A Presentation of TeachUcomp Incorporated.
Copyright © TeachUcomp, Inc. 2014

Mastering Introductory SQL Made Easy™

TeachUcomp, Inc.®

...it's all about you

TeachUcomp, Inc.

Phone: (877) 925-8080
Web: http://www.teachucomp.com

Introduction and Overview

Welcome to TeachUcomp, Inc.'s Mastering Introductory SQL Made Easy™ tutorial. This tutorial is designed to give a complete overview of how to use SQL, or Structured Query Language, to create and manipulate databases.

SQL is a standardized programming language that is used to create, edit and delete databases and database objects. It is also the language that is used to extract, add, update, and delete data within a database. SQL is used in nearly every aspect of database interactions.

This course will provide the student with the "core," or essential, statements within SQL. Variations of the core statements for the specific database systems of MySQL 5.7, SQL Server 2012, and Access 2013 will also be referenced by hyperlinks to the associated online documentation for each system. The goal of this course to give the student the knowledge of which SQL statement they will need to use to accomplish specific tasks within a database, as well as provide links to database-specific implementations of those core statements.

Table of Contents

CHAPTER 1- Introduction to Databases and SQL

1.1- Overview of a Database:

SQL, an acronym that stands for Structured Query Language, is a standards-based language used within a relational database management system (RDBMS) A relational database management system is software that stores and manages data. Some relational database management systems are MySQL, SQL Server, Access, and SQLite, among others.

Before discussing SQL, you should understand the place of SQL within the system of relational databases. SQL is used within relational database management systems to create, edit, insert, query, update and delete data. In addition, SQL is used to manage data access within these databases. Therefore, before you can understand SQL, you must learn some basic concepts about relational databases. In this lesson, you will learn some terminology used in relational database design and implementation.

A database is an organized collection of stored data. The term "database" is often used in different ways by people with varying knowledge of relational database systems. Many times you may hear the term used interchangeably with the term "table." A database is actually a container or system that holds all of the tables, queries, and other objects within the relational database management system. A "table" is an organized structure within a database that holds data within its columns and rows.

You also need to understand the concept of a "relational database." In a relational database, you store large amounts of data into the smallest possible increments within tables. You then relate these tables by joining common fields between them. This way, you store less redundant data and your database operates quickly and efficiently. When you relate tables, you can access any data in the related tables.

The most fundamental object within a database is the **table**. A table is a collection of data about a certain subject: like customers, vendors or suppliers. It consists of columns and rows into which you store data. The columns all contain only one type of data, and are called "**fields**." For example, within a customer table you might have a "First_Name" field into which you place only the customer's first names. The rows in a table contain one set of related field information about a single entry, and are called "**records**." For example, in a customer table you may see a customer record that contains all of the field information about a single customer contained within one row.

Tables are the building blocks of almost every other type of database object. Tables contain all of the data that is to be stored, manipulated and retrieved within the relational database management system. Therefore, almost everything within a database is fundamentally dependent on the tables and their structure. So, while tables are often the database objects with which new users are most familiar, it is important not to approach table design haphazardly. Errors made during the creation and design of the tables will often cause problems in the functionality of the related objects, forcing you to go back and re-design or edit the tables as well as other related objects if you proceed with your database design too quickly. Creating well-designed data tables and joining them appropriately is one of the most difficult aspects of database design. It is also the most important aspect of database design.

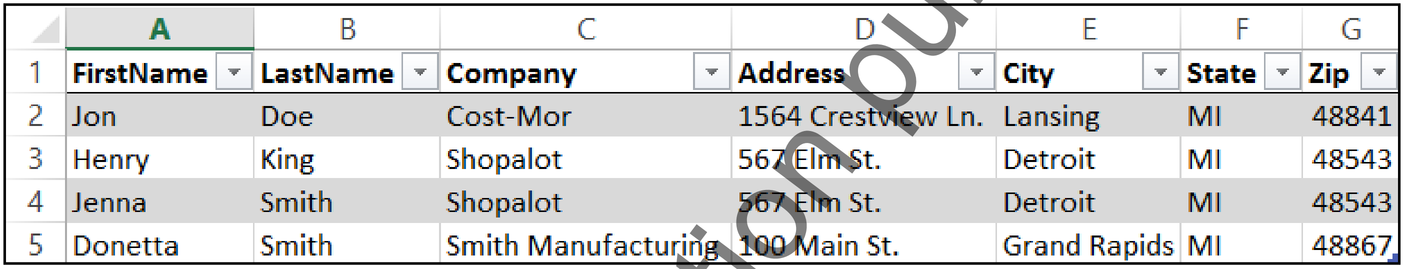
The next type of database object to discuss is the **query**. The purpose of a query is to extract only the data that you want or need to view from the tables. These objects are the "heart" of database design-and the whole point of using databases. The queries provide the data that is needed by the other database objects, often working in the background. So mastering queries will also be an important part of creating a functional database. SQL is the language used to create all of the objects within relational database management systems.

A database should be simple, logical, and straightforward in its design. In general, data is entered into **tables**. The data is stored within these tables, which are *related* to each other as necessary. You use **queries** to extract specific information from the **tables** in the database. The **queries** often form the basis for **view** and **reports,** which allow you to view the information requested. *This is the main reason that you use databases:* ***to enter, store, and retrieve data.***

1.2- The 'Flat-File' Method of Data Storage:

In the previous lesson, the term *relational* database was used. So what does the term *relational* mean, and how is this important? The term *relational* describes the method used for storing data within the database tables. However, it may be easier to understand the relational model of data storage by contrasting it with another method of storage that you may be more familiar with: the 'flat-file' method.

Information is frequently stored in large 'flat-files.' For example, assume that you want to create a database file that stores your company's customer information. You would begin by listing the different *attributes* of the customer that you wish to record. You may want to record customer information like the "first name," the "last name," the "company name," and other relevant pieces of information. Perhaps you could create a table in an application like Microsoft Excel where you can create columns for each piece of information that you wish to record. You can then list each customer's information in the rows underneath the columns, creating a basic table. Assume it looks like the following example.

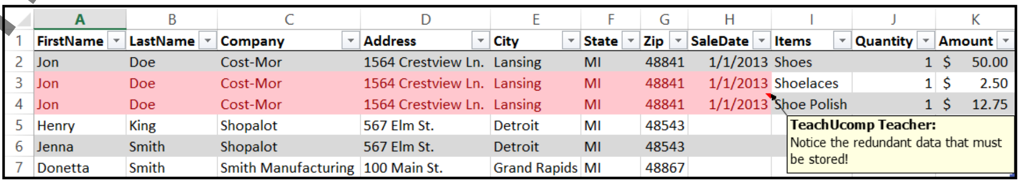


	A	B	C	D	E	F	G
1	FirstName	LastName	Company	Address	City	State	Zip
2	Jon	Doe	Cost-Mor	1564 Crestview Ln.	Lansing	MI	48841
3	Henry	King	Shopalot	567 Elm St.	Detroit	MI	48543
4	Jenna	Smith	Shopalot	567 Elm St.	Detroit	MI	48543
5	Donetta	Smith	Smith Manufacturing	100 Main St.	Grand Rapids	MI	48867

For many types of databases, the structure shown would work well. This is a 'flat-file' list or table. What you are doing when using this type of database is recording a single piece of information, like the "FirstName," "LastName," or "Address," about a single entity- in this example, a customer. The reason that this type of data structure works well in the example given is because for each entity (the customer), you are only recording information that has a "1 to 1" relationship to the entity.

So, what does this "1 to 1" relationship between the entity (the customer) and the data you are recording ("FirstName," "LastName," etc.) mean? What this means is that for each entity or subject (in this case- the customer), you are only recording information about that entity for which there would only be one "answer." For example, each customer would only have one "first name" and one "last name." They would work for only one "company." So the term "1 to 1" refers to the relationship between the subject of the table (customers) and the data being collected about the entities. Because for each (one) customer, there is only one possible piece of data to record in the column, the relationship between the data and the entity is "1 to 1." If this is the type of database that you are trying to create, simple Microsoft Excel tables will work well.

The problem occurs when you try to use a 'flat-file' approach to model a more complex entity or subject, like "sales." For example, assume you wanted to expand the customer database from the last 'flat-file' database to include sales data. Now, in addition to the information already being collected, you also want to record each customer sale. First, you would start by listing what data about each sale that you want to record. Keeping the example simple, assume you decide to record the "sale date," the "items" purchased, the "quantity" of items purchased, and the "amount" paid for each item. You may decide to add the following columns to the 'flat-file' data structure.



	A	B	C	D	E	F	G	H	I	J	K
1	FirstName	LastName	Company	Address	City	State	Zip	SaleDate	Items	Quantity	Amount
2	Jon	Doe	Cost-Mor	1564 Crestview Ln.	Lansing	MI	48841	1/1/2013	Shoes	1	$ 50.00
3	Jon	Doe	Cost-Mor	1564 Crestview Ln.	Lansing	MI	48841	1/1/2013	Shoelaces	1	$ 2.50
4	Jon	Doe	Cost-Mor	1564 Crestview Ln.	Lansing	MI	48841	1/1/2013	Shoe Polish	1	$ 12.75
5	Henry	King	Shopalot	567 Elm St.	Detroit	MI	48543				
6	Jenna	Smith	Shopalot	567 Elm St.	Detroit	MI	48543				
7	Donetta	Smith	Smith Manufacturing	100 Main St.	Grand Rapids	MI	48867				

TeachUcomp Teacher:
Notice the redundant data that must be stored!

1.2- The 'Flat-File' Method of Data Storage- (cont'd.):

This may appear to work, at first glance. However, you will immediately begin to encounter problems when you begin to enter records into the file. To begin with, each time a customer makes a purchase, you must re-enter all of the "FirstName," "LastName," etc. information again. This alone is irritating enough.

However, you will also soon run into another problem: What do you do when a customer purchases multiple items in an order? One solution often proposed at this point is to enter another row (with all of the redundant information) for each item purchased. However, you will find that this file will grow quite quickly down the table, and you will also have to enter a lot of redundant customer data for each item purchased. This is not an elegant solution and will inevitably waste data storage space as well as the time and effort of the person who performs data entry.

Another solution often proposed at this point is to create additional columns (like "Item1," "Item2," "Item3," "Quantity1," "Quantity2," Quantity3," etc.) instead of having to enter additional rows of information. While this may seem like a good alternate solution, what will you do when someone purchases 100 items? Will you really create a set of 3 columns ("Item," "Quantity," "Amount") for each item purchased, producing a table over 300 columns across? Would you simply leave them blank if the person orders only 1 item, wasting valuable storage space? In this solution, you are simply substituting columnar growth (across) for vertical growth (down). This is not an elegant solution either.

So why is there a problem now, when there wasn't one earlier? The answer is that now you are no longer trying to model a "1 to 1" data relationship in the table. Recording sales information is simply more complex than recording customer information. What you are trying to record now is what is referred to as a "1 to many" relationship. Basically, for each entity (the customer), you are now trying to record data in the columns which could occur more than once per customer (the "Items" ordered). You would be in a sorry state if each customer could only purchase a single item. You must allow for the fact that in a sale, each customer may order *many* items. The relationship between customers and items purchased is a "1 to many" relationship. When you find that you are trying to model a "1 to many" relationship, it is then that you must abandon the 'flat-file' method of data storage where you try to place all of the information that you want to record into a single table, and instead turn to the relational model of data storage for the solution.

1.3- The Relational Model of Data Storage:

The relational model of data storage allows you to more easily and effectively model a complex entity or subject, like sales. The relational model of data storage eliminates redundant data entry and also creates less data to store, making the relational database model smaller and faster than the 'flat-file.'

When you create a relational database, you will first need to perform some *data modeling*. Data modeling allows you to ensure that you are recording all of the information needed, and also helps you identify the entities involved and their relationships to each other.

When you create a relational database, you need to identify the unique entities involved in the process that you are modeling. These "entities" will often become the various tables in your database. So, for example, in the sales database example from the last lesson, the "Customer" is an entity. Within each table created for each entity, you must *only* list fields, or columns, of information which share a "1 to 1" relationship with the entity, or subject, of the table. So for example, in a "Customer" table, you would want to place the field "FirstName," assuming that each customer only has a single "FirstName" to record. You would **not** want to place "Item" in the "Customer" table, as the relationship between the customer and the items purchased is "1 to many." So what would one do with the column of "Item?" In the relational model, *each field (column) of information is an attribute of an entity*. For what entity is "Item" an attribute? In other words, with what "entity" does the "Item" (a description of the item purchased) have a "1 to 1" relationship?

1.3- The Relational Model of Data Storage- (cont'd.):

Perhaps you may initially think that the “Item” is an attribute of the “Sale.” However, could you have a single “Sale” with multiple “Items” ordered? Probably so. In that case, it must be an attribute of something else. In this case “Item” is probably going to be an attribute of an “Item” entity; meaning that you will probably need to create an “Item” table.

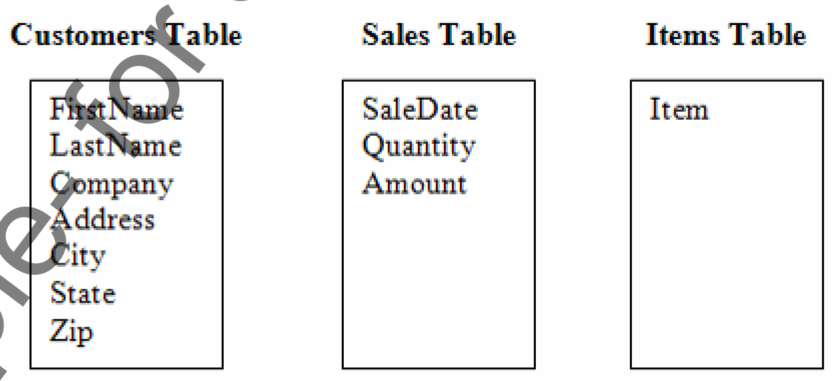
Many times, when initially approaching data modeling, it may be easier to list the various attributes that you wish to record, and then try to find what “entities” the attributes describe. The “entities” will become the various tables in your database. The “attributes” will become columns within the entity tables. Remember that each attribute (column) in your table must share a “1 to 1” relationship with the “subject” of the table (the entity).

In either case, you should probably keep your information written down on paper until you have a rough idea of what information it is that you want to record about the various entities involved with the process or system which you are trying to model. It is a rare feat to have your preliminary sketch of the relational database tables turn out to be the finished model that you will actually create in your RDBMS. Many times you will need to create a model, look for problems with the model you have created, and then edit the design until you are ready to attempt creating the tables.

Let’s take a look at a preliminary model of the “sales” database from the prior example. First, make a listing of the various pieces of information that you want to record. These become the attributes of the various entities. Next, try to find what entities these attributes describe and list those too.

Attribute:	**Belongs to Entity:**		**Attribute:**	**Belongs to Entity:**		**Attribute:**	**Belongs to Entity:**
FirstName	Customers		City	Customers		Items	Items
LastName	Customers		State	Customers		Quantity	Sales
Company	Customers		Zip	Customers		Amount	Sales
Address	Customers		SaleDate	Sales			

Next, make some sketches of the tables that show the fields of information within them. This can help you start to visualize what tables you will need to create, and will also allow you to see how the tables will eventually be *related* to each other in a larger, relational database structure.

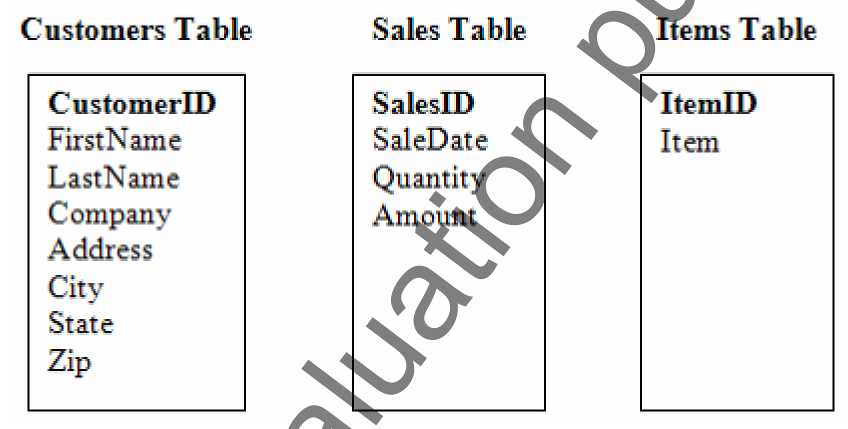


Once you have a rough idea of what you would like to record and what tables you will need to record the information, you must then ensure that each table has what is called a “primary key.” A **primary key** is a column, or combination of columns, that will produce a unique value for each **record**, or row, in a table. Many times, an additional column is added to the tables in order to provide this unique identification. You can assign each record a unique number in an “ID” column. For example, that is what your social

1.3- The Relational Model of Data Storage- (cont'd.):

security number is used for by the government. You also have a unique driver's license number, as well. If you were recording any of these pieces of information, you could use those as the "primary key" in the table. If, however, you aren't recording any type of unique information, then often you must assign your own unique values. Many companies, for example, assign "Customer ID" numbers in order to uniquely identify each customer. Let's look at how your data model will change once you assign "primary keys" to your tables.

For example, you need a way to uniquely identify each customer. In the current data model, there isn't any kind of information that would enable you to uniquely identify each record (row) within the "Customers" table. So you could add an additional field (column) of information to this table: "CustomerID." Assume that you then add another column for "SalesID" to the "Sales" table, and an "ItemID" field to the "Items" table. In the sketch below, each "primary key" field is shown in bold within each table diagram. So, the data model would look something like this:



The "primary key" is a very important concept in a relational database, because it is through the primary key assignment that you create the necessary relationships between the data tables. For example, examine the relationship between the "Customers" table and the "Sales" table in terms of the "1 to 1" and "1 to many" relationship. In this case, for each (1) customer there can potentially be many sales. So, the tables will share a "1 to many" relationship. This is the most common type of relationship between tables, with extremely few exceptions. What you need to do next is find a way to join the "many" side of this relationship to the "one" side of the relationship. You need to relate each entry in the "Sales" table to a customer in the "Customers" table.

