Chapter Eight
Structured Query Language

Our brokerage database is very easy to query – thanks to the wonderful query *design grid* that Access offers. The design grid gives everyone, beginners included, the tool they need to perform some remarkably complex queries. Unfortunately, there are plenty of times when you need answers from the database, but won't be sitting in front of it. If you aren't using Access to interface with your database, you can't use the design grid.

Remember, one of our goals is to provide a web interface to the database. In this case, the database will be sitting on a web server (it could be in Tibet!) and there will be no way of directly accessing the database. The details of how we'll remotely access the database will have to wait for Chapter 11 – it's a pretty clever stunt.

This chapter will provide the tools you need when talking to the Brokerage database – or any database for that matter. It will be a few chapters before you absolutely need this material. I'm showing it to you now because queries are fresh in you mind…

Chapter Outline:

8.1 Background

Every database speaks a common language – something called Structured Query Language. Computer types just use the acronym SQL. Pronounce it as "sequel" or just say the letters. It doesn't matter how you say it, computer people use both names. I introduced SQL while we were going over queries in Access. Sure, we used the design grid most of the time, but I did show you the SQL queries that were being executed behind the scenes.

Why bother knowing SQL when the design grid is so simple to use? Well, for one thing, you need to be sitting at the computer and looking at the database for that design tool to work – if you are a thousand miles from the database, there will be no access to that design grid.

I know you've shopped for things on the web. You have asked to see the store's collection of sweaters, books or CDs. You've provided your name, credit card number and shipping address. In both cases, there had to be some interaction with a database. In one case, we had to find all the sweaters and in another case, we had to remotely get data (billing/shipping data) into a database. Knowing a few basic SQL commands is going to make it possible for you to create one of these interactive, database-driven web sites. Remember those guys in chapter one who had the paintball database? Those two guys created a web page that let others search for paintball equipment and fields to play on – and they provided a way for others to provide new data to their paintball database. Clearly the developers of that database were not sitting at the database as requests for information came in from the internet…so how did they ask the database to find things, and how did they arrange for new data to land in their tables? The answer is SQL. They wrote a bit of computer code (not real hard stuff) and had that code pass SQL statements to their database.

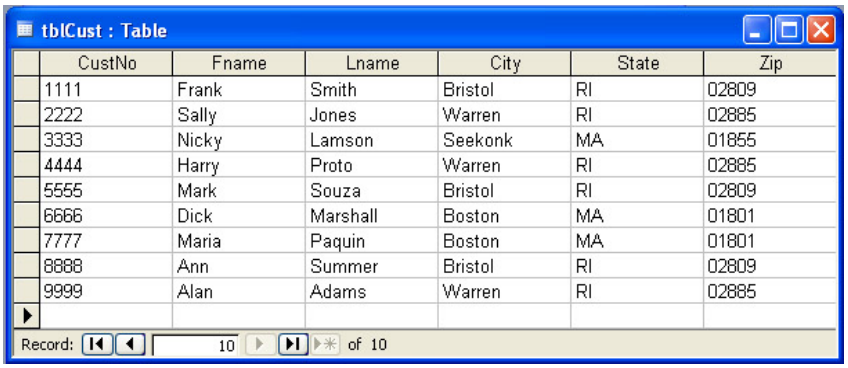
8.2 An overview of SQL statements

SQL statements fall into *four* categories. There are statements that help you search for data, add new data, modify existing data and delete data. The statements have names that match their function.

	Function	**Statement**
➢	Searching for data	(SELECT)
➢	Adding data to a table	(INSERT)
➢	Changing existing data	(UPDATE)
➢	Removing data from the table	(DELETE)

8.2.1 SELECTING data

Let's start with some very basic SELECT statements and gradually build to more sophisticated levels. The first few examples assume we have a table named tblCUST. The fields in tblCust are ***CustNo***, ***Fname***, ***Lname***, ***City***, ***State*** and ***Zip***.
Here's a peek at the data.



tblCust : Table

CustNo	Fname	Lname	City	State	Zip
1111	Frank	Smith	Bristol	RI	02809
2222	Sally	Jones	Warren	RI	02885
3333	Nicky	Lamson	Seekonk	MA	01855
4444	Harry	Proto	Warren	RI	02885
5555	Mark	Souza	Bristol	RI	02809
6666	Dick	Marshall	Boston	MA	01801
7777	Maria	Paquin	Boston	MA	01801
8888	Ann	Summer	Bristol	RI	02809
9999	Alan	Adams	Warren	RI	02885

Record: 10 of 10

Figure 8.1 – Selecting the entire customer table

To retrieve everything from the table we will use a SELECT statement.
The asterisk (computer people call it "star") indicates that we want all 6 fields (CustNo, Fname etc).

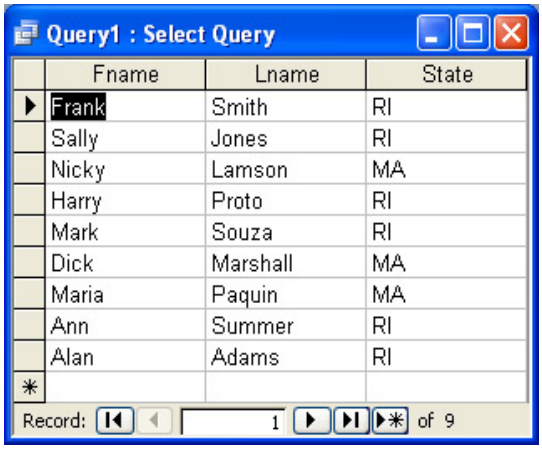
This statement will get every record (possibly millions of them). In our case, we get all 9 records displayed and we see every field. The results look exactly like Figure 8.1.

```
SELECT * FROM tblCust
```

8.2.2 Selecting Fields

To be a bit more selective, let's just ask to see the names and states of all our customers. Rather than use the "star" we'll specify the individual fields we want to see (Fig. 8.2). You're still seeing every record, but are now limiting the data to just viewing the names and states.

```
SELECT Fname, Lname, State FROM tblCust
```



Query1 : Select Query

Fname	Lname	State
Frank	Smith	RI
Sally	Jones	RI
Nicky	Lamson	MA
Harry	Proto	RI
Mark	Souza	RI
Dick	Marshall	MA
Maria	Paquin	MA
Ann	Summer	RI
Alan	Adams	RI

Record: 1 of 9

Figure 8.2 – Selecting just the fields we want

8.2.3 Setting Criteria

To get only customers from Rhode Island, you can use a WHERE clause – just be careful to place single quotes around RI (Fig. 8.3).

```
SELECT Fname, Lname, State FROM tblCust WHERE State = 'RI'
```

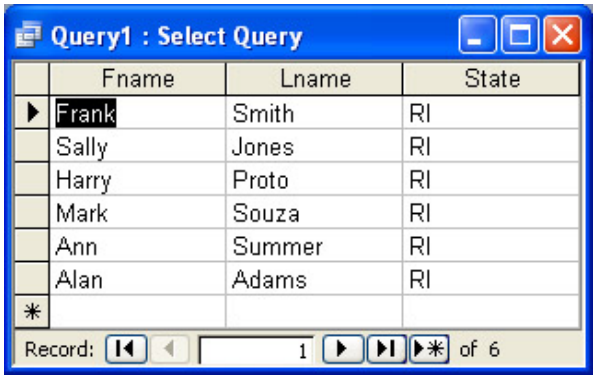


Figure 8.3 – Using a WHERE clause to restrict the listing to RI customers

8.2.4 Sorting the query results

To get that list sorted by last name, you can use an ORDER BY clause. Check the results in Figure 8.4 – notice the sorting of last names.

```
SELECT Fname, Lname, State FROM tblCust WHERE State = 'RI'
ORDER BY Lname
```

