JOURNAL OF INFORMATION SYSTEMS
Vol. 20, No. 1
Spring 2006
pp. 117–137

Using SQL to Develop Database Query Proficiency: A Follow-Up Note to Borthick, Jones, and Kim (2001)

Alan I. Blankley
The University of North Carolina at Charlotte

ABSTRACT: In the Spring 2001 issue of *Journal of Information Systems* (*JIS*), Borthick et al. (2001) presented a case in which students were to query a Microsoft Access database in order to determine the mutual assurance concerning customer service for an automobile manufacturer and its local dealer network. I follow-up this case by presenting an additional teaching note to the case that develops query strategies and solutions for the case using Oracle and its SQL application, SQL-Plus.

I. INTRODUCTION

In the Spring 2001 issue of *Journal of Information Systems* (*JIS*), Borthick et al. (2001) (hereafter, BJK) presented a case in which students need to query a Microsoft Access database in order to determine the continuous assurance concerning customer service for an automobile manufacturer and its local dealer network. After receiving inquires from interested customers, the automobile maker was supposed to refer those inquires to the dealer having the closest geographical proximity to the customer. After receiving the customer referral, the dealer was then supposed to contact the customer within 48 hours. The case involved having students first build the database and populate tables with data, then query the database in order to determine the dealers' response times to customers, as well as the manufacturers' compliance with the referral agreement. In order to meet the requirements of the case, students were to build the tables using Microsoft Access, then query the database using Access's "query-by-example" (QBE) capability in order to answer questions relevant to both the manufacturer's and dealers' business concerns. The authors mention that other database management system software (DBMS) may be used, and other query languages, like structured query language (SQL), could be used as well, but the instructions in the teaching note were all related to thinking through the query strategy and then generating the necessary queries using QBE.

The purpose of this follow-up note is to provide guidance for using the case with the Oracle database and its standard SQL application, SQL-PLUS.[1] This note may prove useful to faculty who would like to teach students SQL and/or use the Oracle DBMS, but who may not have the proficiency required to do so, and who may not be able to invest the time

[1] SQL may also be used within MS Access, of course, and the guidance provided here could also be helpful in developing the SQL code within MS Access. This strategy may be useful for faculty at institutions without access to Oracle, but it does have some limitations. First, students could use QBE instead of SQL, then turn in the SQL code generated by Access as their own. Second, the version of SQL used by Access is more limited in its capabilities. Third, using certain SQL commands requires that the user interact with Access Graphical User Interface (GUI) screens (creating tables using data definition queries, for example), which results in an inefficient combination of going back and forth between a GUI interface and an editor interface.

necessary to gain it. This note will provide faculty with the specific SQL instructions and techniques necessary to answer the case questions. More than simply being a solution set, though, this note will also allow faculty to efficiently gain the expertise needed to feel comfortable adding SQL and/or Oracle to their syllabi. I will discuss not only the SQL commands necessary to generate the correct query results, but also alternative approaches and rationales where appropriate.

Rationale for Teaching SQL

Using the case as a means of teaching SQL offers several advantages to both faculty and students. First, SQL is the industry-standard query language applicable to all large-scale DBMS software. It is widely used in industry, and is not platform-specific.[2] Teaching the case using SQL, then, allows faculty to provide solid training in a database language that has widespread industry acceptance and broad business use. This exposure is important for accounting students. The fact that many accounting information systems (AIS) texts now include some instruction in, or discussion of, SQL underscores this point (see, for example, Romney and Steinbart [2003] or Gelinas et al. [2005])

Second, In addition to being the most important query language in use, SQL functions as a data definition language, a data manipulation language, and a data communication language. Using SQL to address the issues in the case utilizes all these functions and allows students to see how tables get created, keys and constraints get applied, and data get manipulated.