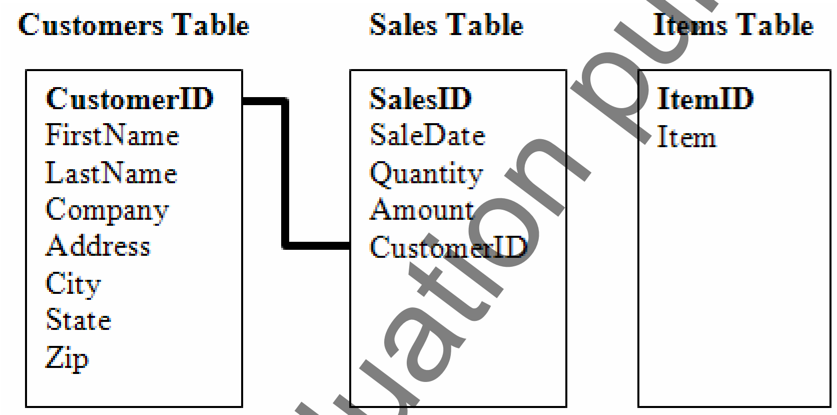
In order to join tables, they must have a shared, or common, field between them. This would be a field that contains the same kind of data in both tables. So, in this example, you are trying to assign each sale to a customer. To do this, you would want to add a field to the "Sales" table that corresponds to a field in the "Customers" table. Which field would you choose? The answer is: the "primary key" field!

Remember that each primary key field is designed to uniquely identify each record in the table, so you can add a field to the "Sales" table that will make a reference to the values in the "CustomerID" field of the "Customers" table. That way, when you enter a record into the "Sales" table in the future, you will only need to enter the "CustomerID" number of the customer to whom the sale was made, practically eliminating redundant data entry! So you can see one advantage of the relational model. In this model, you only have to enter the customer's data once in the "Customers" table, and then assign them a unique "CustomerID." When you then enter sales for that customer into the "Sales" table, you only need to make a reference to the appropriate "CustomerID" in the "Sales" record to indicate who made the purchase! This allows you to

1.3- The Relational Model of Data Storage- (cont'd.):

store much less redundant data, making the relational tables smaller and faster to use than the 'flat-file" table. It is also important to note that the "CustomerID" field which is added to the "Sales" table is **not** a primary key! That table already has a "primary key" field in the "SalesID" field, which uniquely identifies each sale- like a receipt number. Technically, the field in the "many" table which makes a reference back to the primary key in the "one" table is called a "foreign key." It's only purpose is to relate the two tables, and the values within a foreign key are almost always non-unique within the column.

Don't worry about the mechanics of the data entry, or how to create primary keys and table joins just yet. It will explained in later lessons. For now, just try to comprehend the concepts and reasoning behind the relational database design. Let's examine how the table diagram has changed to reflect the newly created relationship between "Customers" and "Sales."



Next, you will want to examine the other relationships between the tables. For example, what is the relationship between "Customers" and "Items?" Don't be hasty- not every table in the database has to be directly related to every other table. The only way that customers and items are related is that the customer purchases the items when making a sale. The "Customers" and "Items" do not have a direct connection. However, in a relational database, as long as every table is connected in an appropriate manner to the correct table, you can find out how they are related to each other through the tables by which they are connected. In summary, the "Customers" are connected to the "Items," but only through the "Sales."

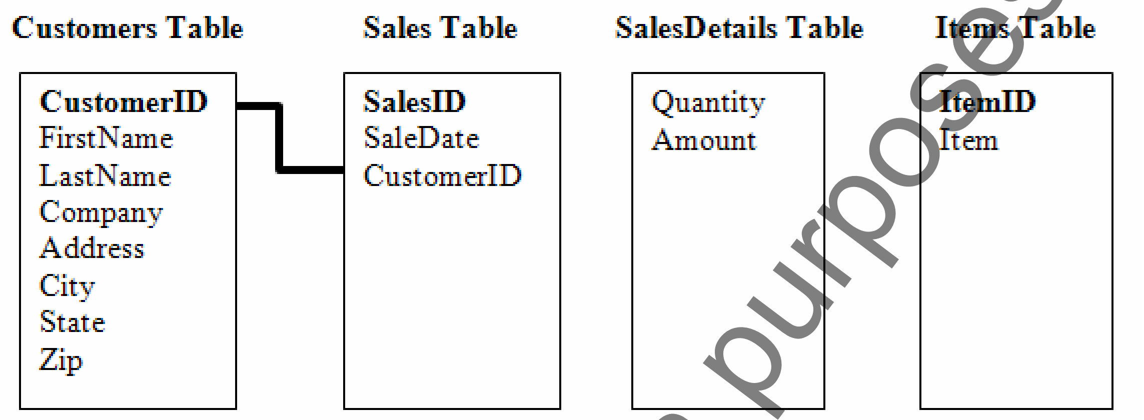
So, how are the "Sales" table and the "Items" table connected? Well, for each sale, there may be many items ordered. Also, each item may appear in more than one sale! In relational database design, you cannot (or **should** not) create a "many to many" relationship. That would make no sense from a strictly logical point of view. You need to be able to tell which items were ordered in which sale, while reducing the amount of data entry. Also, you may notice another problem with the current data model- the "Amount" field is attached to the "Sales" table. In this context- this field would be the "SalesTotal." If that is the case, then how can you record the price of each item at the time of sale? What if the price of each item changes in the future? Is the "Amount" also an attribute of the item?

What you are starting to see is that you need to be able to link the unique sales records to the unique items ordered on each sale. You need a "SalesDetails" table in order to do this. But what fields do you place into the new "SalesDetails" table? The answer is: anything that is an aspect of the "many" side of the "Sales" transaction. For example, the "SaleDate" field can stay in the "Sales" table because each sale only happens on a specific date. The "Quantity" of the items purchased at the time of sale is actually part of the "many" aspect of the sale and should be moved to the new "SalesDetails" table, along with the "Amount" field. The "CustomerID" will stay tied to the "Sales" table, as each purchase is made by a single customer.

1.3- The Relational Model of Data Storage- (cont'd.):

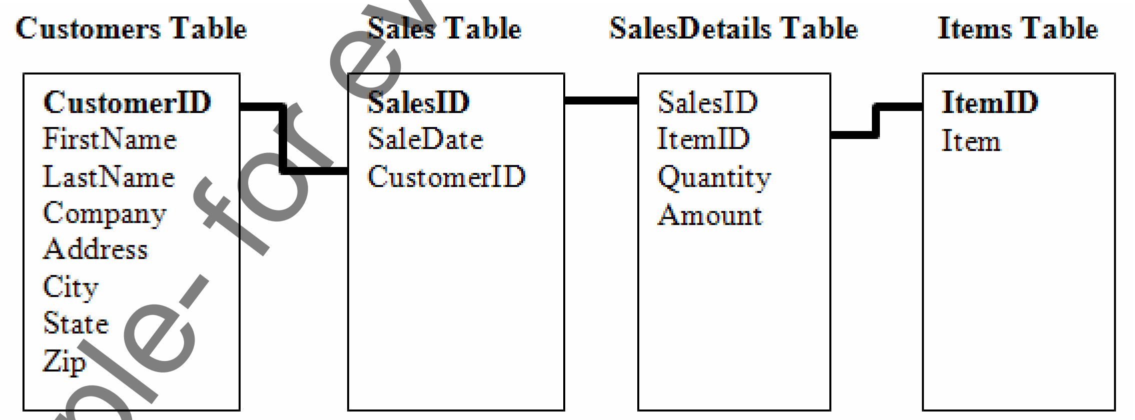
So now examine how this new "SalesDetails" table will affect the data model.

Below is a diagram of the new tables in the data model. You must also remember that the new table of "SalesDetails" will also need to contain a "primary key" field.



Before assigning the "primary key" field, look at how you will relate the "Sales" table to the "SalesDetails" table. The tables are related in that each sale may have one or more items purchased in each sale. So, you need to join each record in the "SalesDetails" table to the "Sale" record to which it corresponds. To do this, you will add a "foreign key" into the "SalesDetails" table that corresponds to the data in the "primary key" in the "Sales" table. So, you will add the "SalesID" field to the "SalesDetails" table.

Next, examine the relationship between the "Items" table and the "Sales Details" table. In this case, for each item ordered in a transaction shown in the "SalesDetails" table, it must make a reference to a unique item in the "Items" table. So, you will add the foreign key field of "ItemID" to the "SalesDetails" table. Then you can "join" the tables through their "shared" or common fields. Examine how the data table diagrams in the data model may look after performing these two tasks.

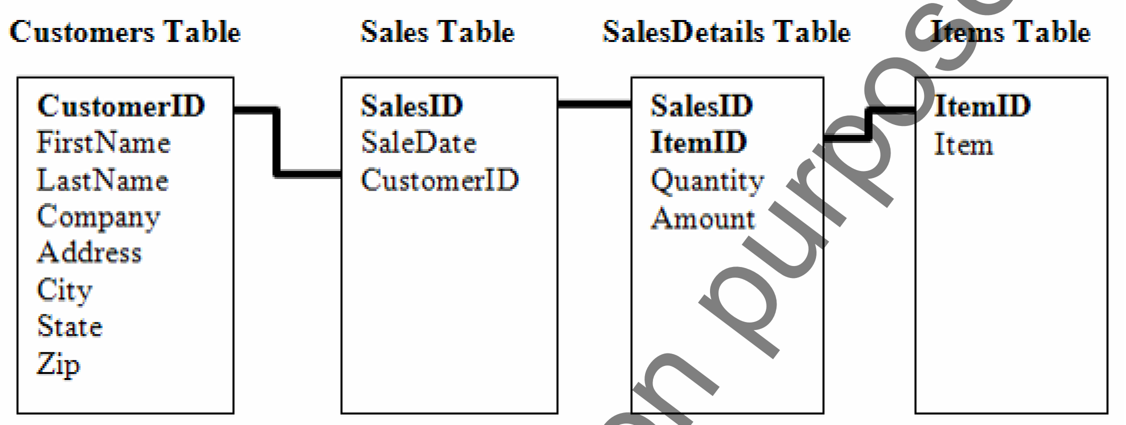


Now you have created all of the necessary relationships between the tables. However, the "SalesDetails" table is still missing a "primary key" field. You could add another field as the primary key, such as "SalesDetailID." However, you could also see if there is a combination of fields that already exists that, when combined, produce a unique row value. In fact there is: the combination of "SalesID" plus "ItemID." There should never be a repeating combined value in those two columns. If there were, it would mean that the same item was recorded in the same order twice. If that were the case, it should only be recorded once in the "SalesDetails" table with a "2" into the "Quantity" field. So, assuming that you make

1.3- The Relational Model of Data Storage- (cont'd.):

this combination of fields the primary key for the table, let’s examine the data diagram.

This is the final data diagram based on the information that was needed to record the sales. Obviously, there is more information that could be recorded, but this example is only supposed to illustrate some of the decisions that should go into table design before you begin to create tables in a relational database management system (RDBMS).



1.4- Tips for Creating a Relational Database:

While there are no “hard and fast” rules about creating relational database tables, there are a few tips that you should try to follow when beginning database design. First, examine all current documentation used to collect and store the information that you now want to store in the new database. This step ensures that when you are creating your data tables and performing your data modeling, you won’t leave out a critical part of your database. Doing that often leads to frustrating periods of re-design. Also, consider what the database will need to contain in terms of the views that you need to design. Also consider the need of the users who will want to run reports and perform data entry. You should gather information from those users who need to use the database that you create.

Next, use the entity/attribute and relationship modeling that was discussed in the previous lesson. This is a helpful first step in discovering how your tables should be structured.

When performing data modeling, you may want to start by listing the entities, or “subjects” of the tables, in the database along with their properties or attributes that you want or need to record. You may also find it easier to begin by listing the attributes and then trying to discover to which entities the attributes refer. Once you have accomplished this part, sketch the entities as tables and find or create the primary keys needed for each table. Sketch relationships between the tables and list the type of relationship that the tables share. About 99% of the time, this will be a “1 to many” relationship.

After you have a preliminary table sketch, you can then turn to “normalization” guidelines to assist you in analyzing the database structure for its relational “soundness” of design. These guidelines were created to assist the relational database designer in creating sound relational structures that do not break any of the foundational tenets of relational database design. While these are not “rules” per se, you shouldn’t violate one of the normalization guidelines without having a very good reason for doing so. If you decide to do so, document your reasoning for making such a break. When a relational database follows one of the normalization guidelines, it is said to follow the “form” of the guideline.

While there have actually been many normalization guidelines proposed, many database designers find it is adequate to design their relational databases to satisfy the normalization guidelines through the

1.4- Tips for Creating a Relational Database- (cont'd.):

third or fourth normal “forms.”

The first normal form requires atomic, or unique, values at each column and row intersection in the entity table. There should be no repeating groups. Thus, no “Item1,” “Item2” column design like you may see in a ‘flat-file’ table layout.

Second normal form requires that every “non-key” column in a table must depend on the primary key. A table must also not contain a “non-key” column that pertains to only part of a composite, or multi-column, primary key.

The third normal form requires that no “non-key” column should depend on another “non-key” column. This is very similar to the second normal form. You shouldn’t have a field that is an attribute of a non-primary key column in a table.

Fourth normal form forbids independent “1-to-many” relationships between primary key columns and non-key columns.

As you begin your modeling your database tables, be sure to document your work as you create your initial designs. Correct violations of normal form that you see, or make conscious decisions to override them. Always document why you chose to make the changes that you do make. After you create your basic tables and relationships, review your design. Then create the database tables and enter some preliminary or “test” data to see if your design works or how well it works. Reevaluate your design and fix flaws as required. Always document the reasons that you decide to change the table design.

1.5- What is SQL?:

SQL, pronounced as the individual letters “S” “Q” “L” or often “sequel,” is the language used to manage and interact with all relational database management systems. While the specific implementation of the language may vary from vendor to vendor within database management software, SQL is designed as a standardized language that is supposed to implement some basic functionality within these programs. The standard functions include running queries on data, modifying the data within the tables, displaying views of the data, and creating and modifying database structures and data access.

Since the purpose of using SQL is fairly narrowly defined, you may find that SQL seems very simple when compared to other programming languages. SQL, as a “standard” language, is jointly governed by the ISO (International Organization for Standardization) and the IEC (International Electrotechnical Commission). However, most vendors have included their own vendor-specific extensions to the language. Therefore, you will often encounter vendor-specific variations to the standard SQL when it is encountered in real-world applications. These vendor-specific implementations will often have their own names, such as the “Transact-SQL” that is used within Microsoft SQL Server, for example. The “core” SQL, as defined by the IEC and ISO, is mandatory and supported by all of the major database management systems, regardless of their own specific implementations and extensions. This course will most often refer to the core SQL within its examples, but will also reference variations used by specific implementations within some database management systems, when needed. The core SQL statements you will learn, which have not changed very much since 1999, will allow you to accomplish almost everything necessary to create and manage a database. Also, most RDBMS implementations, such as SQL Server and Microsoft Access, will require you to add a semicolon to the end of any executable SQL statements you create. While this is a standard that has been widely implemented, note that the examples of the core SQL shown in this course will not show a semicolon at the end of the examples shown, as they are simply demonstrating the core SQL that can be used and are not *literal* examples of SQL for any specific relational database management system, unless otherwise specified by the example.

1.5- What is SQL?- (cont'd.):

The statements within standard SQL can be thought of as being grouped into four categories: Data Definition (alt. Description) Language (DDL), Data Manipulation Language (DML), Data Control Language (DCL) and Transactional Control Language (TCL). The statements within the Data Definition Language, such as CREATE, ALTER, and DROP, allow you to create the database container, tables, and other objects. The statements within the Data Manipulation Language, such as SELECT, INSERT, and UPDATE, allow you to manage the data contained within the tables. The Data Control Language statements, such as GRANT and REVOKE, allow you to determine who can access data within the database and set referential integrity. The Transactional Control Language statements, such as COMMIT and ROLLBACK, manage the actual transactions that occur within a database. The SQL statements most often thought of by new SQL users are contained within the DML statement set. This is where you will find the SELECT statement, which is used to create queries on the data, as well as the INSERT, UPDATE and DELETE statements used to add, edit and remove data from tables.

The most current revision of the ANSI standard of SQL, as of this writing, is the SQL:2011 version. This is the seventh revision of the SQL standard. The most interesting, but optional, new feature of this revision is that it includes new standards for support and implementation of temporal databases. A “temporal database” is a database that supports extensive data handling involving time. This is contrasted with most databases, which are referred to as “current databases,” in that the data contained within them is assumed to be true currently, until the data is deleted or updated. In a temporal database, however, each record is often qualified with a valid date/time stamp, valid date/time period, or valid date/time interval. This can then indicate the time duration of when that information was deemed to be “valid,” or true. Note that most users will not be dealing with temporal databases and vendor implementation of this new standard is still not widespread as of 2014, although both IBM DB2 and Oracle have implemented queries that address this issue with their respective “Time Traveler Queries” and “Flashback Queries.”

1.6- Using SQL in Access 2013:

This tutorial references using SQL in Access 2013. Access 2013 is a RDBMS that creates self-contained databases and provides visual tools to allow users to make relational databases without the need for SQL. As such, there are few places to use SQL in Access. You can enter SQL into the “SQL View” of a query when creating a query in Access. You can also enter SQL into modules you design or into any “Code Builder” attached to form objects within a database. Note, however, that Access may not interpret these SQL statements in Access 2013, unless you enable ANSI-92 compliance within the current database file.

To do this, create a new database file within which you want to enter SQL statements in Access 2013. Click the “File” tab within the Ribbon and then click the “Options” button at the left side of the backstage view to open the “Access Options” window. Click the “Object Designers” category at the left side of the “Access Options” window to display the category options to the right. Under the “Query design” header, check the checkbox for “This database” under the “SQL Server Compatible Syntax (ANSI 92)” section. Then click the “OK” button within the “Access Options” window. Access will display a message onscreen telling you it will need to close, convert, and re-open the current database to apply this change. Click the “OK” button within the message to continue and convert the database.

You can then enter the SQL commands within this tutorial within the “SQL View” of the query design window as well as within modules or any “Code Builder” areas associated with form objects that you create. You can also use the database to connect to external data within SQL Server to execute SQL statements by using SQL within the Access database.

ACTIONS- Introduction to Databases and SQL

DATABASE OBJECTS:

A **table** is a collection of data about a certain subject: like customers, vendors or suppliers. It consists of columns and rows into which you store data.

A **field** is a column within a table. Fields all contain only one type of data. For example, within a customer table you might have a "First_Name" field into which you place only the customer's first names.

A **record** is a row within a table. A record contains one set of related field information about a single entry. For example, in a "Customer" table you may see a customer record that contains all of the field information about a single customer contained within one row.

A **query** is used to extract only the data that you want or need to view from the tables. These objects are the "heart" of database design- and the whole point of using databases. Queries provide the data that is needed by the other database objects, often working in the background.

SQL is the language used to create the objects that exist within relational database management systems.