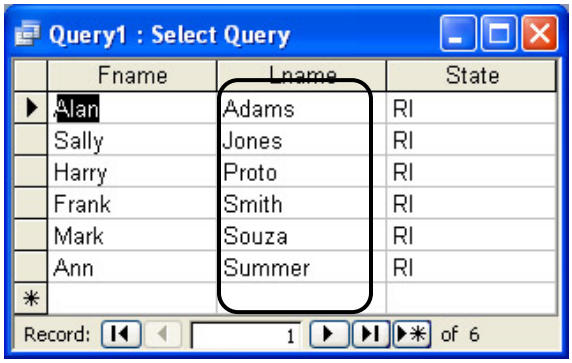


Figure 8.4 – Using ORDER BY to sort our customers by last name

SQL assumes you want the data in ***ascending*** order (**A to Z** or **1 to 9**). If you want the data in reverse order, just say ***DESC*** (descending) at the end of the statement.

```
SELECT Fname, Lname, State FROM Customers WHERE State = 'RI'
ORDER BY Lname DESC
```

8.2.5 Using Numeric fields in the criteria

Here's a simple rule – ***words get quotes around them, numbers don't!*** Notice how we put single quotes around RI as we set the criterion for State. Any time the field is defined as "Text", you are expected to use single quotes. If you try to put quotes around data that is coming from a field that you defined as Numeric (when you set up your table) there will be an error referring to a "***type mismatch***". The error is just a reminder that words need quotes and numbers can't have quotes around them.

So, if we wanted a listing of stocks with purchase prices above $20 (from the tblTrans table in our Brokerage database), the SQL might look like this:

```
SELECT * FROM tblTrans WHERE PurPrice > 20 ;
```

No Quotes around Numbers!

Several queries in this chapter involve finding data for a particular customer. As you encounter these queries, you might ask why I have quotes around the customer number… Well, when I defined the CustID in the table, I defined the data type as "Text". I know it looks like a number, but since I wasn't doing math with the customer number, I didn't bother defining it as a number. Since Access thinks the CustID is "text data", I have to put quotes around the CustID in any query we build.

8.3 Coding Calculated fields (Derived fields)

What if you want to include a calculated field? Let's say we have a table named "Orders" and it includes the retail price of our items. If we sell these same items to wholesalers at a 15% discount (85% of the retail price), we can write a SQL statement that includes that new price. ***When you create a new field you must provide the calculation of that field (in our case, we need a formula to represent the wholesale price) and we must provide a name for the field we just created – we'll name it as "Wholesale".*** Check out the SQL query and resulting run in Fig. 8.5.

SELECT ItemName, Price, **Price*.85 as Wholesale** FROM Orders

Figure 8.5 – Calculated fields can create data that isn't stored in the tables, but that can be derived from existing data

8.4 Dealing with Logical Operators – (NOT, AND, OR)

Most questions involve more than one condition. These are simple when using the Design Grid, but require a bit more planning when writing them as SQL statements. Each question that involves several conditions will use a ***logical operator*** – just a fancy way of referring to the words ***NOT***, ***AND*** and ***OR***. You might ask to see customers living in either "MA ***or*** RI" or perhaps ask to see people living in RI ***AND*** who bought IBM. Most of these are fairly intuitive – the problem comes when you start mixing the operators. I'm sure you understand the order of precedence for math… anything in parentheses is dealt with first, followed by exponents, multiplication and division and finally, any addition or subtraction. When there are items having the same importance, the rule says to go from left to right.

Parentheses
Exponents
Multiplication/Division
Addition/Subtraction

8.4.1 Order of Precedence for Logical Operators

Logical operators also follow precedence rules. ***NOT*** is done first, followed by AND and finally by OR – and parentheses can be used to elevate the importance of any operator, just as parentheses are used in math.

Parentheses
NOT
AND
OR

Does it make any difference to our queries? Let's take a look at an example based on our Brokerage database. The tblTrans table lists Customers, the Stock they own, the date it was purchased and the purchase price. We might want to know who bought IBM or APPLE and paid more than $50 a share. Here's one attempt at the query…

```
SELECT Lname FROM tblTrans WHERE StockAbbrev = 'IBM' OR StockAbbrev =
'Apple' AND PurPrice > 50
```

The query certainly makes sense – until you take the order of precedence into account. In this query, we have both ANDs and ORs. Take a look at the AND – it is more important than the OR, so it gets operated on first. That makes the query look very different from what we had intended:

Condition #1: StockAbbrev = 'IBM'
or
Condition #2: StockAbbrev = 'Apple' AND PurPrice > 50

This query will retrieve a list of people who bought IBM (regardless of price!) and a list of people who bought Apple for more than $50. If condition #1 OR condition #2 is satisfied, you're in the list!

To get the OR operated on first, we need to use parentheses. The correct query would be:

```
SELECT Lname FROM tblTrans WHERE

(StockAbbrev = 'IBM'  OR  StockAbbrev = 'Apple')

AND  PurPrice > 50

* I've placed the conditions on separate lines to help make my point… but you would
type the query as one line…
```

8.5 Working with Dates

Dates questions are crucial to the function of most businesses. For our brokerage, we might want to calculate the number of years a customer has owned each of their stocks, or ask who bought stocks during a certain month, or within the past month. In Access, you know that we can refer to today's date with the DATE( ) function and that specific dates need to be surrounded by pound (number) symbols. Remember also that the difference between two dates will tell you the number of days between the dates; once you know the number of days in the problem, you can slice the days into week, month or year-long units.

Check out some of these examples – most of these were initially developed in Chapter 7 when we were still using the Design Grid…

List the number of years each customer has owned their stock:

```
SELECT Lname, StockAbbrev, ( Date( ) – PurDate ) /365 AS YearsOwned
FROM tblTrans ;
```

Which stocks have been held for more than a year?

```
SELECT Lname, StockAbbrev, PurDate FROM tblTrans
   WHERE ( Date ( ) – PurDate ) / 365 > 1 ;
```

What purchases did Customer 222 make during April of 2006?

```
SELECT Lname, StockAbbrev, DatePur  FROM tblTrans
   WHERE CustNo = '222' AND DatePur >= #4/1/2006# AND DatePur <= #4/30/2006# ;
```

Since SQL understands BETWEEN, the query could also have been written as…

```
SELECT Lname, StockAbbrev, DatePur  FROM tblTrans
   WHERE CustNo = '222' AND DatePur BETWEEN #4/1/2006# AND #4/30/2006# ;
```

8.6 How do I refer to a variable as part of my query?

I hope you're getting the impression that SQL is very close to asking for your data in a nearly "English-like" language. The complication comes when you don't know what you need to retrieve – perhaps a user has entered a State while completing an order form on your web site and the name of the State is now sitting in a variable called fState. **Be very careful**…if you write a statement such as…

```
SELECT Fname, Lname FROM tblCust WHERE State = 'fState'
```

you will retrieve **nothing**, unless we suddenly have a new state named **fState**! If you think of a variable as being something like a box that holds data, there needs to be a way to see inside the box and use the value we find there as the item we are searching for. So, if the variable is holding the value "RI", then we need to find all of the Rhode Island

records. Clearly, there needs to be a way of finding out what data is sitting in the fState variable and locating customers in *those* states. Unfortunately, there is a complication – you still need the quotes around the state name since "word data" (computer people call words "strings") always gets quotes placed around them.

The problem is getting the quotes around that data, while keeping the variable out in the open where the machine can see the value that needs to be retrieved… obviously you don't want fState to be inside the quotes. The solution is a bit confusing – even for some of us "in the business". You need to build the SQL statement as a "string" that ties the various sections together – tying pieces together is something called ***concatenation***. If you built the Brokerage Report in our Reports chapter, then you have experience with concatenation. We'll be using the same approach to tying words together, but doing it with a more complicated problem.