Third, developing the necessary queries by using SQL, rather than a QBE interface, reinforces the logic behind the query because students have to grapple with the relational challenges necessary to extract the relevant information, and then express that using SQL. This may be the most important advantage. Teaching querying using SQL is not simply (or primarily) a matter of teaching a programming language. It is much more a matter of focusing on the thought processes, the logic underlying the queries, than it is a matter of concern over programming syntax. In answering the question, "How do I get the results I want?" students have to focus on the logic and actions required by the query and then develop a strategy for retrieving the desired data. SQL allows them to execute this intellectual strategy without having any prior programming experience.

Fourth, using SQL within Oracle allows students to explore Oracle's capabilities. This has the advantage of exposing students to a robust DBMS product widely used for accounting systems by large and mid-sized companies. More importantly, students can begin to discover the functionality and controls built into a large-scale DBMS that are lacking in desktop products. For example, when populating tables, students can build substitution variables into their SQL statements which allow for more controlled and faster data input.

Taken together, using SQL and Oracle to satisfy the requirements of BJK provides faculty with one means of achieving the "information use" objectives outlined in Borthick (1996). The case itself encourages students to decide what information is relevant to addressing the business problems presented in the case, while the focus on SQL requires that students gain experience first developing, then extracting the necessary information using the data definition language (DDL), data manipulation language (DML), and data query language (DQL) common to most, if not all, large-scale AISs.

[2] While variations in SQL versions do exist, the basics are the same across multiple platforms.

Case Requirements

The case requires students to build six tables and then query those tables in order to find the solutions to questions posed at the end of the case. Only five of the tables are necessary to answer all the questions. The fact that there is an unnecessary table included in the database may lead to discussion about efficient database design. It is not my purpose here to discuss the design of the database, but rather to provide SQL statements that address all the issues and to point out additional items of interest concerning SQL or Oracle that may be brought to bear on the case. For that reason, only the five tables from BJK necessary for querying purposes are presented in Tables 1–5.[3]

Once the students have built the tables, they need to develop a sequence of queries that together address the auto manufacturer's concerns as to whether dealers are complying with their (the manufacturer's) expectations of a 48 hour response time to customer inquiries.

II. METHODS

Logging on to Oracle and Basics

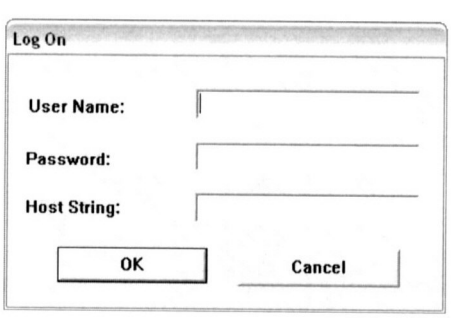
Before any work can commence on the case, users must first establish accounts for themselves and their classes. Generally, this can be accomplished by calling the university's IT services department and requesting an account. It may also be possible to set up an account using the web, depending on how each university's account maintenance procedures work. Once you have an account, you will be assigned a username, a password (which may be changed), and a "Host String" by the systems administrator. Key each of these into their respective locations on the start-up dialog box for SQL-Plus. See Figure 1.

After clicking on "OK," you will arrive at the SQL prompt, which looks like this:

```
SQL>
```

SQL-Plus is a line editor, and as such, is unfriendly and unforgiving. Fortunately there is an alternative to using the line editor, so any mistakes made while keying can be fixed quickly and easily. You can invoke the editor after you have begun writing a command at the SQL prompt. For example, if you type in "SELECT" at the SQL prompt, then press

FIGURE 1
SQL-Plus Log On Screen



[3] The original tables from the case included seven digit customer id and dealer id numbers, and nine digit referral ids. I dropped the leading zeros for simplicity.