A **database** is the entire collection of tables, queries and other related objects.

A **relational database management system (RDBMS)** is software used to create and maintain a database.

THE PURPOSE OF A DATABASE:

A database should be simple, logical, and straightforward in its design. In general, data is entered into **tables**. The data is stored within these tables, which are *related* to each other as necessary. You use **queries** to extract specific information from the **tables** in the database. The **queries** often form the basis for **view** and **reports,** which allow you to view the information requested. *This is the main reason that you use databases:* ***to enter, store, and retrieve data.***

THE FIRST 4 FORMS OF NORMALIZATION:

First normal form requires atomic, or unique, values at each column and row intersection in the entity table. There should be no repeating groups. Thus, no "Item1," "Item2" column design like you may see in a 'flat-file' table layout.

Second normal form requires that every "non-key" column in a table must depend on the primary key. A table must also not contain a "non-key" column that pertains to only part of a composite, or multi-column, primary key.

Third normal form requires that no "non-key" column should depend on another "non-key" column. This is very similar to the second normal form. You shouldn't have a field that is an attribute of a non-primary key column in a table.

Fourth normal form forbids independent "1-to-many" relationships between primary key columns and non-key columns.

TO CONVERT AN ACCESS 2013 DATABASE TO ALLOW FOR ANSI 92 SQL STATEMENTS:

1. Create a new database file within which you want to enter SQL statements in Access 2013.
2. Click the “File” tab within the Ribbon.
3. Click the “Options” button at the left side of the backstage view to open the “Access Options” window.
4. Click the “Object Designers” category at the left side of the “Access Options” window to display the category options in the area to the right.
5. Under the “Query design” header, you can check the checkbox for “This database” under the “SQL Server Compatible Syntax (ANSI 92)” section.
6. Click the “OK” button within the “Access Options” window.
7. Access will then display a message onscreen that tells you it will need to close, convert, and re-open the current database to apply this change. Click the “OK” button within the message to continue and convert the database.
8. You can then enter the SQL commands within this tutorial within the “SQL View” of the query design window as well as within modules you create.
9. You can also use the database to connect to external data within SQL Server to execute commands against the tables by using SQL within the Access database.

EXERCISES- Introduction to Databases and SQL

Purpose:

1. To be able to install and use an SQLite desktop relational database system for use in the exercises.

Exercises:

1. Open your computer, connect to the Internet, and open your favorite web browser.
2. The exercises in this tutorial will use SQLite, a no-installation, open-source, desktop database application. You will simply need to download the "Precompiled Binaries" for your operating system from the following web page to use the software: http://sqlite.org/download.html. **Special Note:** If using Mac OSX (or Leopard), the SQLite application is pre-installed in your operating system and there is no need to download a copy of the binaries, if your prefer. However, to run the examples in this tutorial, you must first open the "Terminal" application within the "Utilities" folder in the "Applications" folder within a "Finder" window. You must then enter the following statement **sqlite3 test.db** instead of the **.open test.db** statement whenever that statement is issued within **any** of the Exercises within this tutorial. Note that if you do not want to do this, then you may also download the binaries for Mac OSX to a folder and use them just as the instructions shown in the manual suggest, as well.
3. After downloading the file, unzip the file to extract the "sqlite3.exe" application for Windows or the "sqlite" app for Mac.
4. You can double-click the "sqlite" file that you just extracted to open the "SQLite" command shell application within either a Command Prompt window within a Windows operating system or within a Terminal window within the Mac operating system.
5. You can close the "Command Prompt" or "Terminal" windows to close the connection to the SQLite application.
6. The exercises within this tutorial will build upon one another. It is recommended that you complete them in sequential order to maximize your understanding of SQL with the hands-on exercises. You will need to open SQLite for the exercise at the end of the next chapter, however, you can close the application for now.

CHAPTER 2- DATA DEFINITION LANGUAGE

2.1- The CREATE Statement:

The first statement that you will learn in SQL is the CREATE statement. The CREATE statement is often the first statement that you will execute in SQL if you are using SQL to design a database within a relational database management system.

This statement is commonly used to create a database, table, index, or stored procedure. However, within some implementations, such as “T-SQL,” you will find that the CREATE statement can make many types of database objects, including indexes and schemas.

2.2- The CREATE DATABASE Statement:

Within many relational database management systems, the CREATE DATABASE statement is used to create the database. The core SQL of the statement is shown below.

CREATE DATABASE *database_name*

In this statement, the “*database_name*” parameter is the name you want to give to the database. Note that while most versions of SQL will require a semicolon at the end of the statement, some may not. Also, not every vendor implements this SQL command. In Microsoft Access, for example, you create a database using the graphic user interface. SQLite simply uses its own separate command of “sqlite3 *database_name*” to create a database.

While this is the bare-bones version of the statement, many vendors will also follow this statement with many vendor-specific clauses that will define the specific characteristics of the database. In MySQL 5.7, for example, the statement “CREATE SCHEMA” is a synonym for “CREATE DATABASE.”

The following is a listing of hyperlinks that displays the vendor-specific SQL used and the various clauses available for the CREATE DATABASE statement within MySQL 5.7 and SQL Server 2012. Note that this statement is not available as an SQL statement within Access.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/create-database.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms176061.aspx

2.3- The CREATE TABLE Statement:

One of the most commonly used SQL statements when creating a database is the CREATE TABLE statement, which is used to create tables within a database. The general syntax is shown below.

CREATE TABLE table_name
(
field_name data_type(size),
field_name1 data_type(size),
field_name2 data_type(size),
etc.
)

2.3- The CREATE TABLE Statement- (cont'd.):

The *"table_name"* parameter is the name you want to assign to the table. The *"field_name"* parameters are the names of the fields, or columns, within the table. The *"data_type"* parameter is the declaration of the data type of each field. Each vendor will have vendor-specific names for the data types available. You should check the documentation for the database management system you are using to determine the specific name of each data type available. For example, in Microsoft Access, a text field with a 255 character limit is declared by the "TEXT" data type, but the same type of field is declared by the "TINYTEXT" data type within MySQL. The *"size"* parameter, if needed for the *"data_type"* selected, will define the number of characters to store within the field.

While the above CREATE TABLE statement is the core SQL standard, you will find that the statement tends to be more complex within the actual vendor implementations. Below is a listing of hyperlinks that display the SQL used to implement the CREATE TABLE statement within MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/create-table.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms174979.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff837200.aspx

2.4- The CREATE INDEX Statement:

Within a database, indexes are a way of sorting a table by values contained in one or more fields. The advantage to creating indexes is that they may speed up query processing when used. However, as the user of the database management system, most of the time you can only choose to *create* indexes within the tables, *not* when they will be used. *The database management system will decide when, and if, to use those indexes that you have made available.* While there are some vendor-specific exceptions, this is most often the rule.

Take care when creating indexes to ensure you do not create many, unnecessary indexes on fields that will rarely, if ever, be queried. It is possible to actually decrease query performance by including many unnecessary indexes on tables within the database. Here are some general guidelines to assist you in deciding what indexes to create on the tables within your database. First, only index tables that have a variety of different data types within their fields. Second, indexing is more efficient if the data in your indexed fields gives each record a more unique identification, like a primary key field. Indexing is not usually necessary on fields that have multiple, repeating values. Third, you really only need to index fields which are used for criteria in queries. For example, if you are creating many queries that find records based on phone numbers, you may want to create an index on the field which contains the phone numbers. Assuming your table fields have met these criteria, it can be useful to apply an index to the desired fields to increase the sorting and processing capabilities of data in queries.

The core SQL statements used to create indexes are shown below. Note that there are two different statements used to create an index that only allows unique values and an index that allows duplicate values. Also, be aware that the syntax for creating indexes varies between relational database management systems, so you should check the documentation for the system that you use.

2.4- The CREATE INDEX Statement- (cont'd.):

To create a standard index that allows duplicate field values:

CREATE INDEX *index_name*
ON *table_name* (*field_name*)

To create a unique index that does NOT allow for duplicates field values:

CREATE UNIQUE INDEX *index_name*
ON *table_name* (*field_name*)

Note that in the previous examples, "*index_name"* is the name you want to give to the index. The "*table_name"* parameter is the name of the table that contains the field to index, and the "*field_name"* parameter is the name of the field within the table to index.

Also note that many relational database management systems will allow for the creation of a multi-field, or composite, index. This is often created when the two fields have a logical relationship to one another- such as a "First_Name" and "Last_Name" field. While the implementation of composite index creation will vary by RDBMS, the general syntax is as follows:

CREATE INDEX *index_name*
ON *table_name* (*field_name1, field_name2*)

Like many SQL statements, you will find the CREATE INDEX statement tends to be more complex within the actual vendor implementations. If you would like to view the SQL used to implement the CREATE INDEX statement within MySQL 5.7, SQL Server 2012, and Access 2013, you can find that information at the following hyperlinks.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/create-index.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms188783.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff823109.aspx

2.5- SQL Constraints:

When creating tables in SQL using the CREATE TABLE statement, you can often set various types of constraints on the fields that are created. A constraint is a limitation that is placed upon the allowable values within a field. If a user attempts to add data to the table that violates the constraint, the data will not be added to the table.

Note that while constraints are most often added to a table when using the CREATE TABLE statement, you may also use the ALTER TABLE statement in SQL to later add constraints to an existing table in many relational database management systems. In a later lesson you will see that you can use the

2.5- SQL Constraints- (cont'd.):

ALTER TABLE statement to make changes to the structure of existing tables within a database.

The core SQL syntax to apply constraints when using the CREATE TABLE statement is shown below.

CREATE TABLE *table_name* (*field_name data_type*(*size*) *constraint_name*, *field_name1 data_type*(*size*) *constraint_name*, *etc.*)

Note that the "*table_name*," "*field_name*," "*data_type*," and "*size"* parameters all serve the same function within this statement as they perform within the standard CREATE TABLE statement. The "*constraint_name"* parameter is the name of the constraint to apply to the field. The following constraints are allowed in the core SQL.

NOT NULL is a constraint that you can specify to ensure that the field will not accept NULL values as a valid data entry. A NULL value occurs when no data entry is performed within a field. Note that the NULL value is not equal to an empty text string, a literal space character, or a zero value. Those are all known quantities. A NULL value is simply unknown due to a lack of data entry. Primary key fields and unique index fields cannot contain NULL values. However, you can also specify NOT NULL to disallow null values in fields, regardless of whether or not the field is a primary key field or is uniquely indexed.

UNIQUE is a constraint that you can specify that ensures that all values entered into the field are unique values. There can be no repeating values within a field defined with a UNIQUE constraint.

PRIMARY KEY is the constraint that is specified to define a field (or combination of fields) within a table as being the primary key of the table. Remember that primary key fields must contain unique values and cannot contain NULL values. Therefore, you can think of the PRIMARY KEY constraint as being a combination of the UNIQUE and NOT NULL constraints.

FOREIGN KEY is the constraint that you can specify to ensure that all values within the field correspond to values found within another table's PRIMARY KEY field. This is used as a referential integrity check to ensure that records within a field have a corresponding, or "matching" value within a related table.

CHECK is a constraint that is used to specify the allowable values that can be entered into a table. For example, if you had an "CustomerID" field defined within a CREATE TABLE statement, you could add CHECK (CustomerID > 0) to the CREATE TABLE statement to specify that any values entered into the "CustomerID" field must be values greater than zero.

DEFAULT is a constraint that places a default value into the field specified when a record is entered. For example, if you wanted to have 'East Lansing' appear as the default value for a "City" field within a table when creating the table in the CREATE TABLE statement, you could add DEFAULT 'East Lansing' as the "*constraint_name"* parameter to the "City" field when it is created.

Note that the implementations of the SQL constraints will vary from vendor to vendor. While almost all of these constraints are universally implemented, they are implemented in slightly different ways within each relational database management system. The following hyperlinks will show how to use SQL constraints within MySQL 5.7, SQL Server 2012, and Access 2013.

SQL Constraints (MySQL 5.7):
http://dev.mysql.com/doc/refman/5.7/en/constraints.html

2.5- SQL Constraints- (cont'd.):

Primary Key and Foreign Key Constraints (SQL Server 2012):
http://technet.microsoft.com/en-us/library/ms179610.aspx
Unique Constraints and Check Constraints (SQL Server 2012):
http://technet.microsoft.com/en-us/library/ms187550.aspx

The CONSTRAINT Clause (Access 2013):
http://msdn.microsoft.com/en-us/library/office/ff836971.aspx

2.6- The DROP Statement:

The DROP statement is used within SQL to delete created database objects. You can use this statement to delete databases, tables and indexes. You can also use it in conjunction with the ALTER TABLE statement to delete fields from tables. Note that this is NOT the statement used to delete specified data from tables. You can delete specified data from tables by using the SQL DELETE statement, instead. The following statement shows the core SQL used to delete a database. Note that the parameter "*database_name*" is the name of the database to delete.

DROP DATABASE *database_name*

The next statement is the core SQL statement used to delete a table within a database. Note that the "*table_name*" parameter is the name of the table within the database to delete.

DROP TABLE *table_name*

Note that while you cannot select records to delete with the DROP statement, there is a statement that is considered part of the data definition language in SQL that will delete a table and then re-create the table, thereby effectively deleting all records from the table. This statement is the TRUNCATE TABLE statement. Because it deletes and then re-creates the table, versus selecting individual records to remove from a table, it is considered part of the data definition language versus the data manipulation language. The following statement shows the core SQL used within the TRUNCATE TABLE statement. Note that the "*table_name*" parameter is the name of the table to delete and then re-create.

TRUNCATE TABLE *table_name*

You can also use the DROP statement to delete table indexes. The core SQL of this statement is essentially DROP INDEX, however, it is implemented in slightly different ways within various relational database management systems. The following statement shows the core SQL used within the DROP INDEX statement. Note that the "*index_name*" parameter is the name of the index to delete.

DROP INDEX *index_name*

The implementation of this statement within Microsoft SQL Server is slightly different from the core SQL, in that the parameter needs to know the name of the table associated with the index as well as the index name. The following example shows the DROP INDEX statement in SQL Server. Note that the "*table_name*" parameter is the name of the table associated with the index to delete and the "*index_name*"

2.6- The DROP Statement- (cont'd.):

parameter is the name of the index to delete.

DROP INDEX *table_name.index_name;*

In MySQL, the DROP INDEX statement is only used within the ALTER TABLE statement. So, if using MySQL, the statement would be entered as shown below. Note that the "*table_name*" parameter is the name of the table associated with the index to delete and the "*index_name*" parameter is the name of the index to delete.

ALTER TABLE table_name DROP INDEX *index_name*

Below is a listing of hyperlinks that demonstrate the various implementations of the DROP statement within MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7 (DROP DATABASE):
http://dev.mysql.com/doc/refman/5.7/en/drop-database.html

SQL Server 2012 (DROP DATABASE):
http://technet.microsoft.com/en-us/library/ms178613.aspx

MySQL 5.7 (DROP TABLE):
http://dev.mysql.com/doc/refman/5.7/en/drop-table.html

SQL Server 2012 (DROP TABLE):
http://technet.microsoft.com/en-us/library/ms173790.aspx

MySQL 5.7 (TRUNCATE TABLE):
http://dev.mysql.com/doc/refman/5.7/en/truncate-table.html

SQL Server 2012 (TRUNCATE TABLE):
http://msdn.microsoft.com/en-us/library/ms177570.aspx

MySQL 5.7 (DROP INDEX):
http://dev.mysql.com/doc/refman/5.7/en/drop-index.html

SQL Server (DROP INDEX):
http://technet.microsoft.com/en-us/library/ms176118.aspx

Access 2013 (DROP Statement)
http://msdn.microsoft.com/en-us/library/office/ff821409.aspx

2.7- The ALTER TABLE Statement:

You can use the ALTER TABLE statement in SQL to modify the structure of an existing table in your database. You may do this to add SQL constraints to the tables or fields in the table if you did not do them when initially creating the table. Some relational database management systems may even allow you to use the ALTER TABLE statement to assign a PRIMARY KEY constraint using the ALTER TABLE statement, although that is much more likely to be required within the CREATE TABLE statement in most systems. You can also use this statement to add, edit, or delete fields within a table, if needed.

The core SQL of the ALTER TABLE statement when used to add a field to a table is shown below. Note that the "*table_name*" parameter is the name of the table into which you want to insert the field, the "*field_name*" parameter is the name of the field, and the "*data_type*" parameter is the data type of the field.

ALTER TABLE *table_name*
ADD *field_name data_type*

The core SQL of the ALTER TABLE statement when used to delete a table field is shown below. Note that the "*table_name*" parameter is the name of the table from which you want to remove the field and the "*field_name*" parameter is the name of the field to delete. Note some database systems will not allow you to delete a column. Others simply require that any PRIMARY KEY or FOREIGN KEY constraints on the column first be removed before the field is deleted to protect the referential integrity of the database.

ALTER TABLE *table_name*
DROP COLUMN *field_name*

You can also use the ALTER TABLE statement to modify the data type of fields or add SQL constraints to fields within a table. Note that the SQL used to accomplish this will vary more significantly with each relational database management system. The core SQL for most systems is shown below. Note that the pipe symbol is used to denote alternate commands in different systems and the parenthetical commands are either required or not, once again depending upon the RDBMS used. In the examples shown, "*table_name*" is the name of the table, "*field_name*" is the name of a field, "*data_type*" is the data type of the field, and "*sql_constraint*" is the SQL constraint. The next example shows the core SQL used to change the data type of a field within a table.

ALTER TABLE *table_name*
MODIFY | ALTER COLUMN *field_name data_type*

The next example shows the core SQL used to add or remove an SQL constraint.

ALTER TABLE *table_name*
ADD | DROP (CONSTRAINT) *sql_constraint*

The following hyperlinks contain helpful information about the vendor-specific implementation of the ALTER TABLE statement within MySQL 5.7, SQL Server 2012 and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/alter-table.html

2.7- The ALTER TABLE Statement- (cont'd.):

SQL Server 2012:
http://msdn.microsoft.com/en-us/library/ms190273.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff196148.aspx

2.8- NULL Values in SQL:

When creating tables in SQL, you will need to deal with NULL values. A NULL value is a value that is unknown. These values can occur within database tables when a user who is performing data entry skips entering a value into a field. When a value is not entered, it is said to have a NULL value. Note that a NULL value is not equal to anything, as the value is unknown. It is not equal to zero or a blank text string, such as " ". It is also not greater than or less than any other value. However, you will need to decide how to deal with NULL values when creating fields within tables. You can often choose to disallow NULL values within fields by using the NOT NULL SQL constraint when creating the fields in a table. UNIQUE, PRIMARY KEY, and FOREIGN KEY fields also cannot contain NULL values within them, as NULL values would disrupt the relational abilities of the tables. Note that unless the NOT NULL SQL constraint in employed when creating the fields within a table, the field will typically default to allowing NULL values.