When you concatenate words, you use the ampersand symbol (&) to tie the pieces together. For example, if you have variables for first name and last and want to tie the two together with a space between them, you can do this:

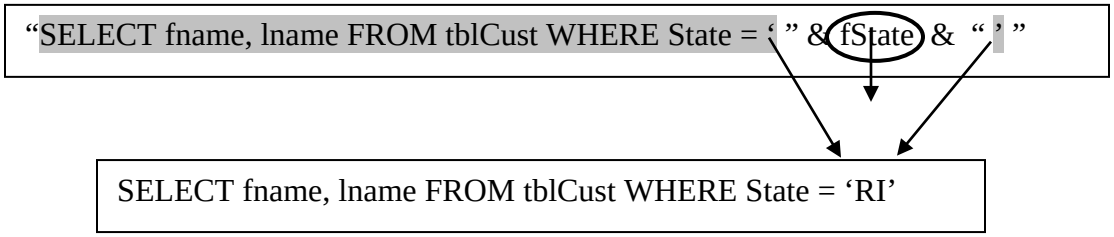
```
Fname  & "   "  &  Lname
```

Our simple SQL statement that lists people from some state (being stored in the fState variable) can take advantage of the concatenating technique to 'pause' the SQL just as the fState variable needs to be positioned in the command – then continue by tying on the closing quote mark.

```
"SELECT fname, lname FROM tblCust WHERE State = ' " & fState  &  " ' "
```

As you read across the statement, notice that we have three sections. The first runs right up to the single quote that will go before the state we need to match. That gets concatenated with the contents of the variable fState, which was left out of the quotes so we can find out what value is hiding inside that variable. Finally, we tie on the last section which is simply the single quote that needs to go after the state name.

Let's compare the SQL before and after the machine reads the variable. If the state had been RI, we now have a SQL statement that reads…



Do your best to get comfortable with the idea of concatenating a SQL statement – it will pay huge dividends when the time comes for us to build an online business. In Chapter 11 we will build an online dating service. Any person searching for a date will let us

know the age range and gender they are looking for. Those answers will be stored in variables that we will need to concatenate into a SQL statement that will SELECT people matching the requested criteria.

8.7 ADDING DATA to the database

SQL provides an ***INSERT*** statement for pushing data into tables. You list the fields that are being filled and then list the data that goes into those fields. For example, we can load data into our Customer table with the following statement:

```
INSERT INTO tblCust (CustNo, Fname, Lname, City, State, Zip) VALUES
('123', 'Sam', 'Smith', 'Bristol', 'RI', '02809' )
```

As you can imagine, this INSERT statement gets a bit hairy once you need to use variables rather than actual data! Concatenating the statement together can be done, but you need to be *very* careful with the quote marks… I've written it for you – try to watch the sections being tied together. Focus on how the single quotes are being positioned before and after every variable, along with the commas that need to separate the values – just as if they were actual names and cities. I've repeated the same statement below this, with some highlighting to make it easier to follow the pattern of quotes and commas.

```
"INSERT INTO tblCust (CustNo, Fname, Lname, City, State, Zip) VALUES ( ' " &
CustNo & "' , ' " & Fname & " ' , ' " & Lname & " ' , ' " & City & " ' , ' " &
 State & " ' , ' " & Zip & " ' )"
```

```
"INSERT INTO tblCust (CustNo, Fname, Lname, City, State, Zip) VALUES ( ' " &

CustNo & "' , ' " & Fname & " ' , ' " & Lname & " ' , ' " & City & " ' , ' " &

State & " ' , ' " & Zip & " ' ) "
```

8.8 UPDATING EXISTING DATA

It would be great if the data in our database could be left alone once we build it, but you know that people get married, last names change, addresses change, telephone numbers change etc…

There needs to be some way to locate a specific record and change some of the data. The SQL **UPDATE** statement can do this, and makes the operation fairly simple. One obvious issue is that you need to be certain you have the right person before you start changing things! That can be done with the WHERE clause – just as though you were searching for a record. You can then use a SET statement to set any field to a new value. Check out the following update of a person's interest in music, from whatever it currently

is – to their new interest in *Jazz*. tblMusicPref would be the name of the database table that holds the preference info for each customer.

```
UPDATE tblMusicPref  SET  Style = 'Jazz'  WHERE CustID = '1234'
```

8.9 Deleting Data:

SQL provides a DELETE statement that will clear records from your database – be very careful with this…you get no warning that you are about to do something stupid, like delete all the data from a table! As with UPDATE, the WHERE clause lets you specify which records you want to deal with. Here's a DELETE statement that clears all records in the Customer table, provide the record is for customer ID "1234".

```
DELETE   FROM  tblCust WHERE CustID = '1234'
```

Careful programmers often do a SELECT first; just to be sure they have the correct person before issuing the actual DELETE statement. Perhaps they can check the last name and match it with the CustId before dropping a valued customer!

8.10 Functions that summarize data – Sum, Avg, Count, Max, Min

What's the average price of IBM? How many customers live in Massachusetts? What is the total number of shares of IBM that the brokerage is holding for customers? SQL has all the basic functions that you might expect, including Sum, Avg, Count, Max and Min. Here's a query that finds the average price of IBM…

```
SELECT StockAbbrev, Avg ( PurPrice ) FROM  tblTrans
WHERE  StockAbbrev = 'IBM'  ;
```

It would be just as simple to Count the CustIDs for everyone living in MA, or to find the highest price paid for "Apple".

```
SELECT State, Count(CustID) FROM tblCustomer WHERE State = 'MA' ;
```

```
SELECT StockAbbrev, MAX( PurPrice )  FROM tblTrans WHERE StockAbbrev = 'Apple' ;
```

What if we wanted to find the average price for *every stock*? Instead of one collection of records (every "Apple" purchase), we would need to create a group of records for each stock and calculate the average for each grouping – that's where Aggregate queries come into play…

8.11 Aggregate (Group) Queries

Aggregate queries (sometimes called 'group' queries) are a powerful way to gather data into groups and then do work with the groups. In Chapter 7 we grouped the data by Customer number and determined the total value of each customer's portfolio. We could also group the data by DATE and determine the total commission for each broker for each day. Here's an interesting one – what's the highest price paid for each stock during a particular time period. In each example, the data had to be gathered into groups before we made our calculation. In SQL this is accomplished by saying "GROUP BY".

Here's an example using the Brokerage database where I want to count the number of customers in each state. You might recall from Chapter 7 that the aggregate query can only list fields that are part of the process. In this example, we want to GROUP the data by State and then Count the Customers (perhaps count the customer numbers). Check out the query:

```
SELECT State, COUNT(CustNum) FROM tblCust
   GROUP  BY  State ;
```

Notice how we included only those fields actually involved in the query – State is there because we are grouping the data by State – CustNum is there because we are counting the Customer Numbers as a way of determining the number of customers we have. Nothing else should be in the query – if you have fields that aren't involved in some way, the query will generate an error telling you that some field you used is not part of the "aggregate function"!

Here's an interesting example – this time with the brokerage system's Trans table. Can we determine the total each customer has spent on each stock? That involves gathering each customer's data into a group – and then, grouping the stocks for each customer. So, for Customer 1111, we will gather all of their data into a group, then find all of their purchases of Apple and form a group out of those stocks, and do the same with every stock they own. We will SUM the total amount they have spent on each stock. Notice that we have a group with groups inside it – each customer represents a group of data and within each customer, there are smaller groups representing each stock that customer owns. The "SELECT" part of the query lists the three items we want displayed (the customer number, the name of each stock and the total spent on that stock). I can list both CustNum and StockAbbrev because they are both involved in the grouping process – and the SUM is the action I want to take with each group. Here's the query:

```
SELECT CustNum, StockAbbrev, SUM (  (PurPrice * Shares) )  AS TotalCost
   FROM tblTrans
      GROUP BY CustNum, StockAbbrev ;
```

The result might appear as Figure 8-6.