TABLE 1
Customer

Primary Key: Customer ID

Customer ID	First Name	Last Name	Street	City	Post Code	Phone Work	Phone Home	Email Address
342512	Ryan	Hong	710 London Rd.	Atlanta	30344	(404) 876-4875	(404) 548-6625	ryan0980@aol.com
342525	Daniel	Lowell	225 Burbank Dr.	Atlanta	30314	(404) 567-2245	(404) 514-8898	low008@aol.com
342539	Terrel	Thomas	2985 Peachtree St.	Atlanta	30360	(770) 975-6521	(770) 548-9658	gold076@hotmail.com
342546	Cathy	Allen	1827 McPherson Rd.	Atlanta	30303	(770) 988-6521	(770) 985-3542	allen23@mci.com

TABLE 2
ReferralToDealer

Primary Key: ReferralID

ReferralID	CustomerID	DealerID	RefDateTime
10345	342512	24145	11/10/99 10:00 AM
10352	342525	16287	11/12/99 1:15 PM
10363	342539	37269	11/15/99 10:00 AM
10379	342546	405718	12/11/99 11:35 AM
10382	342512	24145	11/12/99 11:00 AM
10394	342525	16287	11/15/99 2:20 PM
10407	342546	405718	11/25/99 10:00 AM

TABLE 3
Dealer

Primary Key: DealerID

DealerID	Name	Street	City	PostCode	Phone
16287	Buckhead Auto	3126 Piedment Rd	Atlanta	30305	(404) 261-1851
23718	Neal Pope Motorcar	4420 Buford Hwy	Atlanta	30341	(770) 216-9700
24145	Paul Light	4125 Piedmont Rd	Atlanta	30342	(404) 261-1851
35284	Afford Auto	3350 Cumberland Rd.	Atlanta	30339	(404) 303-1400
37269	Bob Motoring	330 Forrest Rd	Atlanta	30349	(404) 361-3832
405718	Town Touring	141 Piedmont Ave.	Atlanta	30303	(404) 659-3673

TABLE 4
DealerResponseToReferral

Primary Key: ReferralID

ReferralID	DealerID	ResDateTime	PhoneResponse	EmailResponse
10345	24145	11/12/99 11:30 AM	Yes	Yes
10352	16287	11/12/99 4:00 PM	No	Yes
10363	37269	11/15/99 4:30 PM	Yes	No
10379	405718	12/11/99 4:00 PM	No	Yes
10382	24145	11/15/99 12:20 PM	No	Yes
10394	16287	11/16/99 9:00 AM	Yes	Yes
10407	405718	11/25/99 1:00 PM	No	Yes

enter, you will receive an error message and be returned to the SQL prompt. At this point, type in "ed" (for editor), and the database will open Microsoft's Notebook text editor, and create a file in the buffer called *afiedt.buf*. You can key in the code using Notebook, close it when finished, and the database will then display the code at the SQL prompt. Key in a forward slash (/), hit enter, and the database will then process the code. This process is much easier to use when writing SQL code, and vastly more efficient when correcting errors than using the SQL-Plus line editor.

TABLE 5
EmailResponse to Referral[a]

Primary Key: ReferralID

ReferralID	DealerID	EmailAddress	ResDateTime
10345	24145	ryan0983@aol.com	11/12/99 11:30 AM
10352	16287	low008@aol.com	11/15/99 1:36 PM
10379	405718	allen23@mci.com	12/11/99 5:00 PM
10382	24145	ryan0983@aol.com	11/15/99 12:20 PM
10394	16287	low008@aol.com	11/16/99 5:00 PM
10407	405718	allen23@mci.com	11/25/99 1:00 PM

[a] In the *EmailResponseToReferral* table, I have corrected what appears to be a typographical error in BJK. In the table presented in the original case, the date/time for referral ID 10394 was 11/15/99 5:00:00 PM. This would place the email referral 16 hours before the dealer self-reported referral, which is out of character for this dealer. Given that BJK also used 11/16/99 5:00:00 PM date and time in their teaching note, I regarded the 11/15/99 date appearing in the table in the case as a typographical error. The solutions I present use the 11/16/99 date for the time/date calculations.