You have some SQL statements to assist you in dealing with NULL values within table fields. Since NULL values are not comparable to other existing values, you cannot use the equal, greater than, or less than sign within a SELECT statement in SQL to find NULL values within fields. When creating queries using the SELECT statement, you will often need to use the IS NULL or IS NOT NULL statements within the SELECT statement to find values within fields where the value is either NULL or not NULL. While we have not yet examined the SELECT statement, which is covered within the Data Manipulation Language chapter, you can examine how the IS NULL and IS NOT NULL clauses can be added to the core SQL of the SELECT statement to find NULL values.

The core SQL of a simple SELECT statement is shown below. In this example, "*table_name*" is the name of the table, "*field_name*" is the name of the field(s) to display in the query, and "*field_name_to_compare*" is the name of the field within which you want to find NULL values or find values that are not NULL. The next example shows the core SQL used to find null values within a field.

SELECT *field_name*, *field_name1*, *field_name2*, *etc.* FROM *table_name*
WHERE *field_name_to_compare* IS NULL

The next example will find any values where there is NOT a NULL value within the "*field_name_to_compare*" field.

SELECT *field_name*, *field_name1*, *field_name2*, *etc.* FROM *table_name*
WHERE *field_name_to_compare* IS NOT NULL

2.9- Data Types in SQL:

When creating table fields in SQL, you must assign each field a data type. However, the data types available to use will vary for each relational database management system you encounter. In this lesson, you will examine some of the most commonly encountered general data types in SQL.

Data	Description
CHARACTER(N)	Fixed-length text, or character, string. N specifies the number of fixed characters in length that the field will contain.
VARCHAR(N) or CHARACTER VARYING(N)	Variable-length text, or character, string. N specifies the MAXIMUM number of characters that the field can contain.
BINARY(N)	Fixed-length binary string. N specifies the fixed length.
BOOLEAN	Stores a TRUE or FALSE value.
VARBINARY(N) or BINARY VARYING(N)	Variable-length binary string. N specifies the MAXIMUM length.
INTEGER(P)	Integer number values, with a precision of P.
SMALLINT	Integer number values with a precision of 5.
INTEGER	Integer number values with a precision of 10.
BIGINT	Integer number values with a precision of 19.
DECIMAL(P,S) or NUMERIC (P,S)	Stores an exact number with a precision of P and a scale of S. For example, a DECIMAL(9,2) would store a number that has 7 digits before the decimal point and 2 digits after the decimal point, with a total of 9 digits stored.
FLOAT(P)	Approximate-number data types for use with floating point numeric data. Floating point data is approximate; therefore, not all values in the data type range can be represented exactly. P is the number of bits used to store the mantissa of the float number in scientific notation and, therefore, dictates the precision and storage size.
REAL	Approximate-number data with a mantissa precision of 7.
FLOAT or DOUBLE PRECISION	Approximate-number data with a mantissa precision of 16.
DATE	Stores year, month, and day values.
TIME	Stores hour, minute, and second intervals.
TIMESTAMP	Stores year, month, day, hour, minute, and second intervals.
INTERVAL	Contains a number of integer fields, representing a period of time, depending on the type of interval.
ARRAY	A fixed-length, ordered set of elements.

2.9- Data Types in SQL- (cont'd.):

Data	Description
MULTISET	A variable-length, unordered set of elements.
XML	Stores XML data.
MONEY, SMALLMONEY or CURRENCY	Stores a data type that represents monetary values. SMALLMONEY store a smaller ranges of values than MONEY. Not implemented in MySQL or Oracle RDBMS.

Note that the specific data type to use for your particular RDBMS may vary. Some, such as MONEY, are not available within some RDBMS at all. Others, like INTEGER, are stated as INT within SQL Server 2012 and MySQL. Note that you should always check your specific RDBMS documentation for the specific data types and names that are available for you to use.

You can find the listing of available data types for MySQL 5.7, SQL Server 2012, and Access 2013 at the following hyperlinks.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/data-types.html

SQL Server 2012:
http://msdn.microsoft.com/en-us/library/ms187752.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff193793.aspx

2.10- Auto-Increment in SQL:

Many times when creating a primary key field within a table, you will assign the field a numeric data type. You can use the "Auto-Increment" feature when creating this field in many relational database systems to automatically increase the increment used within this numeric field to ensure that each record entered will receive its own unique identification number. However, the implementation of this feature varies widely amongst the individual relational database management systems. There is no core SQL for the implementation of this specific feature. You must check the documentation for your specific RDBMS to create a primary key field that contains auto-incrementing numeric values. Below is the general SQL syntax used to denote an auto-increment field when creating a table in MySQL.

```
CREATE TABLE table_name
(
field_name INT NOT NULL AUTO_INCREMENT,
field_name1, data_type1, sql_constraints1,
PRIMARY KEY field_name
)
```

In the MySQL example shown, you add the "AUTO_INCREMENT" constraint at the end of the field to define it as containing auto-incrementing values. Note that this field was also assigned the "Int" data type

2.10- Auto-Increment in SQL- (cont'd.):

so that it will contain numeric integer values. It was also assigned the NOT NULL SQL constraint so that it cannot contain NULL values. The last line of the syntax used the PRIMARY KEY constraint to define the field as the primary key of the table. Note that when you add records to this table in the future using the INSERT statement, you will not need to specify a value for the auto-incremented field. It will automatically assign a number when a new record is added. The field will begin with the number 1. If you wish to start at a different number, you can use the ALTER TABLE command to set the starting number by using the following general syntax in MySQL. Note that the "*number*" parameter shown is the number at which you want the auto-incrementing field to begin numbering.

ALTER TABLE *table_name* AUTO_INCREMENT=*number*

If using Microsoft SQL Server, you will use the IDENTITY(*x*,*y*) constraint to specify auto-incrementing field values when creating the table. Note that when using the IDENTITY constraint, the "*x*" parameter specifies the starting number of the field and the "*y*" parameter specifies the increasing numbering sequence to use. For example, IDENTITY(1,1) will create a field that starts numbering at the number 1 and increases its value by 1 for each future record added. Alternately, IDENTITY(10,2) will start numbering at 10 and increase consecutively by 2 for each record added (10, 12, 14, 16, etc.). As with many other RDBMS implementations, however, you will not need to specify a value for the auto-incrementing field when using the INSERT statement to add records to the table in the future. The general syntax for creating an auto-incrementing field within SQL Server in the CREATE TABLE statement is shown below.

CREATE TABLE *table_name*
(
field_name INT IDENTITY(*x*,*y*) PRIMARY KEY,
field_name1 data_type1,
etc.
);

If using Microsoft Access, you can basically substitute the keyword AUTOINCREMENT for the IDENTITY keyword as used with SQL Server to create an auto-incrementing field within a table in the CREATE TABLE statement. You can simply use the AUTOINCREMENT keyword by itself to create an auto-incrementing field that starts at 1 and numbers in increasing values by 1. You can also use the AUTOINCREMENT(*x*,*y*) keyword variation to specify both a starting number and a numeric increment by which to increase the values assigned. The general SQL syntax for creating an auto-incrementing primary key field within a Microsoft Access table is shown below. Note that, in Microsoft Access, you do not need to specify a "*data_type*" parameter when creating an AUTOINCREMENT field. This field also corresponds to the "AutoNumber" data type used when creating tables in the visual "Design" view of a table within the Microsoft Access interface. In this way, AUTOINCREMENT is also a declaration of data type within this particular RDBMS.

CREATE TABLE *table_name*
(*field_name* AUTOINCREMENT PRIMARY KEY, *field_name1 data_type1, etc.*);

ACTIONS- DATA DEFINITION LANGUAGE

THE CORE SQL OF THE CREATE DATABASE STATEMENT:

CREATE DATABASE *DatabaseName*

THE CORE SQL OF THE CREATE TABLE STATEMENT:

CREATE TABLE table_name
(
field_name1 data_type(size),
field_name2 data_type(size),
field_name3 data_type(size),
....
)

THE CORE SQL OF THE CREATE INDEX STATEMENT:

1. To create a standard index that allows duplicate field values:

CREATE INDEX *index_name*
ON *table_name* (*field_name*)

2. To create a unique index that does NOT allow for duplicates field values:

CREATE UNIQUE INDEX *index_name*
ON *table_name* (*field_name*)

THE CORE SQL USED TO APPLY SQL CONSTRAINTS:

CREATE | ALTER TABLE *table_name*
(
field_name1 data_type(*size*) *constraint_name*,
field_name2 data_type(*size*) *constraint_name*,
field_name3 data_type(*size*) *constraint_name*,
....
)

THE CORE SQL OF THE DROP STATEMENT:

1. To delete a database:

DROP DATABASE *database_name*

2. To delete a table within a database

DROP TABLE *table_name*

THE CORE SQL OF THE DROP STATEMENT- (CONT'D.):

3. To truncate a table within a database:

TRUNCATE TABLE *table_name*

4. To delete an index within a database

DROP INDEX *index_name*

THE CORE SQL OF THE ALTER TABLE STATEMENT:

1. To alter a table to add a field:

ALTER TABLE *table_name*
ADD *field_name data_type*

2. To alter a table to delete a field:

ALTER TABLE *table_name*
DROP COLUMN *field_name*

3. To alter a table to modify a field's data type:

ALTER TABLE *table_name*
MODIFY | ALTER COLUMN *field_name data_type*

4. To alter a table to add an SQL constraint:

ALTER TABLE *table_name*
ADD (CONSTRAINT) *sql_constraint*

5. To alter a table to delete an SQL constraint:

ALTER TABLE *table_name*
DROP (CONSTRAINT) *sql_constraint*

COMMONLY USED SQL CONSTRAINTS:

1. NOT NULL is a constraint that you can specify to ensure that the field will not accept NULL values as a valid data entry.
2. UNIQUE is a constraint that you can specify that ensures that all values entered into the field are unique values. There can be no repeating values within a field defined with a UNIQUE constraint.
3. PRIMARY KEY is the constraint that is specified to define a field (or combination of fields) within a table as being the primary key of the table. Remember that primary key fields must contain unique values and cannot contain NULL values. Therefore, you can think of the PRIMARY KEY constraint as being a combination of the UNIQUE AND NOT NULL constraints.
4. FOREIGN KEY is the constraint that you can specify to ensure that all values within the field correspond to values found within another table's PRIMARY KEY field. This is used as a referential integrity check to ensure that records within a field have a corresponding, or "matching" value within a related table.
5. CHECK is a constraint that is used to specify the allowable values that can be entered into a table. For example, if you had an "CustomerID" field defined within a CREATE TABLE statement, you could add CHECK (CustomerID > 0) to the CREATE TABLE statement to specify that any values entered into the "CustomerID" field must be values greater than zero.
6. DEFAULT is a constraint that places a default value into the field specified when a record is entered. For example, if you wanted to have 'East Lansing' appear as the default value for a "City" field within a table when creating the table in the CREATE TABLE statement, you could add DEFAULT 'East Lansing' as the "*constraint_name*" parameter to the "City" field when it is created

THE CORE SQL OF A SIMPLE SELECT STATEMENT TO FIND NULL OR NOT NULL VALUES:

1. To find null values within the "*field_name_to_compare*" field.

SELECT *field_name*, *field_name1*, *field_name2*, *etc.* FROM *table_name*
WHERE *field_name_to_compare* IS NULL

2. To find any values where there is NOT a NULL value within the "*field_name_to_compare*" field.

SELECT *field_name*, *field_name1*, *field_name2*, *etc.* FROM *table_name*
WHERE *field_name_to_compare* IS NOT NULL

ACTIONS- DATA DEFINITION LANGUAGE

GENERAL SQL DATA TYPES:

Data	Description
CHARACTER(N)	Fixed-length text, or character, string. N specifies the number of fixed characters in length that the field will contain.
VARCHAR(N) or CHARACTER VARYING(N)	Variable-length text, or character, string. N specifies the MAXIMUM number of characters that the field can contain.
BINARY(N)	Fixed-length binary string. N specifies the fixed length.
BOOLEAN	Stores a TRUE or FALSE value.
VARBINARY(N) or BINARY VARYING(N)	Variable-length binary string. N specifies the MAXIMUM length.
INTEGER(P)	Integer number values, with a precision of P.
SMALLINT	Integer number values with a precision of 5.
INTEGER	Integer number values with a precision of 10.
BIGINT	Integer number values with a precision of 19.
DECIMAL(P,S) or NUMERIC (P,S)	Stores an exact number with a precision of P and a scale of S. For example, a DECIMAL(9,2) would store a number that has 7 digits before the decimal point and 2 digits after the decimal point, with a total of 9 digits stored.
FLOAT(P)	Approximate-number data types for use with floating point numeric data. Floating point data is approximate; therefore, not all values in the data type range can be represented exactly. P is the number of bits used to store the mantissa of the float number in scientific notation and, therefore, dictates the precision and storage size.
REAL	Approximate-number data with a mantissa precision of 7.
FLOAT or DOUBLE PRECISION	Approximate-number data with a mantissa precision of 16.
DATE	Stores year, month, and day values.
TIME	Stores hour, minute, and second intervals.
TIMESTAMP	Stores year, month, day, hour, minute, and second intervals.
INTERVAL	Contains a number of integer fields, representing a period of time, depending on the type of interval.
ARRAY	A fixed-length, ordered set of elements.
MULTISET	A variable-length, unordered set of elements.
XML	Stores XML data.

GENERAL SQL DATA TYPES- (CONT'D.):

Data	Description
MONEY, SMALLMONEY or CURRENCY	Stores a data type that represents monetary values. SMALLMONEY store a smaller ranges of values than MONEY. Not implemented in MySQL or Oracle RDBMS.

SQL* USED TO CREATE AN AUTOINCREMENT FIELD IN A TABLE:

1. General MySQL Syntax:

CREATE TABLE *table_name*
(
field_name Int NOT NULL AUTO_INCREMENT,
field_name1, *data_type1*, *sql_constraints1*,
PRIMARY KEY *field_name*
)

2. General SQL Server Syntax:

CREATE TABLE *table_name*
(
field_name int IDENTITY(*x*,*y*) PRIMARY KEY,
field_name1 data_type1,
etc.
);

3. General Access 2013 Syntax:

CREATE TABLE *table_name*
(*field_name* AUTOINCREMENT PRIMARY KEY, *field_name1 data_type1, etc.*);

* Note that there is no "core" SQL for this statement as the AUTOINCREMENT usage varies widely by RDBMS.

EXERCISES- Data Definition Language

Purpose:

1. To be able to use DDL statements within SQL to create a simple database and tables.

Exercises:

1. Double-click the "sqlite" file that you extracted in the Exercise at the end of Chapter 1 to open the "SQLite" command shell application within either a Command Prompt window within a Windows operating system or within a Terminal window within the Mac operating system.
2. Enter the commands within these Exercises into those windows in your respective operating system.
3. Type the following command line into either the Command Prompt window or Terminal window to create and open a new permanent database file called "test.db" within SQLite.
4. **.open test.db**
5. Press the "Enter" key on your keyboard to create the new database file named 'test.db,' as well as log into the newly created database file. You should now see your cursor appear in front of the words 'sqlite>' within the window. You will enter the following commands after that prompt.
6. Type the following command into the window. Do not follow it with a semicolon as it is not an SQL statement. It is an internal SQLite command that will list the databases used by SQLite. You will do this so that you can note where the 'test.db' file is actually being stored within your computer. The name of the file, and its location within your computer, will be shown within the window.
7. **.databases**
8. Press the "Enter" key on your keyboard to execute the preceding command.
9. Type the following SQL statement into the window to create a "Customers" table. *BE SURE TO INCLUDE THE SEMICOLON AT THE END OF THE STATEMENT!* All SQL statements within SQLite must end with a semicolon, otherwise you will simply display an extension of the command line (shown by the 'sqlite>' prompt) when you press the "Enter" key on your keyboard.
10. **CREATE TABLE Customers (CustomerID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, CompanyName TEXT, Address TEXT, City TEXT, State CHAR(2), Zip CHAR(5), Phone CHAR(10));**
11. Type the following statement into the window to create an "Employees" table. Press the "Enter" key on your keyboard after typing the entire statement.
12. **CREATE TABLE Employees (EmployeeID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, FirstName TEXT, LastName TEXT);**
13. Type the following statement into the window to create an "Items" table. Press the "Enter" key on your keyboard after typing the entire statement.
14. **CREATE TABLE Items (ItemID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, ItemName TEXT);**
15. Type the following statement into the window to create a "Sales" table. Press the "Enter" key on your keyboard after typing the entire statement.
16. **CREATE TABLE Sales (SaleID INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, EmployeeID INTEGER REFERENCES Employees(EmployeeID) NOT NULL, CustomerID INTEGER REFERENCES Customers(CustomerID) NOT NULL, Saledate TEXT NOT NULL);**
17. Type the following statement into the window to create a "SalesDetails" table. Press the "Enter" key on your keyboard after typing the entire statement.
18. **CREATE TABLE SalesDetails (SaleID INTEGER REFERENCES Sales(SaleID) NOT NULL, ItemID INTEGER REFERENCES Items(ItemID) NOT NULL, Price DECIMAL(10,2) NOT NULL, Quantity INTEGER NOT NULL, PRIMARY KEY (SaleID, ItemID));**

Exercises- (cont'd.):

19. Now that you have created the basic tables within the database file, you will now turn to creating a few indexes on those tables. Type the following statement into the window to create an index. Press the "Enter" key on your keyboard after typing the entire statement.

20. CREATE INDEX CustomerID on Customers(CustomerID);

21. Type the following statement into the window to create an index. Press the "Enter" key on your keyboard after typing the entire statement.

22. CREATE INDEX ItemID on Items(ItemID);

23. Type the following statement into the window to create an index. Press the "Enter" key on your keyboard after typing the entire statement.

24. CREATE INDEX SaleID on Sales(SaleID);

25. Type the following statement into the window to create an index. Press the "Enter" key on your keyboard after typing the entire statement.

26. CREATE INDEX EmployeeSales on Sales(EmployeeID);

27. Type the following statement into the window to create an index. Press the "Enter" key on your keyboard after typing the entire statement.