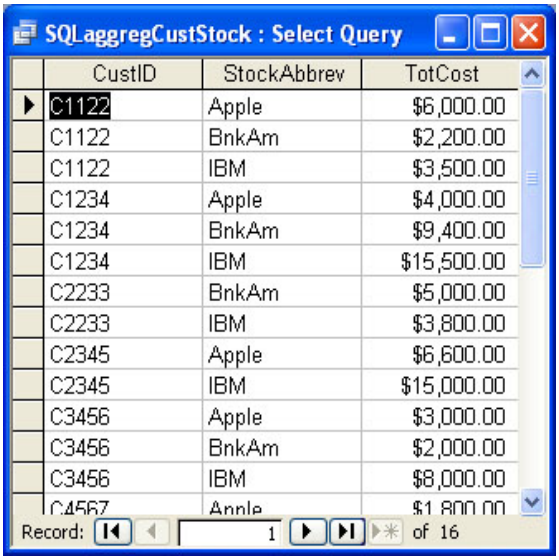


Figure 8-6. Total cost of each stock for each customer

8.11.1 The HAVING clause

One final refinement of the aggregate process… what if you want to see the results, but only for people who have an investment that exceeds $10000? You might be tempted to include a WHERE clause – but WHERE only works with rows of data; it won't work with a GROUP. For Groups, we need to use the word "HAVING". It's just like a "WHERE", but is reserved for filtering groups.

Do you want to see the aggregate list we just created, but restrict the output to the stocks people paid over $10,000 for? Here's the query…and the Run.

```
SELECT CustNum, StockAbbrev, SUM (  PurPrice * Shares )  AS TotalCost
   FROM tblTrans
      GROUP BY CustNum, StockAbbrev
         HAVING SUM ( PurPrice * Shares ) > 10000 ;
```

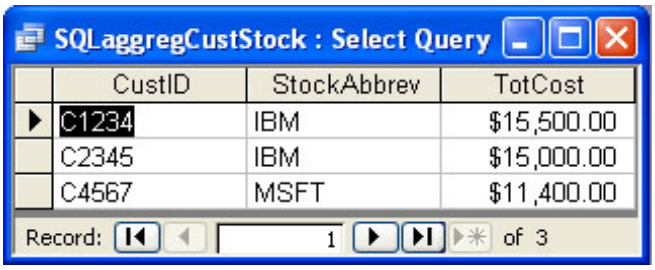


Figure 8-7. Total cost of stocks for each customer where the purchase value exceeds $10,000

8.12 Let's practice our new SQL skills

Well, at this point you have a basic understanding of how SQL works. Now it's time for some hands-on practice.

Begin by creating two tables in Access – one called tblCUST that contains the names and cities of our customers. The other table will be called tblORDERS and will hold the items ordered by each customer.

Here are the table designs and the relationship diagram (Fig. 8.8). Be sure to specify CustID as the primary key in the Cust table and OrderNum as the key in the Orders table. Once you have the tables set up, place a few records in each. You can use my data, or make up your own.

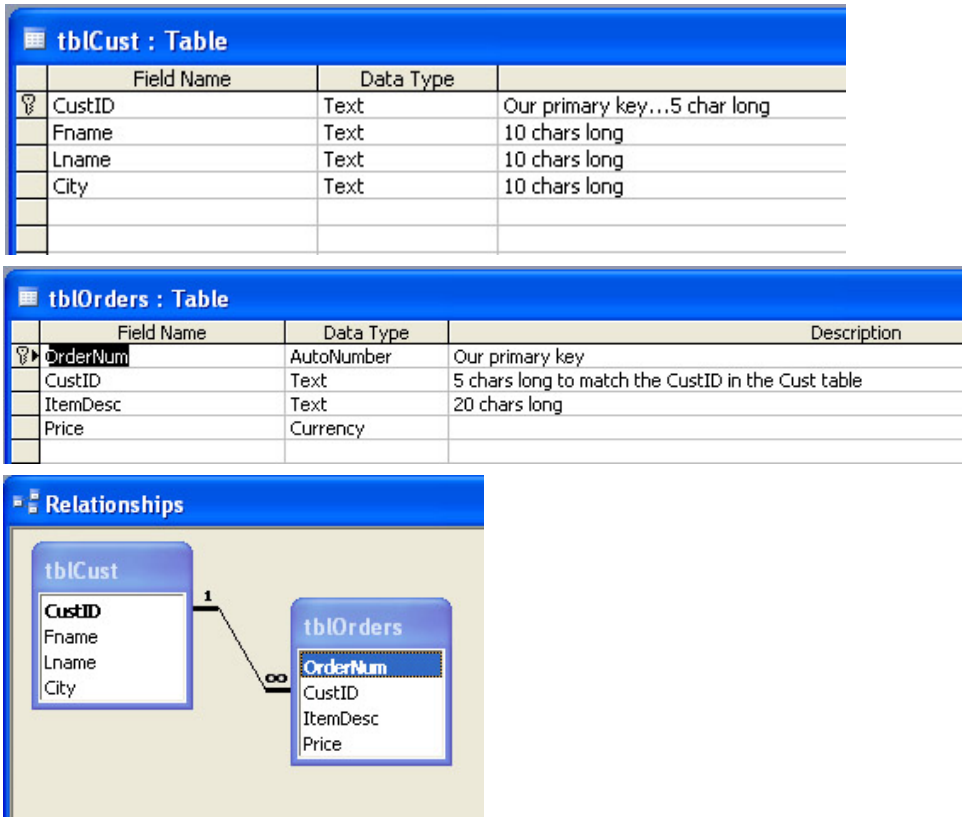
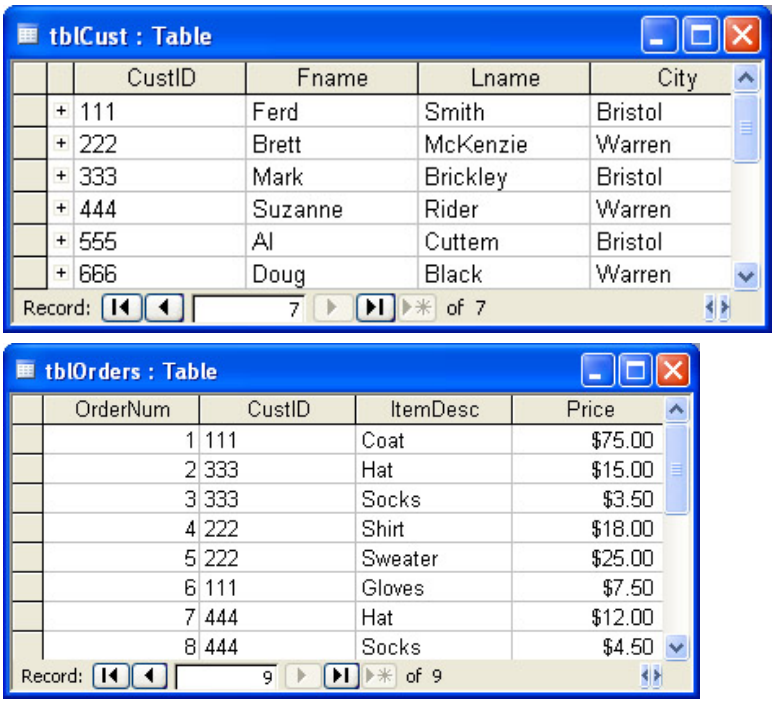


Figure 8.8 – Table designs and relationships for the hands-on SQL practice

Here's some sample data to play with (Fig. 8.9).



