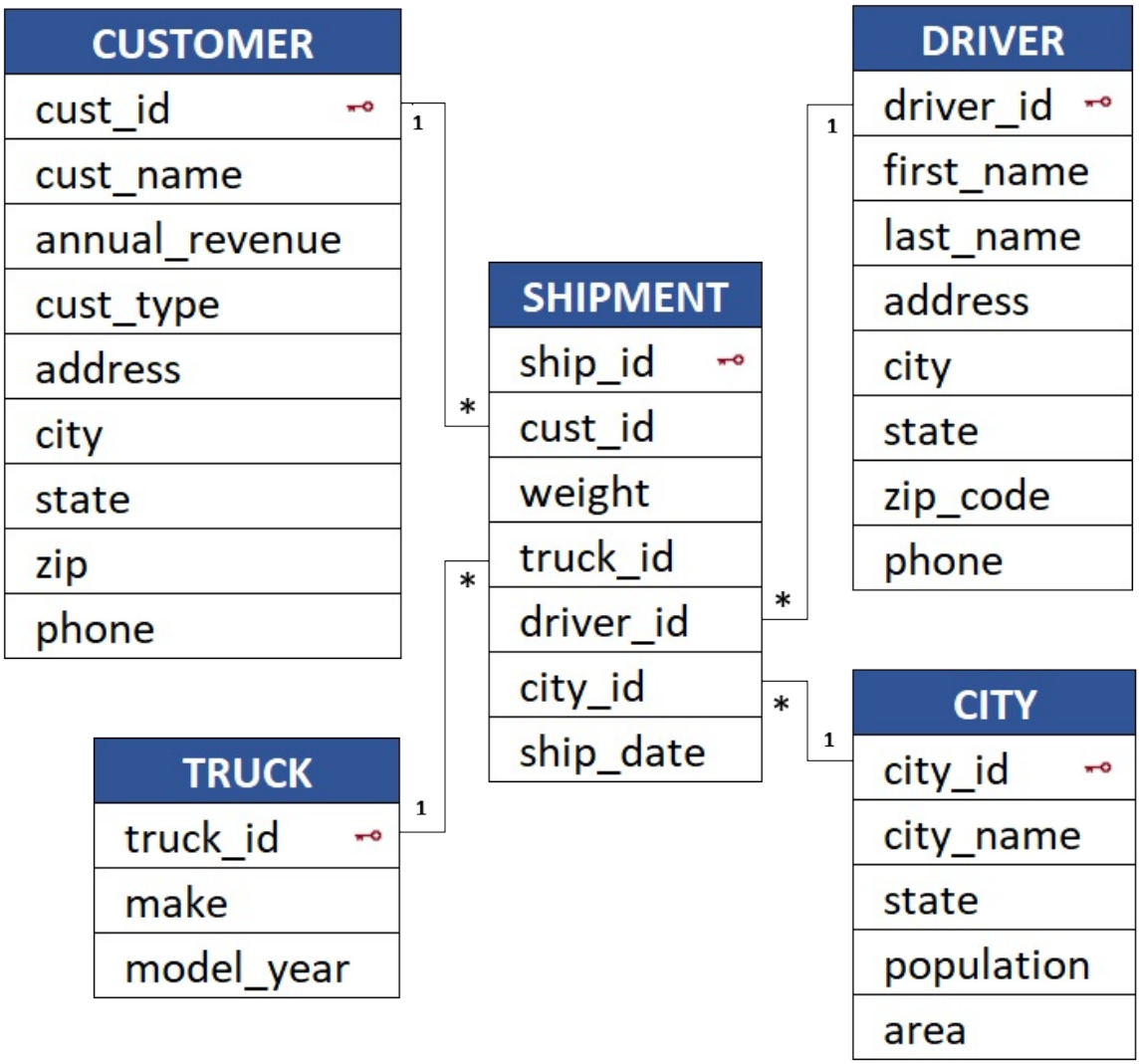

Figure 7.2: Lorenzo Transit Corporation Data

Use the query box below to explore the data set or use the advanced query editor.

```
SELECT * FROM truck;
SELECT * FROM driver;
```

Glossary

Aggregate Functions	Functions that work on multiple rows to get a single value by combining data across rows.
Aliases	Alternate names for tables, used to simplify the writing of a query
Attributes	Informational items that describe the properties of an entity.
Built-in Function	A function that is defined as part of the SQL language.
Correlated	A subquery whose execution changes according to the row being examined by the main query
Correlated Subqueries	Subqueries whose execution changes according to the row being examined by the main query
DISTINCT	Key word used in the SELECT clause to eliminate duplicate rows from result.
Database Management System (DBMS)	Database Management System (DBMS) is the software that manages and controls the data in a database.
Desc	Key word that says results should be in descending order for the specified sort key.
Entity	An entity is a specific type of object or type of thing.

FROM	Identifies which tables to query.
Foreign Key	A set of attributes in one table that is a key in another relation
GROUP BY	A SELECT statement clause which groups rows according to common values in one or more columns
HAVING	Identifies the conditions for the grouped data.
Join	To put rows from multiple tables together to answer a query.
Join Condition	Logical condition that matches a column in one table with a column in another table
Key	Attributes that are identifiers of records by having a unique value for each record
Main Query	Another name for outer query
NULL	A flag that indicates a value does not apply or is unknown.
Non-correlated	A subquery whose execution is independent of the main query containing it
Normalized	A schema that is structured in the simplest and most efficient form to eliminate redundancies and anomalies
ORDER BY	Identifies sorting criteria.
Outer Query	The initial query of which a subquery is a part
Recursive	A relationship within the same entity, which means that a record

Relationship	connects to another record within the same entity as opposed to a foreign key, which connects to a foreign entity.
Relationships	A connection or link between entitites.
SELECT	Identifies which columns to return through use of the query.
SQL Select Statement	The basic SQL statement used to extract data from tables
Scalar Functions	A function that operates on values from individual rows, one row at a time.
Schema	The description of the structure of a database including tables, attributes, keys, and foreign keys.
Selection Conditions	Logical conditions placed in the WHERE clause to select rows for the query results.
Structured Query Language	The standard language used to extract information from a relational database. Also used to update data in the database.
Subquery	A query within another query
Table	The basic data repository in a database consisting of rows and columns
WHERE	Identifies the conditions required for the returned data.
Wild Card Characters	Character symbols that stand for an uspecified string of standard characters such as letters or numbers