28. CREATE INDEX CustomerSales on Sales(CustomerID);

29. You can close the Terminal or Command Prompt window at this point, if desired. Be sure to keep the 'test.db' file that you have created, as you will need it for the upcoming Exercises at the end of each chapter. *The Exercises at the end of each chapter build upon one another and must be completed in sequential order.*

CHAPTER 3- Data Manipulation Language

3.1- The INSERT Statement:

After examining the core SQL statements used to create objects within relational database management systems, you should next learn how to add, edit, delete and select data within these same systems. SQL uses the Data Manipulation Language statements to accomplish these tasks. The first statement to learn is the INSERT statement. This statement is used to insert *new* records into a table. Note that this is NOT the command used to update information within existing table records. That task is accomplished by the UPDATE statement, which will also be discussed within this chapter.

The core SQL of the INSERT statement used to create new records within a table is shown below. The "*table_name*" is the name of the table within which the records are to be created, "*field_name*" is the name of the field within which to place the data, and "*value*" is the value of the data to place into the field.

INSERT INTO *table_name* (*field_name*, *field_name1*, *field_name2*, *etc.*)
VALUES (*value*, *value1*, *value2*, *etc.*)

Note that it is possible to omit the "field_name" parameters within some implementations of the INSERT statement in some relational database management systems. Also, you will not need to specify a *value* within this statement for any fields that are assigned an auto-incrementing field value. You may also skip entering the names of *field_name* parameters into which you would like to enter NULL values, if those fields can accept the entry of a NULL value and you have no actual data to enter into those fields.

You can view the specific implementations of the INSERT statement within MySQL 5.7, SQL Server 2012, and Access 2013 by clicking the following hyperlinks.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/insert.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms174335.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff834799.aspx

3.2- The UPDATE Statement:

After you have data within the tables in your database, you will often need to update those records. SQL uses the UPDATE statement to update field data within specified records in a table. The core SQL of the UPDATE statement is shown below. Note that the "*table_name"* parameter is the name of the table that contains the records to update. The "*field_name"* parameter is the name of a field within the table. The "*update_value"* parameter is the value to which you want to update the associated "*field_name*." The "*existing_value"* parameter is the existing value that you want to update.

UPDATE *table_name*
SET *field_name*=*update_value*, *field_name1*=*update_value1*, *etc.*
WHERE *field_name*=*existing_value*

Note that the WHERE clause within this statement is VERY important! Without this clause, which

3.2- The UPDATE Statement- (cont'd.):

specifies exactly which records within the table to update, you will update EVERY record within the table! Always be sure to double-check your WHERE clause before executing an UPDATE on table records.

The following hyperlinks will demonstrate how to implement the UPDATE statement within SQL for MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/update.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms177523.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff845036.aspx

3.3- The DELETE Statement:

The DELETE statement is used to remove specified records from a table. Like the UPDATE statement, it includes a WHERE clause that you should ensure is correct before executing the DELETE statement. If you do not specify a WHERE clause within the DELETE statement, you will delete ALL the records within the table! The core SQL of the DELETE statement is shown below. Note that the "*table_name*" parameter is the name of the table that contains the records to delete. The "*field_name*" parameter is the name of a field within the table. The "*delete_value*" parameter is the value within the "*field_name*" specified, for which you want to delete any matching records.

DELETE FROM *table_name*
WHERE *field_name=delete_value*

Note that if you want to delete ALL records from a table, you can use the DELETE FROM statement without the WHERE clause. Executing this statement will delete all records from the table specified, but leave the structure and indexes of the table intact. As mentioned earlier, you should ALWAYS use care to double-check your SQL statement before executing a DELETE or UPDATE statement.

The specific SQL implementations of the DELETE statement within MySQL 5.7, SQL Server 2012, and Access 2013 are shown at the hyperlinks listed below.

My SQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/delete.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms189835.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff195097.aspx

3.4- The SELECT Statement:

You use the SELECT statement in SQL to choose specific records to view from a table, or from related tables, within a temporary table called a result set. This tutorial will start the examination of the SELECT statement by looking at the SELECT statement in its simplest form when it is used to select all of the records from a single table. The core SQL of the statement is shown below. Note that the "*table_name*" parameter is the name of the table from which you want to select all the records and the "*field_name*" parameter is the name of a field within the table. Keywords within braces { } are optional.

SELECT { DISTINCT } *field_name*, *field_name1*, *field_name2*, *etc.*
FROM *table_name*

or

SELECT * FROM *table_name*

In the first example shown above, the names of the fields of data you want to see within the result set are specified by name after writing the SELECT statement. You can write a SELECT statement in this manner to only show selected data fields from a table within the result set.

The DISTINCT keyword, if used, will only display unique record values within the *field_names* that are specified. This is used when you do not want duplicate records to be returned within the result set. By default, all records will be returned within a result set unless the DISTINCT keyword is used.

In the second example, the asterisk character is used to specify that ALL of the fields within the table should appear within the result set. The second statement will produce a result set that is, essentially, a temporary copy of all the records and fields within the table specified.

Note that while the statement shown is the basis of the SELECT statement, the following lessons within this chapter will show how the SELECT statement is augmented with additional clauses and keywords to enhance its abilities. The SELECT statement is one of the most powerful and potentially complex statements within the SQL language. It is arguably the most important statement within SQL.

The following hyperlinks display the full potential uses of the SELECT statement within MySQL 5.7, SQL Server 2012, and Access 2013. Note that the SELECT statement, as shown within these web pages, is actually quite complex. This tutorial will continue to explain these various clauses and keywords within the SELECT statement in the upcoming lessons.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/select.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms189499.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff821148.aspx

3.5- The WHERE Clause:

When used within the SELECT statement, the WHERE clause will determine which records you want to select from a specified table or tables. It does this by selecting records from a table where the values within a field you specify match a criteria you specify. Many times, the criteria will be matching, or "equal to," a field value. Note, however, that this will not necessarily be the case in all situations. You can also specify a criteria which looks for values within a field that are greater than or less than a specified value. You can also look for NULL values within a field. There are many ways to compare field values to specified criteria. No matter which criteria you specify, the criteria will always appear within the WHERE clause when used in the SELECT statement. You can examine the core SQL of the SELECT statement with the WHERE clause below. Note that the "*table_name"* parameter is the name of a table from which you want to select the records, the "*field_name"* parameter is the name of a field, and the "*criteria"* parameter is the value you want to find within the specified field. The braces are used to denote an area where a choice must be made, and the pipe symbol is used to separate the choices available in that area.

SELECT {* | *field_name*, *field_name1*, *etc.*}
FROM *table_name*
WHERE *field_name* = *criteria*

As mentioned earlier in this lesson, this example shows a WHERE clause that is used to find records where a field is equal to a value specified. While this is a very common use of the WHERE clause, it can also be used to find values that are not equal to a specified value in some respect, as well. You can use the following comparison operators within the criteria to specify the comparison operation to perform on the criteria value specified within the WHERE clause in SQL.

Operator	Description
=	Equals. Used to find values in a field that are equal to a value you specify.
<> *or* !=	Not equal to. Used to select values in a field that are NOT equal to a value you specify. The symbol used can change depending on the version of SQL implemented within your RDBMS.
>	Greater than. Used to find values in a field greater than a value specified.
<	Less than. Used to find values in a field less than a value specified.
>=	Greater than or equal to. Used to find values in a field greater than or equal to a value specified.
<=	Less than or equal to. Used to find values in a field less than or equal to a value specified.
BETWEEN	Used to find values in a field including and between two values that are specified.
LIKE	Used to find values in a field that match a pattern that is specified.
IN	Used to find values that match multiple values within a list of values that are specified.
AND	Used to join multiple selection criteria together. Selects records that match *both* criteria joined by the AND operator.
OR	Used to join multiple selection criteria together. Selects records that match *either* criteria joined together by the OR operator.

3.5- The WHERE Clause- (cont'd.):

The following hyperlinks display many uses of the WHERE clause within the SELECT statement for SQL Server 2012 and Access 2007, which still applies to Access 2013. While there is no specific official online documentation for the WHERE clause within MySQL 5.7, a listing of expression syntax that can be included within a WHERE statement for MySQL 5.7 is referenced in the hyperlink below.

MySQL 5.7 (Expression Syntax):
http://dev.mysql.com/doc/refman/5.7/en/expressions.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms188047.aspx

Access 2007 (Also Applies to 2013):
http://office.microsoft.com/en-us/access-help/access-sql-where-clause-HA010278156.aspx

3.6- Criteria Notation & Wildcard Characters in WHERE Clause:

You should also ensure that the criteria value you specify matches the data type of the field within which you are searching for that value when you enter a criteria value to find within a field in the WHERE clause. For example, when searching for a matching text value within a field that contains a text data type, you will often enclose the criteria value to find within single or double quotation marks, depending on which set of quotation marks is preferred by the SQL used by your RDBMS. Many implementations will accept either set. For example, if searching for the value of "East Lansing" or "East Grand Rapids" within a "City" field in the WHERE clause, the resulting WHERE clause may look like the following example.

WHERE City = 'East Lansing' OR City = 'East Grand Rapids'

or

WHERE City = "East Lansing" OR City = "East Grand Rapids"

The quotation marks simply indicate to the RDBMS that it should be searching for a text value within the specified field. This means that you must know the data type of the field specified by the WHERE clause. Note that these criteria values are often called "literal values" or "constants."

The following table lists some of the most commonly used data type notation for literal values and wildcard characters when specifying criteria values within the WHERE clause in RDBMS implementations. You must always check the specific documentation for your RDBMS to be sure you are using the correct criteria notation. DATE, TIME, DATETIME, and TIMESTAMP values, in particular, can have more varied criteria notation within relational database management systems. Note that in most RDBMS implementations, numeric criteria values do not have any specified notation.

Wildcard characters, which are often used with the LIKE operator, represent unknown values within a text field. You can use wildcard characters in the WHERE clause to denote "unknown" values within a pattern you want to find within a field. For example, if you wanted to find the name of any city that started with the word "East" within a "City" field, you could use the following wildcard characters within the criteria of the WHERE clause shown in the following possible examples. Note that the specific wildcard character to use for this purpose will vary, depending on which RDBMS you are using.

3.6- Criteria Notation & Wildcard Characters in WHERE Clause- (cont'd.):

WHERE City LIKE 'East %'

or

WHERE City LIKE "East *"

Notation	Use
' ' *or* " "	Specifies a text value or text string. Sometimes used to denote any non-numeric value.
None	Numeric values do not use any notation. Numbers are signified by a lack of notation.
# #	Specifies a date/time value in Microsoft Access.
[ ]	Specifies a parameter or other field name value to use as the criteria in Microsoft Access. Example: [table.field]. Used to denote a set of initial characters to match when used in conjunction with wildcard characters in most other RDBMS. Example: '[a-d]%'
% *or* *	Wildcard character. Used to denote multiple unknown characters.
_ *or* ?	Wildcard character. Used to denote a single unknown character.

The following web pages provide additional resources that show how to denote literal values or constants within MySQL 5.7, SQL Server 2012, and Access 2013. Note that the resource page for Access 2013 shows many different examples of query criteria used within the "Design View" of that application. Note that you can use these same criteria when designing queries within the "SQL View" within that application. It also displays the notation of literal values within the application.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/literals.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms179899.aspx

Access 2013:
http://office.microsoft.com/en-us/access-help/examples-of-query-criteria-HA102815718.aspx?CTT=1

3.7- The ORDER BY Clause:

When viewing the result set of a SELECT statement in SQL, the records will be displayed in the order that they were selected from the table. This means that, by default, they will not be displayed in any particular order within the result set. You use the ORDER BY clause within the SELECT statement to sort the records selected within the result set. The core SQL of the SELECT statement with the ORDER BY clause is shown below. Note that the "*table_name*" parameter is the name of the table from which you want to select the records, the "*field_name*" parameter is the name of a field, and "*criteria*" is a value to find within the specified field. ASC stands for "ascending" and DESC stands for "descending. The braces are used to denote an area where a choice must be made, and the pipe symbol is used to separate the

3.7- The ORDER BY Clause- (cont'd.):

choices available in that area.

SELECT {* | *field_name*, *field_name1*, *etc.*}
FROM *table_name*
WHERE *field_name* = *criteria*
ORDER BY *field_name* {ASC | DESC}, *field_name1* {ASC | DESC}, *etc.*

Note that you often will not need to specify "ASC" for each field by which you want to sort the result set, as the field will be sorted in ascending order (1-9, A-Z) by default. However, to sort the records by the values within the field in descending order (9-1, Z-A), you must specify the DESC keyword after each named field that you want to sort in descending order.

The following hyperlinks display the use and optimization of the ORDER BY clause within the SELECT statement for MySQL 5.7, SQL Server 2012 and Access 2007, which also applies to Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/order-by-optimization.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms188385.aspx

Access 2007 (Also Applies to 2013):
http://office.microsoft.com/en-us/access-help/order-by-clause-HA001231493.aspx

3.8- The GROUP BY Clause and Aggregate Functions:

You will often want to perform some type of function on the records selected within a SELECT statement. For example, you may want to count the number of records selected. Likewise, you may want to count sets of records within a result set. For example, you may want to count the number of records within a table by state, or by country. You use aggregate functions to perform these tasks in SQL. A function is simply a type of mathematical or computational operation, such as COUNT, SUM, or AVERAGE, for example. The term aggregate means a whole or total composed of separate parts. The term aggregate function is used to describe the way that the function will perform its computation over a range, or group, of records. The grouping of records upon which the function is performed could be all of the records selected, or they could be groupings created by the unique values found within one or more fields.

While you may sometimes want to perform an aggregate function, such as COUNT, on all of the records within a SELECT statement to find the total count of the records; in real-life application of SQL, you will most often use the aggregate functions in conjunction with the GROUP BY clause. In this lesson you will examine using the GROUP BY clause in the SELECT statement to group records by values within a field and perform aggregate functions on those grouped values. For example, grouping the distinct values found within a "State" field and then finding the SUM of a "Sales" field for each grouping created within the "State" field to find the total amount of sales by state.

The core SQL of the SELECT statement with the GROUP BY clause is shown next. Note that the "*table_name*" parameter is the table name from which you want to select the records, the "*field_name*" parameter is the name of a field, "*aggregate_function*" is an aggregate function you want to perform upon

3.8- The GROUP BY Clause and Aggregate Functions- (cont'd.):

the specified field, and *criteria* is a criteria expression used for selection. Elements shown within brackets [ ] are optional. Elements shown within braces { } are simply optional extensions of the SELECT clause that can be incorporated, if needed, into a single statement. These are only shown so that you will be aware of the order in which the clauses should be placed within a single SELECT statement.

SELECT *field_name*, *aggregate_function*(*field_name1*), [*etc.*]
FROM *table_name*
{ WHERE clause }
GROUP BY *field_name*, [*etc.*]
[HAVING *criteria*]
{ ORDER BY *field_name* [*DESC*] }

In many relational database systems, the fields named within the GROUP BY clause must also appear within the SELECT clause, meaning that you may only GROUP BY a field that has been selected within the table. Also note that some RDBMS implementations may also allow for the use of field labels, field numbers (order of field from left to right), and more complex expressions within the GROUP BY clause.

Note that when using the GROUP BY clause, you may also choose to use the optional HAVING clause within the SELECT statement. The HAVING clause is simply a selection criteria that determines which groups to display. It functions like a WHERE clause for the GROUP BY clause, however the *criteria* used in the HAVING clause can include aggregate functions that the WHERE clause will not allow. Consider the following example that shows the sum of a "Sales" field for groupings within a "State" field.

```
SELECT State, SUM(Sales)
FROM Table
GROUP BY State
HAVING SUM(Sales) > 100000
```

In the previous example, the HAVING clause would restrict the display of any groups where the total of the "Sales" field for the "State" group was less than 100,000. Note that this is not a selection that could be accomplished within the WHERE clause. When creating a SELECT statement that includes both a WHERE clause and a HAVING clause, remember that the WHERE clause is used to select *which records* are used to create the aggregated groups, while the HAVING clause is used to select *which groups* to display *after* the selected records have been aggregated.

The following hyperlinks display the use of the GROUP BY clause within the SELECT statement for SQL Server 2012 and Access 2010 and 2007, which also applies to Access 2013. Additional functions and modifiers that can be used with the GROUP BY clause within MySQL 5.7 are also listed at the following hyperlink as an additional learning resource.

MySQL 5.7 (Functions and Modifiers for Use with GROUP BY Clauses):
http://dev.mysql.com/doc/refman/5.7/en/group-by-functions-and-modifiers.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms177673.aspx

3.8- The GROUP BY Clause and Aggregate Functions- (cont'd.):

Access 2010 (Also Applies to 2013):
http://office.microsoft.com/en-001/access-help/group-by-clause-HA001231482.aspx

The following web page hyperlinks list aggregate functions, used with the GROUP BY and HAVING clauses, that are accepted within MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/group-by-functions.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms173454.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff197054.aspx

3.9- The JOIN Clause:

The next aspect of the SELECT statement to discuss is the JOIN clause. Joins between two tables occur when the values within a PRIMARY KEY field of one table are linked to values within the FOREIGN KEY field of another table. Creating joins within the SELECT statement allows the user to access data from multiple tables within a single result set, and is one of the primary functions of a relational database.

There are many variations on the type of join that can occur between the PRIMARY KEY and FOREIGN KEY fields between two tables in SQL. Each type of join is referenced by a slightly different name. The different types of SQL joins are the INNER JOIN, the LEFT JOIN, the RIGHT JOIN, and the FULL JOIN. Within different RDBMS these same joins may also be referred to as JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, and FULL OUTER JOIN. You should check your RDBMS documentation to see which version of the name is preferred. Often both are acceptable.

The core SQL of the SELECT statement with a JOIN clause is shown below. Note that the "*table_name*" parameter is the name of a table to join and from which you want to select records, the "*field_name*" parameter is the name of a field within a joined table, the "*primary_key*" parameter is the name of the primary key field in the table and the "*foreign_key*" parameter is the name of the foreign key field in the table. Choices are shown within braces {} with the pipe symbol | separating the available choices.

SELECT *field_name*, *field_name1*, *field_name2*, *etc.*
FROM *table_name*
{ INNER JOIN | LEFT JOIN | RIGHT JOIN | FULL JOIN } *table_name1*
ON *table_name.primary_key* = *table_name1.foreign_key*

Note that in the ON clause, which explicitly names the two fields that are to be joined together, you see the dot notation of *table_name.field_name* being used. This dot notation is commonly used to refer to a specific field within a specific table. It is often used to refer to a field in any part of a SELECT statement where there is a possible ambiguous or duplicate field name. Most often, it is used within the ON statement which joins two fields, which often share the same field name, in two different tables.