tblCust : Table

	CustID	Fname	Lname	City
+	111	Ferd	Smith	Bristol
+	222	Brett	McKenzie	Warren
+	333	Mark	Brickley	Bristol
+	444	Suzanne	Rider	Warren
+	555	Al	Cuttem	Bristol
+	666	Doug	Black	Warren

Record: 7 of 7

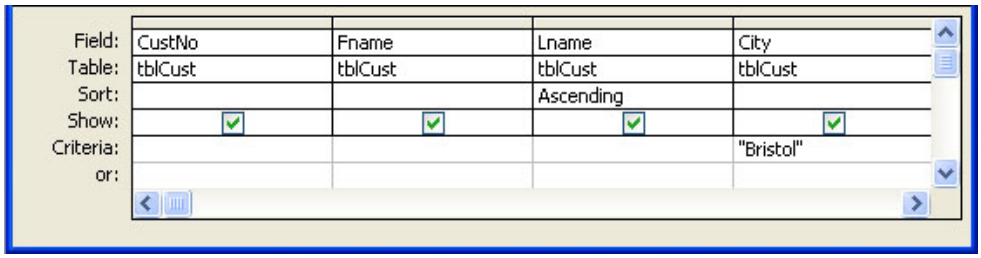
tblOrders : Table

OrderNum	CustID	ItemDesc	Price
1	111	Coat	$75.00
2	333	Hat	$15.00
3	333	Socks	$3.50
4	222	Shirt	$18.00
5	222	Sweater	$25.00
6	111	Gloves	$7.50
7	444	Hat	$12.00
8	444	Socks	$4.50

Record: 9 of 9

Figure 8.9 – Sample data for the Customer and Order tables

Once you've got data, start a new query (design mode). Add just the CUST table to the query grid and select each field in the CUST table. Set the criteria to include only the town of Bristol and have the results sorted by last name. Your query grid will look like Figure 8.10:



Field:	CustNo	Fname	Lname	City
Table:	tblCust	tblCust	tblCust	tblCust
Sort:			Ascending	
Show:	☑	☑	☑	☑
Criteria:				"Bristol"
or:				

Figure 8.10 – Use the query grid to check out your first SQL statement

Access provides this very intuitive interface for the development of queries. It's great for new users, but under the hood Access is actually running SQL. You can check this out by changing the view from "Design" to "SQL" (Figure 8-11). Give it a try – this SQL view displays the SQL statement that matches your query design.

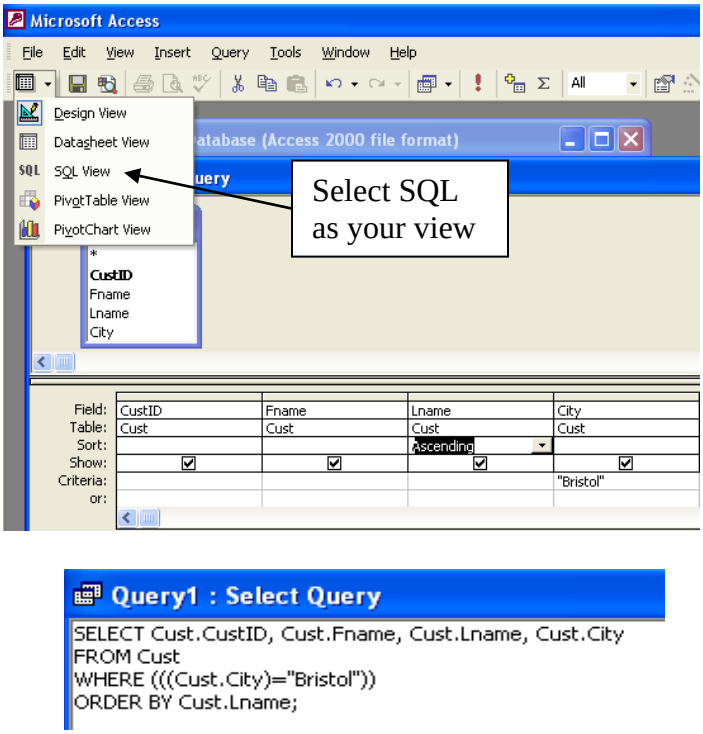


Figure 8.11. Moving into SQL view is a great way to learn SQL

Access is more careful with naming than we've been in our examples – and that makes this statement appear more complicated than it really is. Each field is prefaced with the name of the table the data is coming from. The statement is also careful to follow SQL conventions such as ending the statement with a semicolon and putting each command in uppercase. Would it work without all the extra flourishes? Give it a try. Delete the SQL statement and enter this:

```
SELECT fname, lname FROM tblCust WHERE city = 'Bristol' ORDER BY Lname
```

Click the exclamation mark that runs the query and it should work nicely. In fact, if you look at the design view, you'll see that it changes as you change the SQL statement.

Let's practice a few more statements. Can we add data to the Cust table – by doing an INSERT statement?

```
INSERT INTO tblCust (CustID, Fname, Lname, City) VALUES ('777', 'Tom', 'Jones', 'Barrington') ;
```

What if Tom moves to Bristol – we need a way to update his city. Try this update query:

```
UPDATE tblCust SET City = 'Bristol' WHERE CustID = '777' ;
```

Finally, what if Tom asks us to delete him from the list of customers – we can run a Delete Query…

```
DELETE FROM  tblCust  WHERE CustID = '777'  ;
```

Just For Geeks

Getting Fancy with SQL

As simple as SQL appears at first glance, it can get fairly complex. Each of the queries we played with in this chapter was based on a single table and could be answered by a single query. This installment of 'Geeks' enhances your SQL skills by introducing the coding of JOINS and the writing of SUBQUERIES. In the main part of this chapter we avoided queries involving joins and subqueries in order to keep focused on the basics. Many interesting questions, however, can't be answered without connecting two or more tables – and some questions can't be answered without knowing the answer to some other question first.

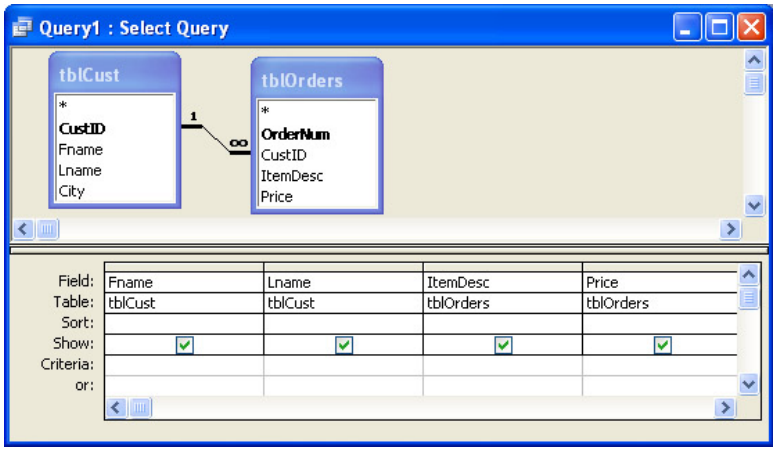
Think of how limiting a relational database would be if you couldn't establish a relationship between two tables. In our brokerage, we would be able to list our customers – but wouldn't be able to list the stocks each customer owned. In fact, most of the interesting questions for the brokerage really demand a connection between tables…

Subqueries are a great feature of SQL that let you run a query that requires the answer to some other query as one of the criteria. Here is a simple example: List the customers who bought IBM for more than the average purchase price of IBM. If we can write a query to find the average price of IBM, then we could use the query about the average as part of the condition for our initial question (Who bought IBM for more than the average purchase price of IBM?) Rather than compare the PurPrice with some specific value, we could substitute the query that computes the average price. Along with joins, subqueries are a great addition to your SQL background.

May I JOIN you? (Some background on Joins…)

So far we've been working with one simple table, but queries often involve multiple tables that are joined together (as defined in our relationship diagram). Close your current query and start a new one – this time include *both* the tblCust and tblOrders tables

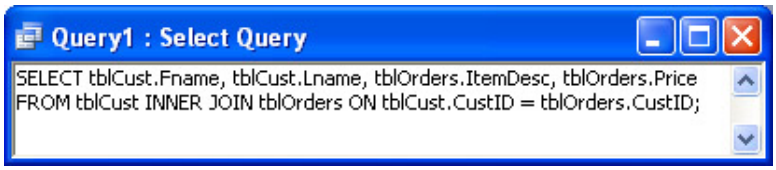
in the query grid (don't bother with SQL for now). Ask to see Fname, Lname, ItemDesc and Price.



You know that this query involves two tables related by a common field. In this case, the common field is CustID. CustID appears as the ***primary key*** in the tbl*Cust* table and as a ***foreign key*** in the tbl*Orders* table. Recall that a foreign key is just a field that is a primary key in some table – but just happens to be in some other table where it is not playing the role of a primary key.

We've taken the joining of tables for granted. Somehow by dragging our mouse from CustID in the tblCust table to CustID in the tblOrders table we magically created the join while we were working with the Relationship diagram. Well, we managed to describe the join, but whenever Access needs to involve two or more tables to create a Form, Report or Query, Access writes a SQL statement that expressly specifies the join. Simple joins are not hard to write, but they get pretty complicated as you involve more tables.

Let's take a look at the SQL statement we generated with our latest query. Focus on the JOIN. For now, notice that Cust is involved in a join with Orders and that the join is based on the field they have in common (CustID). Don't worry about the "Inner Join" part of this statement; we'll go over INNER and OUTER joins in a moment



Let me walk you through the join. We are selecting fields such as Fname, Lname, ItemDesc and Price that are part of a temporary table that results from the tblCust table being joined with the tblOrders table – with the common field being CustID of the tblCust table and CustID of the tblOrders table.

The join certainly makes the SQL statement more complicated looking – but it's not that tough once you see a few.

Introducing the coding of JOINs

Writing the code to connect two tables isn't hard. Our example will use the most common type of join – the INNER JOIN. With an inner join, data is listed only if there are matching rows in each table. For our brokerage database (with a join of the Customer and Transaction tables) a customer will only be listed if the customer actually owns stocks.

The join has two parts:

1. Identify the tables on each side of the join
2. Identify the fields the tables have in common

Let's use the tblCust and tblOrders tables as our example. The tables have CustID as their common field.

Focusing on the FROM part of the query, simply name the tables involved in the join and place them on either side of the INNER JOIN. It looks like:

The second part of the join involves naming the common field – in this database, the tblCust and tblOrders tables have CustID in common. Since CustID appears in two tables, we'll need to qualify which CustID we're talking about. In SQL you can do that by naming the table, then the field (with a dot between the table and field names). That will make the join appear as:

Let's put our new experience with joins to work – let's list the names of customers and the items they have bought.

```
… FROM tblCust INNER JOIN tblOrders

        ON tblCust.CustID = tblOrders.CustID
```

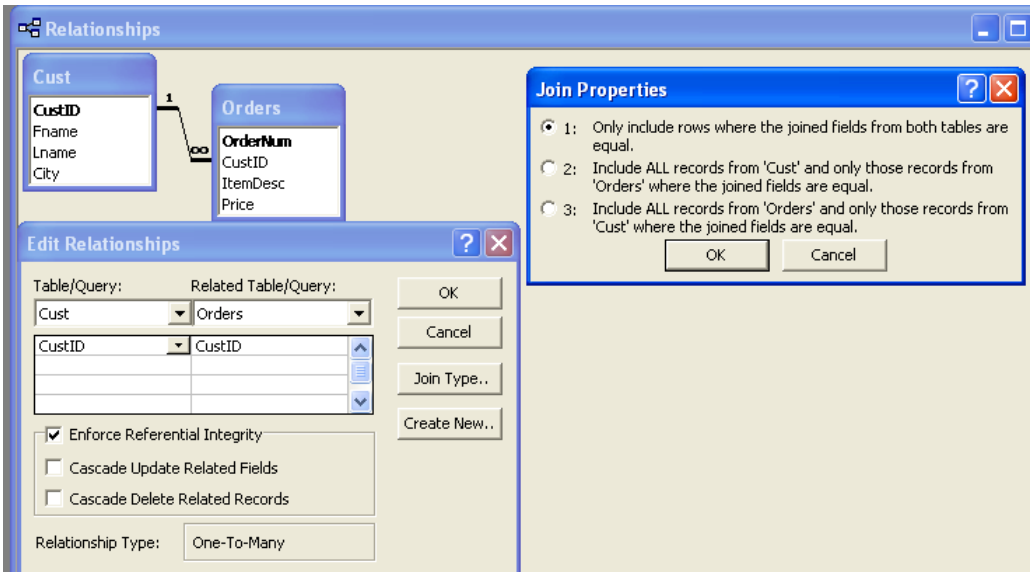
```
SELECT Lname, ItemDesc, Price
        FROM tblCust INNER JOIN tblOrders
                ON tblCust.CustID = tblOrders.CustID ;
```

That's actually pretty simple. Unfortunately, SQL starts to look very nasty once you involve three or more tables. If you want to see what it looks like, build a query in Access that involves the Customer, Trans and Stock tables from our Brokerage system – then check the query out by moving to SQL view.

You should be aware that people code joins in several different ways – for instance, you can do the join without saying "INNER JOIN". I've written our join examples to match what Access does. If you design a query involving more than one table, then switch to SQL view, you'll find that Access is specific about the kind of join being performed (INNER, LEFT OUTER, RIGHT OUTER) and that Access uses the word "ON" when identifying the common fields.

INNER Joins and OUTER Joins

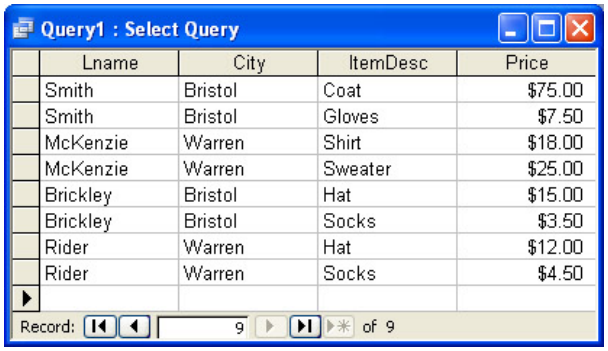
Before we end our discussion of Joins, let's spend a moment on the kind of joins you can build. Go to the Relationships diagram (Tools menu / Relationships), Right-click the line for your join and select **Edit**. Click the ***Join Type*** button to display the various options for join type.



Every join you've used in Access has probably been the first choice in this list. It's called an ***Inner Join***. Using our sample database, an inner join will retrieve every customer and their order, provided there were matching records in **both** tables. If a customer doesn't have orders, they won't be listed. Why not give that a try – create a new customer in the Cust table, but don't give them any orders. When you run a query that includes fields from both tables, a customer is listed only if they have matching records in the orders table.