The first join type you will examine is named the INNER JOIN, or often simply JOIN, as it is the most commonly used type of join between tables. An inner join will display records that have a matching

3.9- The JOIN Clause- (cont'd.):

value in the PRIMARY KEY and FOREIGN KEY columns between two tables. Records that do not have a matching value within *both* of these two columns will not be displayed within the result set. For example, if you have a "Customers" table with a primary key column that is joined to a "Sales" table with a related foreign key column, then choosing an INNER JOIN between the two tables will show any customers who have associated sales records. Customers without associated sales records will not be shown.

The next type of join to examine is the LEFT JOIN, which is also sometimes called a LEFT OUTER JOIN. A LEFT JOIN between two tables will display all records from the left table (the *table_name* table), as well as any associated records from the RIGHT table (the *table_name1* table) where the values between the joined fields are equal. Using the "Customers" and "Sales" example from before, using a LEFT JOIN between the two tables would show ALL "Customer" records as well as any associated "Sales" for those customers.

The next type of join to examine is the RIGHT JOIN, which is also sometimes called a RIGHT OUTER JOIN. A RIGHT JOIN between two tables will display all records from the right table (the *table_name1* table), as well as any associated records from the LEFT table (the *table_name* table) where the values between the joined fields are equal. Using the "Customers" and "Sales" example from before, using a RIGHT JOIN between the two tables would show ALL "Sales" records as well as any associated "Customers" for those sales.

The last type of join to examine is the FULL JOIN, which is sometimes also called a FULL OUTER JOIN. A FULL JOIN between two tables will display ALL records from BOTH tables, as well as their associations where the values between the joined fields are equal. Using the "Customers" and "Sales" example from before, using a FULL JOIN between the two tables would show ALL "Customer" records as well as ALL "Sales" records. It will also display which "Customer" records" are associated with which "Sales" records.

Note that the JOIN clause is placed after the FROM clause and before the WHERE clause within the larger syntax of the SELECT statement in SQL. There are also variations of the syntax of the JOIN clauses in many RDBMS. For example, in SQL Server 2012, you can simply use the WHERE *table_name.primary_key* = *table_name1.foreign_key* clause within the WHERE statement to create an implicit join between them, although this is not the recommended method. Likewise, MySQL 5.7 supports the keyword NATURAL JOIN, which will create a join between two tables with a shared field name. You should always check the documentation for your RDBMS to see what is allowed when joining tables and what terms are used. Below are hyperlinks to web pages that provide information on creating joins within the SELECT statement in MySQL 5.7, SQL Server, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/join.html

SQL Server:
http://technet.microsoft.com/en-us/library/ms191517.aspx

Access 2013:
(INNER JOIN): http://msdn.microsoft.com/en-us/library/office/ff197346.aspx
(LEFT & RIGHT JOIN): http://msdn.microsoft.com/en-us/library/office/ff198084.aspx

3.10- The UNION Operator:

The UNION operator allows you to combine the result sets of two or more individual SELECT statements into a single result set. Note that the UNION operator is not a JOIN, but rather a way to combine the data within two result sets into the same fields in single result set. Because of this, a UNION operation can only combine two SELECT statements with the same number of fields. These fields must also share compatible data types within the two separate SELECT statements. If combining two SELECT statements that contain multiple fields, the fields must also be in the same order from left to right.

By default, the UNION clause will only return unique values in the fields that it is combining. If you want the UNION clause to return ALL records from both SELECT statements, then you must use the UNION ALL clause, instead. Also note that if combining fields with different field name values, the field name values of the first SELECT statement will be the ones that are used in the combined result set.

The core SQL of the UNION clause is shown below. The "*select_statement"* parameter is simply the SELECT statement that you want to combine with another SELECT statement.

select_statement
UNION
select_statement1

or

select_statement
UNION ALL
select_statement1

The following web page hyperlinks list the syntax of the UNION operator when used within MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/union.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms180026.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff821131.aspx

3.11- The SELECT INTO Statement:

The SELECT INTO statement in SQL is used to copy the result set of a SELECT statement into a new table that is created by the SELECT INTO statement. The core SQL of the SELECT INTO statement is shown below. Note that the "*table_name"* parameter is the name of the table from which you want to select the records, the "*new_table_name"* parameter is the name of the new table to create and into which you want to copy the selected records, the "*field_name"* parameter is the name of a field, and "*criteria"* is a value to find within the specified field. Braces { } are used to denote an area where a choice must be made, and the pipe symbol | is used to separate the choices that are available in that area.

SELECT {* | *field_name*, *field_name1*, *etc.*}
INTO *new_table_name*
FROM *table_name*
WHERE *field_name* = *criteria*

Note that there are some variations within the implementations of this statement within each RDBMS. You should check the documentation for your RDBMS to see what options are available for you to use. For example, you can use the SELECT INTO statement in MySQL 5.7 to copy the result set to either a new table or into a variable. However, you cannot copy the result set to a variable within SQL Server 2012. Also note that this type of a query is known as a "Make Table" query type within Access 2013. You can use the hyperlinks below to view the implementation of this statement within MySQL 5.7, SQL Server 2012, and Microsoft Access.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/select-into.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms188029.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff192059.aspx

3.12- The INSERT INTO SELECT Statement:

You can use the INSERT INTO SELECT statement to copy records from one table with a SELECT statement, and then append them into another existing table. When you copy the records from one table into another by using the INSERT INTO SELECT statement, the existing records within those tables will not be affected. The core SQL statement used to copy records between existing tables is shown below. Note that the "*table_name*" parameter is a name of a table within the database, the "*field_name*" parameter is a name of a field within a table, and "*criteria"* is a value that you want to find within the specified field.

INSERT INTO *table_name* (*field_name*, *field_name1*, *field_name2*, *etc.*)
SELECT {* | *field_name3*, *field_name4*, *etc.*}
FROM *table_name1*
WHERE *field_name3* = *criteria*

3.12- The INSERT INTO SELECT Statement:

There is some variation by RDBMS when using this statement, and you should check your documentation to see all of the options available. Note that in Access 2013, this type of query is referred to as an "Append Query." There are hyperlinks below that show the use of this statement in MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/insert-select.html

SQL Server 2012 (Specific example on the page of the more general INSERT statement):
http://technet.microsoft.com/en-us/library/ms174335.aspx#OtherTables

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff834799.aspx

3.13- Subqueries:

You can nest a query within another query to create a subquery. A subquery is simply a query that is placed within another query. Subqueries can be nested inside of many types of DML statements, such as the standard SELECT statement, a SELECT INTO statement, an INSERT INTO statement, an UPDATE statement, and the DELETE statement. You can also nest subqueries within other subqueries. However, there is often a limitation on the number of nested subqueries allowed by your RDBMS.

Also, you should exercise caution when using subqueries to ensure that their use does not degrade the performance of the query within which they are embedded. A poorly-designed subquery can negatively impact the performance of a query. You will need to check with your RDBMS documentation to see what limits are placed on the use of subqueries within a SELECT statement. Most systems will allow the use of a subquery in any part of a DML statement where you can use an expression. So, for example, you can use a subquery to create a type of calculated field by placing it into the SELECT field list within the primary SELECT statement. Alternately, they are found within the WHERE clause, determining which records are selected within the primary query.

Subqueries are placed within parentheses inside of the main SELECT statement. The core SQL of a subquery is shown below. The "*primary_query*" parameter simply notes where the primary SELECT statement is located. Note that the "*table_name*" parameter is the name of the table from which you want to select the records, the "*field_name*" parameter is the name of a field, and "*criteria*" is a criteria expression used for selection. Elements shown within brackets indicate that a choice must be made, and the choices shown are separated by the pipe symbol. Elements shown within braces { } are simply optional extensions of the SELECT clause that can be incorporated, if needed. Note that a subquery does allow for most clauses that can be incorporated within a primary SELECT statement.

```
primary_query
(
SELECT [ * | field_name, field_name1, etc. ]
FROM table_name
{ WHERE clause }
{ GROUP BY field_name, etc. }
{ HAVING criteria }
```

3.13- Subqueries- (cont'd.):

{ ORDER BY *field_name* [*DESC*] }
)
primary_query

Below are links to documentation that shows the abilities and limitations of using subqueries within MySQL 5.7, SQL Server 2012, and Access 2013. You can also find related links within these pages that also provide examples and related subquery information for each RDBMS.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/subqueries.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms189575.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff192664.aspx

ACTIONS- DATA MANIPULATION LANGUAGE

THE CORE SQL OF THE INSERT STATEMENT:

INSERT INTO *table_name* (*field_name*, *field_name1*, *field_name2*, *etc.*)
VALUES (*value*, *value1*, *value2*, *etc.*)

THE CORE SQL OF THE UPDATE STATEMENT:

UPDATE *table_name*
SET *field_name=update_value*, *field_name1=update_value1*, *etc.*
WHERE *field_name=existing_value*

THE CORE SQL OF THE DELETE STATEMENT:

DELETE FROM *table_name*
WHERE *field_name=delete_value*

THE CORE SQL OF THE SELECT STATEMENT:

1. To select specified fields from a table:
SELECT *field_name*, *field_name1*, *field_name2*, *etc.*
FROM *table_name*

2. To select specified fields from a table and only return unique record values:
SELECT DISTINCT *field_name*, *field_name1*, *field_name2*, *etc.*
FROM *table_name*

3. To select all fields from a table:
SELECT * FROM *table_name*

THE CORE SQL OF THE SELECT STATEMENT CONTAINING A WHERE CLAUSE:

SELECT *field_name*, *field_name1*, *etc.*
FROM *table_name*
WHERE *field_name* = *criteria*

ACTIONS- DATA MANIPULATION LANGUAGE

COMMON COMPARISON OPERATORS USED WITHIN THE WHERE CLAUSE:

Operator	Description
=	Equals. Used to find values that are equal to a value that you specify.
<> *or* !=	Not equal to. Used to select values that are NOT equal to a value you specify. The symbol used can change depending on the version of SQL implemented within your RDBMS.
>	Greater than. Used to find values that are greater than a value specified.
<	Less than. Used to find values that are less than a value specified.
>=	Greater than or equal to. Used to find values that are greater than or equal to a value specified.
<=	Less than or equal to. Used to find values that are less than or equal to a value specified.
BETWEEN	Used to find values including and between two values that are specified.
LIKE	Used to find values that match a pattern that is specified.
IN	Used to find values that match multiple values within a list of values that are specified.
AND	Used to join multiple selection criteria together. Selects records that match *both* criteria joined by the AND operator.
OR	Used to join multiple selection criteria together. Selects records that match *either* criteria joined together by the OR operator.

COMMON NOTATIONS USED FOR LITERAL VALUES AND WILDCARD CHARACTERS:

Notation	Use
‘ ‘ *or* “ “	Specifies a text value or text string. Sometimes used to denote any non-numeric value.
None	Numeric values do not use any notation. Numbers are signified by a lack of notation.
# #	Specifies a date/time value in Microsoft Access.
[ ]	Specifies a parameter or other field name value to use as the criteria in Microsoft Access. Example: [table.field]. Used to denote a set of initial characters to match when used in conjunction with wildcard characters in most other RDBMS. Example: ‘[a-d]%’
% *or* *	Wildcard character. Used to denote multiple unknown characters.
_ *or* ?	Wildcard character. Used to denote a single unknown character.

ACTIONS- Data Manipulation Language

THE CORE SQL OF THE SELECT STATEMENT CONTAINING AN ORDER BY CLAUSE:

SELECT *field_name*, *field_name1*, *etc.*
FROM *table_name*
WHERE *field_name* = *criteria*
ORDER BY *field_name* {ASC | DESC}, *field_name1* {ASC | DESC}, *etc.*

THE CORE SQL OF THE SELECT STATEMENT CONTAINING A GROUP BY CLAUSE:

SELECT *field_name*, *aggregate_function*(*field_name1*), [*etc.*]
FROM *table_name*
{ WHERE clause }
GROUP BY *field_name*, [*etc.*]
[HAVING criteria]
{ ORDER BY clause }

THE CORE SQL OF THE SELECT STATEMENT CONTAINING A JOIN CLAUSE:

SELECT *field_name*, *field_name1*, *field_name2*, *etc.*
FROM *table_name*
{ INNER JOIN | LEFT JOIN | RIGHT JOIN | FULL JOIN } *table_name1*
ON *table_name*.*primary_key* = *table_name1*.*foreign_key*

THE CORE SQL OF THE UNION OPERATOR:

1. To create a UNION of two SELECT statements that returns only unique records values:
select_statement
UNION
select_statement1

2. To create a UNION of two SELECT statements that returns ALL records values within the statements:
select_statement
UNION ALL
select_statement1

THE CORE SQL OF THE SELECT INTO STATEMENT:

SELECT {* | *field_name*, *field_name1*, *etc.*}
INTO *new_table_name*
FROM *table_name*
WHERE *field_name* = *criteria*

THE CORE SQL OF THE INSERT INTO SELECT STATEMENT:

INSERT INTO *table_name* (*field_name*, *field_name1*, *field_name2*, *etc.*)
SELECT {* | *field_name3*, *field_name4*, *etc.*}
FROM *table_name1*
WHERE *field_name3* = *criteria*

THE CORE SQL OF A SUBQUERY WITHIN A PRIMARY QUERY:

primary_query
(
SELECT [* | *field_name*, *field_name1*, *etc.*]
FROM *table_name*
{ WHERE clause }
{ GROUP BY *field_name*, *etc.* }
{ HAVING *criteria* }
{ ORDER BY *field_name* [*DESC*] }
)
primary_query

EXERCISES-
DATA MANIPULATION LANGUAGE

Purpose:

1. To be able to execute Data Manipulation Language (DML) statements in SQL on a database.

Exercises:

1. Ensure that you have completed the Exercise at the end of the previous chapter.
2. Double-click the “sqlite” file that you extracted in the Exercise at the end of Chapter 1 to open the “SQLite” command shell application within either a Command Prompt window within a Windows operating system or within a Terminal window within the Mac operating system.
3. Enter the commands within these Exercises into those windows in your respective operating system.
4. Type the following command line into either the Command Prompt window or Terminal window to create and open a new permanent database file called “test.db” within SQLite.
5. **.open test.db**
6. Press the “Enter” key on your keyboard to log into the database file. You should now see your cursor appear in front of the words ‘sqlite>’ within the window. You will enter the following commands after that prompt.
7. Now you will insert new data into the tables you created in the Exercise at the end of the last chapter. Type the following statements into the window to insert data into the “Employees” table. Press the “Enter” key on your keyboard after typing each statement.
8. **INSERT INTO Employees (FirstName, LastName) VALUES ('Joe', 'Smith');**
9. **INSERT INTO Employees (FirstName, LastName) VALUES ('Fred', 'Smith');**
10. **INSERT INTO Employees (FirstName, LastName) VALUES ('Mary', 'Jones');**
11. **INSERT INTO Employees (FirstName, LastName) VALUES ('Greg', 'King');**
12. **INSERT INTO Employees (FirstName, LastName) VALUES ('Jack', 'Wells');**
13. Now you will insert new data into the “Customers” table you created in the Exercise at the end of the last chapter. Type the following statements into the window to insert data into the “Customers” table. Press the “Enter” key on your keyboard after typing each statement.
14. **INSERT INTO Customers (CompanyName, Address, City, State, Zip) VALUES ('Compco', '100 Main St.', 'Lansing', 'MI', '48912');**
15. **INSERT INTO Customers (CompanyName, Address, City, State, Zip) VALUES ('The Auto Shop', '550 Elm St.', 'Holt', 'MI', '48842');**
16. **INSERT INTO Customers (CompanyName, Address, City, State, Zip) VALUES ('Capital Consulting', '125 Crescent', 'Lansing', 'MI', '48912');**
17. **INSERT INTO Customers (CompanyName, Address, City, State, Zip) VALUES ('The Food Store', '625 Lincoln St.', 'Ionia', 'MI', '48846');**
18. **INSERT INTO Customers (CompanyName, Address, City, State, Zip) VALUES ('Flowers and More', '233 E. Grand River', 'East Lansing', 'MI', '48823');**
19. **INSERT INTO Customers (CompanyName, Address, City, State, Zip) VALUES ('Kandy Korner', '180 Hagadorn', 'East Lansing', 'MI', '48823');**
20. **INSERT INTO Customers (CompanyName, Address, City, State, Zip) VALUES ('Rodgers Roofing', '250 Pine St.', 'Lansing', 'MI', '48821');**
21. Now you will insert new data into the “Items” table. Type the following statements into the window to insert the data. Press the “Enter” key on your keyboard after typing each statement.
22. **INSERT INTO Items (ItemName) VALUES ('Pens');**
23. **INSERT INTO Items (ItemName) VALUES ('Paperclips');**
24. **INSERT INTO Items (ItemName) VALUES ('Markers');**

Exercises- (cont'd.):

25. **INSERT INTO Items (ItemName) VALUES ('Paper');**
26. **INSERT INTO Items (ItemName) VALUES ('Palm Pilots');**
27. **INSERT INTO Items (ItemName) VALUES ('Notebooks');**
28. **INSERT INTO Items (ItemName) VALUES ('Manilla File Folders');**
29. **INSERT INTO Items (ItemName) VALUES ('Filing Cabinets');**
30. **INSERT INTO Items (ItemName) VALUES ('Hanging File Folders');**
31. **INSERT INTO Items (ItemName) VALUES ('Staples');**
32. Now you will insert new data into the "Sales" table. Type the following statements into the window to insert the data. Press the "Enter" key on your keyboard after typing each statement.
33. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (1, 2, '2007-01-01');**
34. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (1, 2, '2007-01-02');**
35. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (1, 2, '2007-01-03');**
36. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (2, 1, '2007-01-01');**
37. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (2, 3, '2007-01-02');**
38. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (2, 3, '2007-01-03');**
39. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (3, 4, '2007-01-01');**
40. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (3, 2, '2007-01-15');**
41. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (4, 4, '2007-01-10');**
42. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (4, 2, '2007-01-10');**
43. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (5, 6, '2007-01-21');**
44. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (5, 6, '2007-01-22');**
45. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (4, 7, '2007-01-25');**
46. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (5, 5, '2007-01-25');**
47. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (4, 5, '2007-01-15');**
48. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (1, 4, '2007-01-10');**
49. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (1, 3, '2007-01-24');**
50. **INSERT INTO Sales (EmployeeID, CustomerID, Saledate) VALUES (2, 2, '2007-01-20');**
51. Now you will insert new data into the "SalesDetails" table. Type the following statements into the window to insert the data. Press the "Enter" key on your keyboard after typing each statement.
52. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (1, 1, 1.50, 2);**
53. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (2, 2, 2.00, 5);**
54. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (3, 3, 5.00, 10);**
55. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (4, 4, 20.00, 1);**
56. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (5, 5, 99.00, 1);**
57. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (6, 6, 0.50, 10);**
58. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (6, 7, 1.00, 150);**
59. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (6, 8, 125.00, 1);**
60. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (6, 9, 1.25, 20);**
61. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (7, 10, 2.00, 2);**
62. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (8, 8, 125.00, 2);**
63. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (9, 1, 1.50, 3);**
64. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (9, 2, 2.00, 2);**
65. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (9, 4, 20.00, 6);**
66. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (10, 1, 1.50, 10);**
67. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (11, 8, 125.00, 2);**

Exercises- (cont'd.):

68. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (12, 9, 1.25, 6);**
69. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (13, 10, 2.00, 5);**
70. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (14, 4, 20.00, 5);**
71. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (15, 6, 0.50, 11);**
72. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (16, 5, 99.00, 3);**
73. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (17, 1, 1.50, 4);**
74. **INSERT INTO SalesDetails (SaleID, ItemID, Price, Quantity) VALUES (18, 5, 99.00, 2);**
75. Now you will create a query that will display the total amount of each sale by "SaleID" for each employee by "Employee ID." To do this, enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
76. **SELECT Sales.EmployeeID, Employees.FirstName, Employees.LastName, SalesDetails.SaleID, Sales.Saledate, SUM([Price]*[Quantity]) AS SaleAmount FROM Sales INNER JOIN SalesDetails ON Sales.SaleID = SalesDetails.SaleID INNER JOIN Employees ON Employees.EmployeeID = Sales.EmployeeID GROUP BY Sales.EmployeeID, Employees.FirstName, Employees.LastName, SalesDetails.SaleID, Sales.Saledate;**
77. Now you will create a query that will display the total amount sold for each product. Enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
78. **SELECT Items.ItemName, SUM([Price]*[Quantity]) AS Amount FROM Sales INNER JOIN SalesDetails ON Sales.SaleID = SalesDetails.SaleID INNER JOIN Items ON Items.ItemID = SalesDetails.ItemID GROUP BY Items.ItemName;**
79. Now you will create a query that displays the total amount sold of each item to each customer ID for each date. Enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
80. **SELECT Sales.CustomerID, Sales.Saledate, Items.ItemName, SUM([Price]*[Quantity]) AS Amount FROM Sales INNER JOIN SalesDetails ON Sales.SaleID = SalesDetails.SaleID INNER JOIN Items ON Items.ItemID = SalesDetails.ItemID GROUP BY Sales.CustomerID, Sales.Saledate, Items.ItemName;**
81. Now you will create a query that will show all of the records within the "Customers" table. Enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
82. **SELECT * FROM Customers;**
83. Now you will update the "Zip" field for the record with a "CompanyName" of "Rodgers Roofing" from "48821" to "48912" by using the UPDATE statement. Enter the following UPDATE statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
84. **UPDATE Customers SET Zip='48912' WHERE CompanyName='Rodgers Roofing';**
85. Now you will display the "Rodgers Roofing" record within the "Customers" table by using the SELECT statement so that you can verify the update. Enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
86. **SELECT * FROM Customers WHERE CompanyName = 'Rodgers Roofing';**
87. You can close the Terminal or Command Prompt window at this point, if desired. Be sure to keep the 'test.db' file that you have created, as you will need it for the upcoming Exercises at the end of each chapter. *The Exercises at the end of each chapter build upon one another and must be completed in sequential order.*

CHAPTER 4- DATA CONTROL LANGUAGE

4.1- The CREATE USER and CREATE ROLE Statements:

Data Control Language (DCL) is used within SQL to grant or deny privileges, which are permissions to perform specific tasks within the database, to users. This lets the database administrator or owner choose which users can perform which actions upon selected database objects. Note that not every RDBMS will fully implement DCL within their SQL. Some database systems, like SQLite, do not implement any DCL statements, as it is a single-user desktop database system. Other database programs, like Microsoft Access 2013, use a separate security mechanism to determine user access and privileges not tied to SQL. You must check your RDBMS documentation to find out the level of DCL implementation that exists within its SQL, if at all. DCL statements are used by the database administrator, or owner, to create users and roles within the database and to grant or revoke privileges to and from these users and roles. In some more complex relational database management systems (RDBMS), a "role" may be created in addition to creating users. Each role is then assigned its own set of access privileges. Then individual users can be assigned to the roles that have been created to acquire their necessary database privileges. Doing this can simplify database privilege assignment for large databases that contain many users. Some relational database management systems also come with pre-defined roles to which you can assign users.

The first statement to examine is the CREATE USER statement, which is used to create a user within the database. This statement is technically classified as a Data Definition Language (DDL) statement, as it creates a database object. It is only added to this chapter because of its use in conjunction with the DCL statements. The core SQL of the CREATE USER statement is shown below. Note that the "*user_name"* parameter is the name of the user that is to be created.

CREATE USER *user_name*

The CREATE ROLE statement is used to create a role within the database. It is also classified as Data Definition Language (DDL), as it creates a database object. It too, is only added to this chapter because of its use in conjunction with the DCL statements. The core SQL of the CREATE ROLE statement is shown below. Note that the "*role_name"* parameter is the name of the role that is to be created.

CREATE ROLE *role_name*

The creation of users and roles within each RDBMS will vary, and you should ensure that you check the documentation available. The CREATE USER statement, as implemented in MySQL 5.7 and SQL Server 2012 is shown in the first two hyperlinks below. Note the CREATE USER command in Microsoft Access 2013 is only used with the ANSI-92 compliant databases. Normally in Access 2013 databases, security is handled by an entirely different process. You will also find the CREATE ROLE syntax statement within SQL Server 2012 shown at the bottom of the listing of hyperlinks below. Note that there is no CREATE ROLE or equivalent statement within MySQL 5.7.

MySQL 5.7 (CREATE USER):
http://dev.mysql.com/doc/refman/5.7/en/create-user.html

SQL Server 2012 (CREATE USER):
http://technet.microsoft.com/en-us/library/ms173463.aspx

Access 2013 (CREATE USER):
http://msdn.microsoft.com/en-us/library/office/ff194914.aspx

4.1- The CREATE USER and CREATE ROLE Statements- (cont'd.):

SQL Server 2012 (CREATE ROLE):
http://technet.microsoft.com/en-us/library/ms187936.aspx

4.2- Privileges:

Privileges provide various levels of permission to create, edit, and delete database objects. The exact types of permissions that can be granted will vary depending upon the relational database management system that you are using. You must check your specific RDBMS documentation to be sure which privileges are available. The following table lists commonly used privileges that will often be implemented within relational database management systems that support DCL statements.

Privilege	Description
CREATE	Allows users to create objects. Often cited as CREATE *db_object*, where the "*db_object"* parameter is the type of database object that the privilege allows them to create. Also cited as CREATE ALL to allow the user to create any type of database object.
ALTER	Allows users to alter objects. Often cited as ALTER *db_object*, where the "*db_object"* parameter is the type of database object that the privilege allows them to alter. Also cited as ALTER ALL to allow the user to alter any type of database object.
DROP	Allows users to delete objects. Often cited as DROP *db_object*, where the "*db_object"* parameter is the type of database object that the privilege allows them to delete. Also cited as DROP ALL to allow the user to delete any type of database object.
INSERT	Allows the user to insert records into a table.
UPDATE	Allows the user to update records within a table.
DELETE	Allows the user to delete records within a table.
SELECT	Allows the user to select records within a table.
EXECUTE	Allows the user to execute a stored procedure or function

4.3- The GRANT Statement:

The GRANT statement is a DCL statement used by the database administrator or owner to grant privileges to other database users and roles. The core SQL of the GRANT statement is shown below. Note that the "*privilege"* parameter is the name of the specific privilege to give, the "*db_object"* parameter is the name of the database object to which the privilege is given, the "*user_name"* parameter is the name of the user to which the privilege is given, and the "*role_name"* parameter is the name of the role to which the privilege is given. Options shown in braces { } indicate that you must make a choice. The pipe symbol | is used to separate the choices shown. Optional clause are shown as underlined text.

GRANT *privilege*
ON *db_object*

4.3- The GRANT Statement:

TO { *user_name* | *role_name* | PUBLIC }
WITH GRANT OPTION

Note that the PUBLIC keyword is used within the statement to grant a named privilege to all users of the database. The *optional* WITH GRANT OPTION clause is used to allow the user or role to assign the privilege that they are being given to other users, and should be used with caution. For example, assume a user is given a privilege using the WITH GRANT OPTION clause and then they GRANT that privilege to other users. If you later REVOKE the original user's privilege, the users to whom the privilege was granted will still have access unless there is a CASCADE clause available within the REVOKE statement for your RDBMS and it is used when issuing the REVOKE statement. Use caution if choosing to apply the WITH GRANT OPTION clause.

The GRANT statement, as implemented within MySQL 5.7, SQL Server 2012, and Access 2013 is shown at the web pages referenced by the following hyperlinks.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/grant.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms187965.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff193820.aspx

4.4- The REVOKE Statement:

The REVOKE statement is a DCL statement used by the database administrator or owner to revoke privileges from database users and roles. The core SQL of the REVOKE statement is shown below. Note that the "*privilege"* parameter is the name of the specific privilege to remove, the "*db_object"* parameter is the name of the database object from which the privilege is removed, the "*user_name"* parameter is the name of the user from which the privilege is removed, and the "*role_name"* parameter is the name of the role from which the privilege is removed. Options shown in braces { } indicate that you must make a choice. The pipe symbol | is used to separate the choices shown.

REVOKE *privilege*
ON *db_object*
FROM { *user_name* | *role_name* | PUBLIC }

You should check your RDBMS documentation to see if the REVOKE statement, when used in conjunction with the PUBLIC clause, will also revoke the privilege from the object owner or database administrator. If it does, you may need to specify the owner's name in several specific GRANT statements following a general REVOKE. Be careful when using the PUBLIC clause with the REVOKE statement. You should check how a PUBLIC clause within a GRANT or REVOKE statement interacts with individually specified GRANT and REVOKE statements that were previously issued. In some RDBMS implementations, they will override the previous individual statements and in others they will ignore them.

The REVOKE statement, as implemented within MySQL 5.7, SQL Server 2012, and Access 2013 is

4.4- The REVOKE Statement- (cont'd.):

shown at the web pages referenced by the following hyperlinks.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/revoke.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms187728.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff195272.aspx

4.5- The ALTER USER and ALTER ROLE Statements:

You can use the ALTER USER statement in SQL to alter information about a user, which may include the user's database password, schema or database access, and login information. Note that this statement is technically classified as Data Definition Language (DDL), as it defines the objects within the database. It is only included within this chapter because of its use in conjunction with the DCL statements. The exact user aspects that can be altered will vary depending on the RDBMS used, and you should check your documentation to see what aspects of the user can be altered. The core SQL of the ALTER USER statement is shown below. Note that the "*user_name*" parameter is the user name of the user to alter and the "*alterations*" parameter is the specific alterations you would like to apply. Note that the WITH clause is shown within braces as it is not universally applied amongst RDBMS implementations.

ALTER USER *user_name*
{ WITH } *alterations*

You can also alter existing roles within your RDBMS to add and remove users from being associated with that role, to rename a role, or to change other attributes of the role in some way. The exact abilities that are provided by the ALTER ROLE statement will vary quite a bit between RDBMS implementations and you must check your RDBMS documentation to see what options are available. Note that this statement is also technically classified as Data Definition Language (DDL), as it defines the objects within the database. It is only included within this chapter because of its use in conjunction with the DCL statements. The core SQL of the ALTER ROLE statement is shown below. Note that the "*role_name*" parameter is the name of the role to alter and the "*alterations*" parameter is the specific alterations you would like to apply. Note that the WITH clause is shown within braces as it is not universally applied amongst RDBMS implementations.

ALTER ROLE *role_name*
{ WITH } *alterations*

Following are hyperlinks that show documentation for the implementation of ALTER USER within MySQL 5.7, SQL Server 2012, and Access 2013. You will also find documentation on the implementation of ALTER ROLE within SQL Server 2012. Remember that MySQL 5.7 does not support roles in its SQL.

4.5- The ALTER USER and ALTER ROLE Statements- (cont'd.):

MySQL 5.7 (ALTER USER):
http://dev.mysql.com/doc/refman/5.7/en/alter-user.html

SQL Server 2012: (ALTER USER):
http://technet.microsoft.com/en-us/library/ms176060.aspx

Access 2013: (ALTER USER):
http://msdn.microsoft.com/en-us/library/office/ff197012.aspx

SQL Server 2012: (ALTER ROLE):
http://technet.microsoft.com/en-us/library/ms189775.aspx

4.6- The DROP USER and DROP ROLE Statements:

You can use the DROP USER statement in SQL to delete a user from the database. Note that this statement is technically classified as Data Definition Language (DDL), as it defines the objects within the database. It only included within this chapter because of its use in conjunction with the DCL statements. The core SQL of the DROP USER statement is shown below. Note that the "*user_name"* parameter is the user name of the user to delete.

DROP USER *user_name*

You can delete existing roles within your database by using the DROP ROLE statement. Note that this statement is technically classified as Data Definition Language (DDL), as it defines the objects within the database. It is only included within this chapter because of its use in conjunction with the DCL statements. The core SQL of the DROP ROLE statement is shown below. Note that the "*role_name"* parameter is the name of the role to delete.

DROP ROLE *role_name*

Below are hyperlinks that show documentation for the implementation of DROP USER within MySQL 5.7, SQL Server 2012 and Access 2013. You will also find documentation on the implementation of DROP ROLE within SQL Server 2012. Remember that MySQL 5.7 does not support roles in its SQL.

MySQL 5.7 (DROP USER):
http://dev.mysql.com/doc/refman/5.7/en/drop-user.html

SQL Server 2012 (DROP USER):
http://technet.microsoft.com/en-us/library/ms189438.aspx

Access 2013 (DROP USER):
http://msdn.microsoft.com/en-us/library/office/ff193192.aspx

SQL Server 2012: (DROP ROLE):
http://technet.microsoft.com/en-us/library/ms174988.aspx

ACTIONS- Data Control Language

THE CORE SQL OF THE CREATE USER STATEMENT:

CREATE USER *user_name*

THE CORE SQL OF THE CREATE ROLE STATEMENT:

CREATE ROLE *role_name*

COMMON RDBMS PRIVILEGES:

Privilege	Description
CREATE	Allows users to create objects. Often cited as CREATE *db_object*, where the "*db_object*" parameter is the type of database object that the privilege allows them to create. Also cited as CREATE ALL to allow the user to create any type of database object.
ALTER	Allows users to alter objects. Often cited as ALTER *db_object*, where the "*db_object*" parameter is the type of database object that the privilege allows them to alter. Also cited as ALTER ALL to allow the user to alter any type of database object.
DROP	Allows users to delete objects. Often cited as DROP *db_object*, where the "*db_object*" parameter is the type of database object that the privilege allows them to delete. Also cited as DROP ALL to allow the user to delete any type of database object.
INSERT	Allows the user to insert records into a table.
UPDATE	Allows the user to update records within a table.
DELETE	Allows the user to delete records within a table.
SELECT	Allows the user to select records within a table.
EXECUTE	Allows the user to execute a stored procedure or function

THE CORE SQL OF THE GRANT STATEMENT:

GRANT *privilege*
ON *db_object*
TO { *user_name* | *role_name* | PUBLIC }
WITH GRANT OPTION

THE CORE SQL OF THE REVOKE STATEMENT:

REVOKE *privilege*
ON *db_object*
FROM { *user_name* | *role_name* | PUBLIC }

THE CORE SQL OF THE ALTER USER STATEMENT:

ALTER USER *user_name*
 { WITH } *alterations*

THE CORE SQL OF THE ALTER ROLE STATEMENT:

ALTER ROLE *role_name*
 { WITH } *alterations*

THE CORE SQL OF THE DROP USER STATEMENT:

DROP USER *user_name*

THE CORE SQL OF THE DROP ROLE STATEMENT:

DROP ROLE *role_name*

EXERCISES-
DATA CONTROL LANGUAGE

Purpose:

1. None.

Exercises:

1. As SQLite is a self-contained database file system, there are no users or system privileges available other than the file system privileges on your particular operating system. Therefore, there are no exercises for this chapter.

CHAPTER 5- Transaction Control Language

5.1- The TRANSACTION Statement

5.1- The TRANSACTION Statement:

The Transaction Control Language (TCL) statements within SQL are the statements used to control a set of Data Manipulation Language (DML) statements that are executed as a single unit called a transaction. TCL ensures that either all of the changes made by the DML statements within the transaction are committed to the database or none of them are. This can help to prevent **partial** changes from a series of statements from being accomplished. For example, if you wanted to run a series of DML statements on a database without TCL statements and you had a power outage or system crash occur in the middle of the set, you may only have some of those changes applied to the database. TCL statements ensure that all of the statements are successfully made within the database or that none of them are.

Note that while many of the major commands within TCL are fairly standard, there is quite a bit of variation in the execution of the transactions within each RDBMS. You should check your RDBMS documentation to see what options are available for transactions within your system.

In SQL, the BEGIN TRANSACTION or START TRANSACTION statement is used to initiate a transaction within an RDBMS. If the transaction statements that follow fail to execute, then the changes that were made by the statements can be set back to this point. The exact command used to initiate a transaction will vary by RDBMS, but is most commonly either BEGIN TRANSACTION or START TRANSACTION. The core SQL of this statement is shown below. Note that the "*work*" parameter represents the SQL statements that will be executed within the transaction and clauses shown separated by the pipe symbol indicate that you must select a choice.

BEGIN | START TRANSACTION
work
COMMIT | ROLLBACK

Note that this is a very simplified core of the SQL involved in creating transactions within multiple relational database management systems (RDBMS) and most vendors will have more complex options available for their TRANSACTION statements. The COMMIT statement is used after the *work* has been completed to commit those changes to the database. The ROLLBACK statement is used to undo any changes to the database caused by the *work* back to the point at which the TRANSACTION statement was initiated.