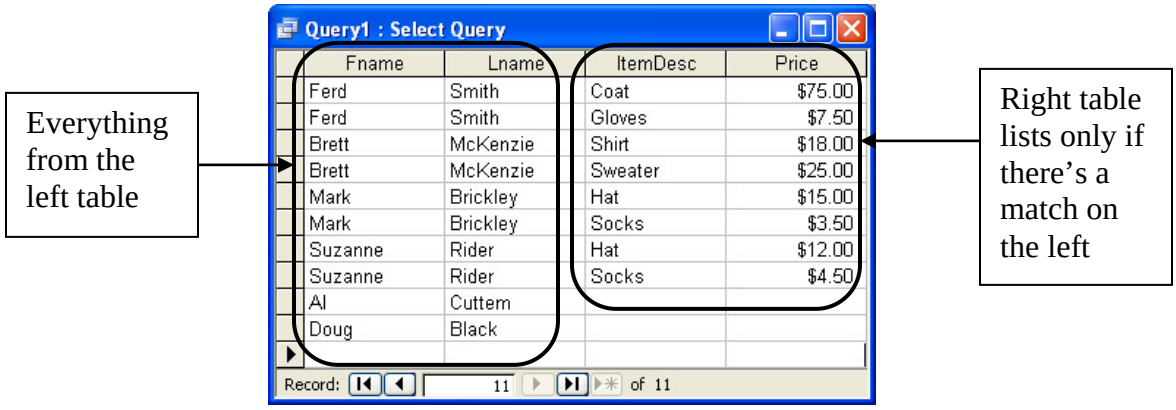
Note: You can also adjust the Join type while designing the query – simply right click on the join line connecting the tables and select ***Join Properties***.

Notice how customers such as Cuttem and Black are missing from a query that lists customers and the items they bought. An inner join only includes rows of data when matching records appear in both tables. In this case, neither of these people had a CustID mentioned in the tblOrders table.



The second and third joins are called **OUTER** joins. There are *Left outer joins* and *right outer joins*. Outer joins list every record from one side of the join (whether they have a matching record on the other side or not) and only those records from the other side that do have a match.

Let's say we wanted every customer listed – whether they have orders or not. A left outer would do that for us. Every customer from the table on the left (tblCust) will be listed, but only those records from the Orders table that have a matching customer will be displayed. You can verify this by changing the relationship to a left outer (option #2 in the "Join Type" list – and then run a query that lists customers and their orders. This time, Cuttem and Black will be listed, even though they have ordered nothing – however only rows from the tblOrders table with matching CustIDs in the tblCust table are shown.



The left outer is a great way to find people who aren't active yet. For example, list all students – even if they haven't started taking classes yet, find customers who haven't

bought anything, list all of our salesmen even if they haven't sold anything (maybe *especially* those who haven't sold anything!).

A **right outer** is easier to illustrate in a database where you aren't enforcing referential integrity. Recall that referential integrity insists that a record exist on the ONE side of the relationship before you start filling the many side. For example, you can't sell items to customer 222 until customer 222 exists. If you haven't enforced referential integrity, it is possible to find yourself with records in the orders table that don't have corresponding records in the customer table. That's bad news for a business and we should regularly check for those situations. A **right outer join** will list all of the records in the orders table (the many-side of the relationship) even if there is no mention of the customer in the customer table. Sorting the data should help float the bad records to the top of the listing.

There is one last join type – called a **full outer join**; it isn't listed among the choices in Access. As you might guess, it includes all records from the left side and all records from the right. This listing would include all *customers* (even if they haven't ordered yet) and all *orders* (even if we can't find who ordered the item)!

The basics of subqueries

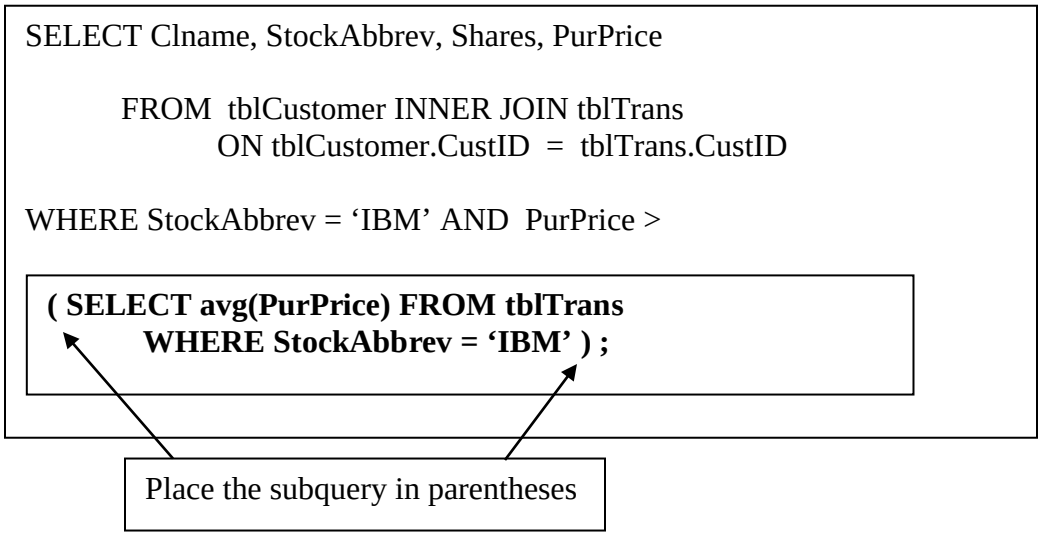
The second topic of this Geeks installment introduces subqueries – essentially queries within queries. Each of these examples involves a question that needs the answer to some other question before the "main" question can be answered.

Can you find the average price of IBM? Of course you can – simply use the AVG function. Here's a SQL statement that determines the average purchase price of IBM.

```
SELECT Avg(PurPrice)  FROM  tblTrans

        WHERE  StockAbbrev  =  'IBM' ;
```

Once you know the average purchase price, it would be great to go back and compare each customer's purchase price against the average in an attempt to find people who paid more than the average purchase price for IBM. I suppose we could run this query to find the average and then enter that average into a query that lists customers above that average… but we would need to re-run the "average query" every time we wanted to generate the list of customers. Why not just use the "average query" as part of the condition?



Here's another example of a subquery: Let's find the names of the people who bought IBM most recently. We can write a query to find the "biggest" purchase date for IBM and then use that date to identify the customers who bought IBM on that specific date. The query would be:

```
SELECT Clname, StockAbbrev, PurDate
        FROM tblCustomer INNER JOIN tblTrans
                ON tblCustomer.CustID = tblTrans.CustID
        WHERE  StockAbbrev = 'IBM' and  PurDate =
          (SELECT  max(PurDate)  FROM  tblTrans WHERE  StockAbbrev = 'IBM') ;
```

You'll find lots of opportunities to use both joins and subqueries. My suggestion is that you use Access to help write any join involving more than two tables. For that matter, Access can be a testing ground for any SQL statement that you want to test out.

Keywords

Aggregate query	SET
Calculated field	Structured Query Language (SQL)
Concatenation	SubQuery
DELETE query	Summary Functions (sum,avg,count,max,min)
GROUP BY	UPDATE query
HAVING	Geeks: INNER JOIN
INSERT "query"	Geeks: LEFT and RIGHT OUTER JOINS
Logical Operators (Not, And, Or)	Geeks: FULL OUTER JOIN

Review Questions:

1. Which of the following is the correct order of precedence for logical operators?
   a) AND OR NOT
   b) NOT AND OR
   c) NOT OR AND
   d) OR AND NOT

2. Which query condition will list students in either the CIS or MKT major who have at least a 3.0 GPA? (I'm just showing the WHERE part of the query…)
   a) Major = 'CIS' OR Major = 'Mkt' AND GPA >= 3
   b) Major = 'CIS' AND GPA >= 3 OR Major = "Mkt' AND GPA >= 3
   c) (Major = 'CIS' OR Major = 'Mkt') AND GPA >= 3
   d) both A and C
   e) both B and C
   f) none will work

3. Which query will compute a person's pay and display it in a column labeled "PAY"?
   a) SELECT Lname, Hours, Rate, Hours * Rate AS Pay FROM Payroll;
   b) LIST Lname, Hours, Rate, Pay **:** [Rate]*[Hours] FROM Payroll ;
   c) SELECT Lname, Hours, Rate AS Pay (Hours * Rate) FROM Payroll ;
   d) LIST Lname, Hours, Rate Col **:** Hours * Rate FROM Payroll ;