Below are hyperlinks to web pages that explain the use of the TRANSACTION statement within MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/commit.html

SQL Server 2012 ("Transaction Statements"- specific statements are within the "In This Section" area):
http://technet.microsoft.com/en-us/library/ms174377.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff193241.aspx

ACTIONS-
TRANSACTION CONTROL LANGUAGE

<u>THE CORE SQL OF THE TRANSACTION STATEMENT:</u>

BEGIN | START TRANSACTION
work
COMMIT | ROLLBACK

EXERCISES-
Transaction Control Language

Purpose:

1. To be able to use SQL to make database changes within a transaction for additional data security.

Exercises:

1. Ensure that you have at least completed the Exercise at the end of Chapter 3.
2. Double-click the "sqlite" file that you extracted in the Exercise at the end of Chapter 1 to open the "SQLite" command shell application within either a Command Prompt window within a Windows operating system or within a Terminal window within the Mac operating system.
3. Enter the commands within these Exercises into those windows in your respective operating system.
4. Type the following command line into either the Command Prompt window or Terminal window to create and open a new permanent database file called "test.db" within SQLite.
5. **.open test.db**
6. Press the "Enter" key on your keyboard to log into the database file. You should now see your cursor appear in front of the words 'sqlite>' within the window. You will enter the following commands after that prompt.
7. Enter the following statement to begin a new transaction. Press the "Enter" key on your keyboard when you have finished entering the statement.
8. **BEGIN TRANSACTION;**
9. Now you will execute a SELECT statement to examine the records within the "Customers" table. Enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
10. **SELECT * FROM Customers;**
11. Now you will delete a record within the "Customers" table. Enter the following DELETE statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
12. **DELETE FROM Customers WHERE CompanyName = 'Rodgers Roofing';**
13. Now you will execute a SELECT statement to examine the change within the "Customers" table. Enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
14. **SELECT * FROM Customers;**
15. Now you will use the ROLLBACK statement to rollback the entire transaction back to how the database was when the BEGIN TRANSACTION statement was issued, thus undoing the deletion form the table. Enter the following statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
16. **ROLLBACK;**
17. Now examine the records within the "Customers" table to see that the records are intact once again. Enter the following SELECT statement into the window and then press the "Enter" key on your keyboard when finished to view the results.
18. **SELECT * FROM Customers;**
19. You can close the Terminal or Command Prompt window at this point, if desired. Be sure to keep the 'test.db' file that you have created, as you will need it for the upcoming Exercises at the end of each chapter. *The Exercises at the end of each chapter build upon one another and must be completed in sequential order.*

CHAPTER 6-
SQL FUNCTIONS AND ALIASES

6.1- Understanding SQL Functions:

Functions serve a wide range of purposes within SQL. In lesson "3.8- The GROUP BY Clause and Aggregate Functions," you learned how you can use aggregate functions to perform calculations on grouped values within a query. That is one very common use of functions within SQL. However, there are other types of functions that can serve other purposes within SQL, as well. In this lesson, you will learn about the various types of functions within SQL and how they can be used within queries.

A function is simply a computation that is executed by invoking a single-word command that returns a value. While some functions, such as SUM, are universally accepted by all RDBMS, there are also some variations of function names between RDBMS. For example, while almost all RDBMS can provide you with the current date in a function, the name of the function used to accomplish that is SYSDATE within MySQL and Oracle, while it is named GETDATE within SQL Server and DATE within Access. Therefore, you need to check the documentation of your specific RDBMS to know which function names are available to use.

All functions can be classified as being either deterministic or non-deterministic. A deterministic function will always return the same result whenever it is used with the same set of inputs. A non-deterministic function may return a different result when used with the same set of inputs. For example, the SUM function is a deterministic function, as it will always return the same result if given the same input parameters or variables. In contrast, the SYSDATE or GETDATE functions, which are used to retrieve the current date in MySQL or SQL Server, are non-deterministic functions as they will return a different answer when used, even though they use a consistent input.

One reason to know whether a function is deterministic or non-deterministic is because different RDBMS will allow for slightly different use of deterministic and non-deterministic functions. For example, in SQL Server 2012 you cannot create an index on a computed column that references non-deterministic functions. You will need to check the documentation of your specific RDBMS to determine what restrictions may apply to the use of deterministic and non-deterministic functions.

In lesson "3.8- The GROUP BY Clause and Aggregate Functions," you learned how aggregate functions perform calculations on grouped values. Aggregate functions are considered one of the primary types of functions within SQL. These functions perform a summarizing operation upon a group of values or inputs to return a summary value. In contrast to this, there are also scalar functions. A scalar function is performed upon a single value and returns a single value. For example, the SUM function is used to create a summary value from selected values within a field. However the LOWER function (called LCASE in Access), will simply return a single field value in all lowercase characters.

You can use scalar functions in many places within SQL. While aggregate functions are most often used in conjunction with the GROUP BY clause, unless it is the only value returned by a SELECT statement, scalar functions have more flexibility in where they can be used. You can use scalar functions within the SELECT clause of a SELECT statement to create calculated fields, the INSERT INTO clause of the INSERT statement to insert calculated values, and many other places within SQL. Below is a listing of hyperlinks that show the available scalar functions within MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7 (Function and Operator Reference):
http://dev.mysql.com/doc/refman/5.7/en/func-op-summary-ref.html

SQL Server 2012 (Scalar functions grouped by categories):
http://technet.microsoft.com/en-us/library/ms174318.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff835353.aspx

6.2- Calculated Fields and Column Aliases:

You can use scalar functions within SQL to create calculated fields within SELECT statements. This allows you to display the result of a calculation within a field in the result set of a query. The calculation can refer to columns within the tables selected in the FROM clause of the SELECT statement. When you create calculated fields, you may want to give the calculated field, also called a "calculated column," its own name so that the user will not see the calculated expression as the field name within the result set. You can change the displayed title of any field within a result set, calculated or not, by using a column alias. While calculated fields are most often given column aliases, you can apply an alias to any field in the result set. For example, they can be useful to simplify the display name in the result set of technical or overly-long field names within the source tables.

For example, assume that you are selecting a "QuantitySold" field and a "UnitPrice" field from a "Sales" table. You want the result set of the query to display these two columns with spaces within their names and you also want a third column to display the result of multiplying the values within the "QuantitySold" and "UnitPrice" columns for each record. You want the third calculated field in the result set to have the display name of "Sales Subtotal." The AS keyword is used to apply an alias to a field within the statement. You could create a simple query that would contain the following SQL.

```
SELECT QuantitySold AS 'Quantity Sold', UnitPrice AS 'Unit Price', QuantitySold * UnitPrice AS 'Sales
Subtotal'
FROM Sales
```

When creating the calculated field, enter the calculation to perform within the field listing of the SELECT clause within the SELECT statement. If you want the field to use a column alias, follow the calculation or field name with the AS keyword, and then the name of the column alias within either single or double quotation marks per your RDBMS specifications. The core SQL of creating calculated fields and column aliases is shown below. The "*table_name"* parameter is the name of the table from which you are extracting the records. The "*field_name"* parameter is the name of a field within the table referred to by the "*table_name"* parameter. The "*calculated_field"* parameter is the calculation that you want to return as a field within the result set. The "*alias_name"* parameter is the name that you want to give to the field.

SELECT *field_name*, *field_name1*, *etc.*, *calculated_field* AS '*alias_name*', *etc.*
FROM *table_name*

6.3- Table Aliases:

You can use table aliases in SQL to make referencing table names within complex SQL statements easier. For example, if you have a "CustomerID" field within a "Customers" table and also have a "CustomerID" field within a "Sales" table, you must use the dot notation reference of *table_name.field_name* for the "CustomerID" field within the SQL statement so that the database will know which "CustomerID" field you are referencing within the SQL statement. To simplify the SQL statement, you can apply an alias to a table name to simplify the *table_name.field_name* expressions.

The core SQL used to create a table alias within the FROM clause of the SELECT statement is shown below. The "*field_name"* parameter is the name of a field within a table referred to within the FROM clause. The "*table_name"* parameter is the name of a table within the database. The "*alias_name"* parameter is the alias name that you want to give to the table.

6.3- Table Aliases- (cont'd.):

SELECT *alias_name.field_name*, *alias_name.field_name1*, *alias_name.etc.*
FROM *table_name AS alias_name*

Here is an example from the Exercise at the end of this chapter that shows table aliases being used in SQLite. In this case, the tables named "Sales," "SalesDetails," and "Items" have been given the aliases of "S," "SD," and "I," respectively. Note that within a SELECT statement where the AS keyword is used to assign a table alias, you can substitute the table alias name for the associated table name when needed. This allows you to simplify the expression when dealing with SELECT statements that refer to tables with very long table names.

SELECT S.Saledate AS Date, I.ItemName AS Product, SUM([Price]*[Quantity]) AS Amount FROM Sales AS S INNER JOIN SalesDetails AS SD ON S.SaleID = SD.SaleID INNER JOIN Items AS I ON I.ItemID = SD.ItemID GROUP BY S.Saledate, I.ItemName;

ACTIONS-
SQL FUNCTIONS AND ALIASES

THE CORE SQL OF THE SELECT STATEMENT WHEN USED TO CREATE CALCULATED FIELDS AND COLUMN ALIASES:

SELECT *field_name*, *field_name1*, *etc.*, *calculated_field* AS '*alias_name*', *etc.*
FROM *table_name*

THE CORE SQL OF A SELECT STATEMENT USED TO CREATE A TABLE ALIAS:

SELECT *alias_name.field_name*, *alias_name.field_name1*, *alias_name.etc.*
FROM *table_name AS alias_name*

EXERCISES-
SQL FUNCTIONS AND ALIASES

Purpose:

1. To be able to create queries in SQL that use SQL functions, as well as column and table aliases.

Exercises:

1. Ensure that you have at least completed the Exercise at the end of Chapter 3.
2. Double-click the "sqlite" file that you extracted in the Exercise at the end of Chapter 1 to open the "SQLite" command shell application within either a Command Prompt window within a Windows operating system or within a Terminal window within the Mac operating system.
3. Enter the commands within these Exercises into those windows in your respective operating system.
4. Type the following command line into either the Command Prompt window or Terminal window to create and open a new permanent database file called "test.db" within SQLite.
5. **.open test.db**
6. Press the "Enter" key on your keyboard to log into the database file. You should now see your cursor appear in front of the words 'sqlite>' within the window. You will enter the following commands after that prompt.
7. You will now enable headers within the SQL result display, so that you can see the column alias names that you will provide in the following queries. To do this, enter the following command into the window. Note that this is a SQLite command, not an SQL command, and will *not* need a semi-colon at the end of the command to execute it.
8. **.headers ON**
9. Enter the following SELECT statement to create a query that will display the total sales of each product for each day. Note the use of both column and table aliases within this SELECT statement. Press the "Enter" key on your keyboard when you have finished entering the statement to view the results.
10. **SELECT S.Saledate AS Date, I.ItemName AS Product, SUM([Price]*[Quantity]) AS Amount FROM Sales AS S INNER JOIN SalesDetails AS SD ON S.SaleID = SD.SaleID INNER JOIN Items AS I ON I.ItemID = SD.ItemID GROUP BY S.Saledate, I.ItemName;**
11. Now enter the following SELECT statement to create a query that will display the total sales for each day. Once again note the use of table and column aliases within this statement. Press the "Enter" key on your keyboard when you have finished entering the statement to view the results.
12. **SELECT S.Saledate AS Date, SUM([Price]*[Quantity]) AS Amount FROM Sales AS S INNER JOIN SalesDetails AS SD ON S.SaleID = SD.SaleID GROUP BY S.Saledate;**
13. You can now disable headers within the SQL result display to hide the display of column titles within queries, if desired. To do this, enter the following command into the window. Note that this is a SQLite command, not an SQL command, and will *not* need a semi-colon at the end of the command to execute it.
14. **.headers OFF**
15. You can close the Terminal or Command Prompt window at this point, if desired. Be sure to keep the 'test.db' file that you have created, as you will need it for the upcoming Exercises at the end of each chapter. *The Exercises at the end of each chapter build upon one another and must be completed in sequential order.*

CHAPTER 7- VIEWS

7.1- About Views:

A view is a virtual table that is based on the results of a SELECT statement. They are very much like the result set of a query. They always display the most recent data from the underlying tables from which they are constructed because whenever a user queries a view, most RDBMS will recreate the view from the SELECT statement upon which the view is based. A view will often present data from joined base tables in a relational database, which stores data according to the specifications required by relational database design and the rules of normalization, in a more user-friendly format. It is also possible to query a view by using the name of the view within the FROM clause of the SELECT statement. This allows more sophisticated end-users to query and access the data they need to view for reports within the view, so that they will not need to access the base tables within the database.

You create views by using the CREATE VIEW statement in SQL. You can most often change the structure of an existing view by using the ALTER VIEW statement. You can delete a view by using the DROP VIEW statement. Note that all of these statements are considered Data Definition Language statements in SQL. However, since you must create a SELECT statement in order to define a view, they have been included in their own separate chapter within this course. In this chapter, you will examine how to implement views in SQL by using the CREATE VIEW, ALTER VIEW, and DROP VIEW statements. Note that these three statements are classified as being Data Definition Language (DDL) statements within SQL. They are only being discussed in their own separate chapter from the rest of the DDL statements because you can only create a view after you have learned how to use the SELECT statement in SQL.

7.2- The CREATE VIEW Statement:

You use the CREATE VIEW statement to create a view of the data within your base tables as a separate virtual table within your database. The core SQL of the CREATE VIEW statement is shown below. The "*view_name*" parameter is the name of the new view, and the "*select_statement*" parameter is the SELECT statement used to define which records and fields that will appear within the view.

CREATE VIEW *view_name* AS
select_statement

There is some variety of view implementation within relational database management systems. You will need to check the specific documentation of your RDBMS to familiarize yourself with any restrictions placed upon views within the application. For example, you can often use an ORDER BY clause when creating views in MySQL 5,7, but it is generally not permitted within SQL Server 2012. Below are hyperlinks to web pages that explain the implementation of the CREATE VIEW statement in MySQL 5.7, SQL Server 2012, and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/create-view.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms187956.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff836312.aspx

7.3- The ALTER VIEW Statement:

You can redefine the display of data within a named view by issuing the ALTER VIEW statement in SQL. The syntax of this statement is almost exactly the same as the CREATE VIEW statement. The core SQL of the ALTER VIEW statement is shown below. The "*view_name"* parameter is the name of the view to edit and the "*select_statement"* parameter is the SELECT statement used to define which records and fields will appear within the edited view.

ALTER VIEW *view_name* AS
select_statement

Note that you should check your RDBMS documentation for any specific variations that you can use in conjunction with the ALTER VIEW statement. For example, you can use the CREATE OR REPLACE VIEW command to act as a substitute for the ALTER VIEW command within MySQL 5.7. Below are hyperlinks to web pages that explain the implementation of the ALTER VIEW statement in MySQL 5.7 and SQL Server 2012. Note that Access 2013 does not support the ALTER VIEW command. You can issue a DROP VIEW statement followed by a new CREATE VIEW statement instead, if needed.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/alter-view.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms173846.aspx

7.4- The DROP VIEW Statement:

The DROP VIEW statement is used to remove a named view from a database. The core SQL of the DROP VIEW statement is shown below. The "*view_name"* parameter is the name of the view to delete.

DROP VIEW *view_name*

Below are hyperlinks to web pages that explain the implementation of the DROP VIEW statement in MySQL 5.7, SQL Server 2012 and Access 2013.

MySQL 5.7:
http://dev.mysql.com/doc/refman/5.7/en/drop-view.html

SQL Server 2012:
http://technet.microsoft.com/en-us/library/ms173492.aspx

Access 2013:
http://msdn.microsoft.com/en-us/library/office/ff821409.aspx

ACTIONS- VIEWS

THE CORE SQL OF THE CREATE VIEW STATEMENT:

CREATE VIEW *view_name* AS
select_statement

THE CORE SQL OF THE ALTER VIEW STATEMENT:

ALTER VIEW *view_name* AS
select_statement

THE CORE SQL OF THE DROP VIEW STATEMENT:

DROP VIEW *view_name*

EXERCISES- VIEWS

Purpose:

1. To be able to use SQL to create, query, and delete a view within a database.

Exercises:

1. Ensure that you have at least completed the Exercise at the end of Chapter 3.
2. Double-click the “sqlite” file that you extracted in the Exercise at the end of Chapter 1 to open the “SQLite” command shell application within either a Command Prompt window within a Windows operating system or within a Terminal window within the Mac operating system.
3. Enter the commands within these Exercises into those windows in your respective operating system.
4. Type the following command line into either the Command Prompt window or Terminal window to create and open a new permanent database file called “test.db” within SQLite.
5. **.open test.db**
6. Press the “Enter” key on your keyboard to log into the database file. You should now see your cursor appear in front of the words ‘sqlite>’ within the window. You will enter the following commands after that prompt.
7. Now you will create a view the displays the total amount of each sale for all employees. To do this, enter the statement below into the window and then press the “Enter” key on your keyboard when you are finished.
8. **CREATE VIEW EmployeeSalesView AS SELECT Sales.EmployeeID, Employees.FirstName, Employees.LastName, SalesDetails.SaleID, Sales.Saledate, SUM([Price]*[Quantity]) AS SaleAmount FROM Sales INNER JOIN SalesDetails ON Sales.SaleID = SalesDetails.SaleID INNER JOIN Employees ON Employees.EmployeeID = Sales.EmployeeID GROUP BY Sales.EmployeeID, Employees.FirstName, Employees.LastName, SalesDetails.SaleID, Sales.Saledate;**
9. Now you will create a query on the values shown within that view. This query will show all sales for the employee named “Joe Smith.” To do this, enter the statement below into the window and then press the “Enter” key on your keyboard when you are finished.
10. **SELECT * FROM EmployeeSalesView WHERE FirstName = 'Joe' AND LastName = 'Smith';**
11. Now you will create a query based on the view that shows all sales where the total amount sold was greater than or equal to $100. To do this, enter the statement below into the window and then press the “Enter” key on your keyboard when you are finished.
12. **SELECT * FROM EmployeeSalesView WHERE SaleAmount >= 100;**
13. Now you will delete the view that you just created from the database. To do this, you will use the DROP VIEW statement. Enter the statement below into the window and then press the “Enter” key on your keyboard when you are finished.
14. **DROP VIEW EmployeeSalesView;**
15. If you wish to delete the database file, you can determine its location within your file system by entering the following SQLite command into the window and then pressing the “Enter” key on your keyboard. Note that you will need to terminate your current interface with the file before you can delete it by closing the Terminal or Command Prompt window.
16. **.databases**
17. You can close the Terminal or Command Prompt window at this point. You have completed the Exercises within this tutorial.

Mastering Introductory SQL Made Easy™
Index of Common Tasks

The following index uses a "How do I…?" question format to direct users to the associated resources within this manual. To find out how to perform a specific task, find the task that you are interested in performing in SQL within the index below. You can then find the SQL needed to accomplish the selected task on the associated page or pages.