4. Which of the following properly lists people (LN) who have worked at least 20 hours but no more than 50? HRS is a numeric field.
   a) SELECT LN, HRS FROM tblPay WHERE HRS > 20 AND HRS >50
   b) SELECT LN, HRS FROM tblPay WHERE HRS >=20 AND HRS <= 50
   c) SELECT LN, HRS FROM tblPay WHERE HRS BETWEEN '20' AND '50'
   d) SELECT LN, HRS FROM tblPay WHERE HRS >= '20' AND HRS <= '50'

5. How would you refer to a specific date (such as October 12, 2006) in a SQL statement? (Assuming you are using Access).
   a) October 12 2006
   b) '10/12/2006'
   c) #10/12/2006#
   d) Date(2006,10,12)

6. Which query will list people who've been with the firm for at least 5 years?
   a) SELECT Ln, DateHired FROM tblEmp WHERE DateHired >= 5
   b) SELECT Ln, DateHired FROM tblEmp WHERE (Date()-DateHired)/365 >= 5
   c) SELECT Ln, DateHired FROM tblEmp WHERE (Date()-DateHired)/365 <= 5
   d) SELECT Ln, DateHired FROM tblEmp WHERE Years 5

7. One of the brokers in the brokerage firm has decided to leave us. Until we can hire a new broker, we have decided to move all customers of the departing broker (brokerID 123) to one of our experienced brokers (brokerID 111). Which SQL statement will handle the job?
   a) UPDATE tblCustomer SET BrokerID = '111' WHERE BrokerID = '123';
   b) UPDATE BrokerID TO '111' FOR BrokerID '123';
   c) UPDATE BrokerID IN tblCustomer TO '111' FOR BrokerID '123';
   d) UPDATE tblCustomer BrokerID TO '111' FROM '123';

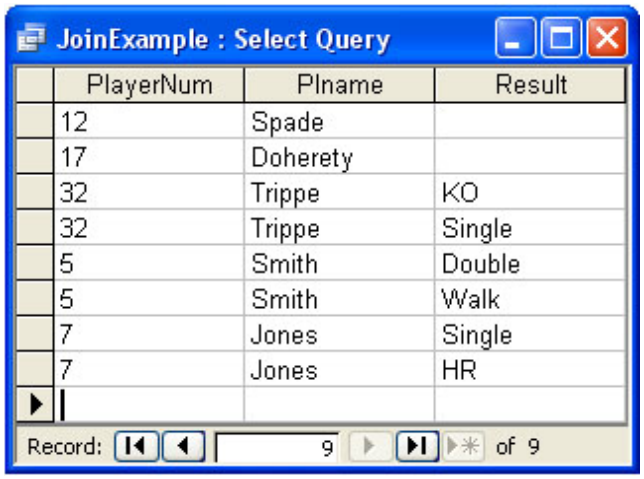
8. Now that customers have been transferred to Broker 111, we can safely delete broker 123. Which SQL will handle that task?
   a) DELETE BrokerID = '123';
   b) DELETE FROM tblBroker BrokerID = '123';
   c) DELETE FROM tblBroker WHERE BrokerID = '123';
   d) DELETE BrokerID = '123' FROM tblBroker;

9. What's the average Purchase Price of IBM? Assume you are working with the tblTrans table from our Brokerage database.
   a) LIST average OF StockAbbrev = 'IBM'
   b) SELECT avg (PurPrice) FROM tblTrans WHERE StockAbbrev = 'IBM'
   c) SELECT average (PurPrice) FROM StockAbbrev = 'IBM'
   d) LIST avg PurPrice FROM tblTrans WHERE StockAbbrev = 'IBM'

10. List the names of each stock and their average Purchase Price. This will require your query to gather all the 'Apple' purchases together and determine the average purchase price…then gather all the 'IBM' purchases etc…
    a) SELECT StockAbbrev, avg (PurPrice) FROM tblTrans
       GROUP BY StockAbbrev
    b) SELECT CustID, StockAbbrev, avg (PurPrice) FROM tblTrans
       GROUP BY StockAbbrev
    c) SELECT avg (PurPrice) FROM tblTrans
       WHERE GROUP BY = StockAbbrev
    d) SELECT StockAbbrev, avg (PurPrice) GROUP BY StockAbbrev
       FROM tblTrans

11. (Geeks Question): I have a "baseball" database with two tables – The Players table includes a PlayerID, the player's jersey number, their name, team, position etc. The second table is called "AtBats" and lists every 'at bat' during every game. It includes the player's ID, the date of the game and what happened at each particular 'at bat'. The tables are related by the PlayerID (a player appears once in the Player table, but can appear many times in the AtBats table (or not at all if they sit on the bench!). What kind of join will list a player ONLY if they have been up to bat?
    a) INNER
    b) LEFT OUTER
    c) RIGHT OUTER

12. (Geeks Question): Staying with the Baseball database for a moment, what kind of join will list players even if they have never been up to bat? The results might look like this…



    a) INNER
    b) LEFT OUTER
    c) RIGHT OUTER

13. A local company maintains a database that tracks the total sales made by each of their salesmen. The sales manager has asked you for a list of salesmen who are at least $10,000 below the average sales total. You decide to use a subquery to identify the average sale and then have the main query look for people who are $10,000 below that. Which of these SQL statements will do that?
    a) SELECT Lname, Sales FROM tblSalesTotals WHERE Sales <
       (SELECT avg (Sales) FROM tblSalesTotals) – 10000 ;
    b) SELECT Lname, Sales FROM (SELECT avg (Sales) - 10000) ;
    c) SELECT Lname, Sales FROM tblSalesTotals WHERE
       (SELECT Lname FROM tblSalesTotals WHERE avg (Sales) < 10000);

Exercises:

Use the Flights database that you've been developing for the past few chapters to practice writing SQL statements. Pass in both the SQL statement and a screen shot of what the run looked like (use a PrintScreen of the run pasted into Word). Except for question #10, you will not need to code a "join" – that is, every query involves a single table (either tblFlights or tblPasseng)

1. List all flights going from Boston to Chicago – sort them by Departure Date.
2. List all flights from Boston to either Miami or Chicago.
3. List all flights taking off within the next two weeks (your instructor may ask you to modify your departure dates to include flights in this time period).
4. List all flights taking off in the upcoming month of April (again, you may need to modify your departure dates to accommodate this query…)
5. Let's assume the federal government has established a security tax of 5% on every ticket (to handle airport security). Write a SQL statement to list each passenger and the security tax they need to pay (the security calculation should display with the name "SecurityTax").
6. Write an aggregate query to count the passengers on each flight
7. Write an aggregate query to determine the total revenue for each flight (you'll be adding up the ticket prices).

8. This afternoon's storm has forced us to cancel a flight and move customers to a different flight. Write a query to update the Flight number for every customer currently on Flight 3333 and set the new flight number as Flight 7777. Since you have Referential Integrity turned on, you will need to create a flight 7777 in the tblFlights table before you attempt to put passengers on that flight.

9. Well, things just aren't improving. The forecast has been updated and predicts strong thunderstorms for most of the evening. The airline has decided not to take chances and has cancelled all flights. Flight 7777 will not take place. Write a SQL statement to delete this flight from the flights table. If you have "cascading deletes" activated, the customers will be deleted as well. If this were a 'real' database, can you suggest a better way to handle this situation? If the customers get deleted when a flight is deleted, does that create a problem – and how might you better deal with it?

10. For people who read the GEEKS section… police have received a tip suggesting that a fugitive will attempt to get on a flight from Boston to Miami. Can you get a passenger listing for any flight going from Boston to Miami? It will involve a join…

11. Again, for people who read the GEEKS section, determine who paid more than passenger "Buckley" did on flight 2222?