After entering the database, the SQL prompt is all you will see on screen, apart from some information about the version of Oracle being used and the date, and it is where you will begin keying in the SQL commands you wish to use. To execute the command once it's been keyed in, you must either end the command with a semi-colon and press enter, or go to a new line, enter a forward slash (/), and then press enter. Either technique will tell the database to process the SQL code. For example, to see a list of tables that have already been created, you would enter the following command at the SQL prompt:

```
SQL>SELECT table_name FROM user_tables;
```

(In this example, table_name is the actual expression you should use. I am not using it here as a convention to represent some other, specific table name).

After pressing enter, the database will return a list of table names of tables that have previously been created. For convenience, I have capitalized SQL keywords here, but keep in mind that Oracle is not case sensitive, and the code may be either upper-case or lower-case, or any combination.

Using SQL to Build Tables

The first task required is to build the tables. To build the customer table, you will need to define the fields, the field type for each field, and the size of the field, if required by the data type. The following code creates the customer table and establishes the customer id as the primary key:

```
CREATE TABLE Customer
(CustomerID NUMBER (6,0) CONSTRAINT customer_custid_pk PRIMARY
    KEY,
FirstName VARCHAR (15),
LastName VARCHAR (15),
Street VARCHAR (25),
City VARCHAR (15),
PostCode CHAR (5),
```

PhoneWork VARCHAR (15),
PhoneHome VARCHAR (15),
EmailAddress VARCHAR (25))

The first line above uses the create table command to create the table, and then names the table "Customer." The second line identifies the first field to be created, the CustomerID field, and assigns it a Number data type, with a size of six digits in total. When assigning size to number fields in Oracle, you can specify the "precision" and "scale" of the number within parentheses as I have done above. The first number identifies the "precision," which is the total number of digits allowable for the number: in this case, six. The second number identifies the "scale," which is the number of digits to the right of the decimal: in this case, zero. So, for example, a field defined as NUMBER (5,2), would allow 123.45 as a legitimate value, but not 1234.5 or 1234.56.

Following the size definition, I have added a constraint. Constraints in Oracle apply controls to the data, and may be applied at the field level (as I have done) or at the table level (which I do in the next example). Constraints may be one of the five types listed below:

Constraint Type	Abbreviation	Function
Primary Key	pk	Sets a column or columns as primary key
Foreign Key	fk	Makes a column a foreign key
Check	cc or ck	Checks the field to be sure data entry conforms to a defined condition
Not null	nn	Ensures that the field has a value in it. Will not let field be blank
Unique	uk	Requires that every value in a column or columns be unique

Any constraint may be added to a field except the foreign key constraint, which must be added at the table level. There is no need in this case to define foreign keys, because solutions can be determined without them, but it is useful to have students create at least one so they can actively participate in developing a vital control feature over the database. Note that to add a constraint, you need to identify it with the CONSTRAINT keyword, followed by the constraint name (in this case, the constraint name is customer_custid_pk), and then the keywords identifying the type of constraint being added (in this case, PRIMARY KEY).

Each of the next lines of code creates the remaining fields in the customer table, and they are defined as VARCHAR or CHAR fields. VARCHAR fields store variable-length character data and have a maximum size limit of 4000 characters. CHAR fields store fixed-length character data and are limited to 2000 characters. The code necessary to create the dealer table is very similar to what was used to create the customer table and can be adapted with relatively simple modifications for that purpose.

Creating the ReferralToDealer Table

The ReferralToDealer table and the other remaining tables can be created in a similar manner to the customer table as well, but will require changing the field names and types as appropriate. I present the code necessary to create the ReferralToDealer table below. In it I illustrate how to define a foreign key constraint and set up a date field.

CREATE TABLE ReferralToDealer
(ReferralID NUMBER (5) CONSTRAINT referraltodealer_refid_pk PRIMARY KEY,

```
CustomerID NUMBER (6),
DealerID NUMBER (6),
RefDateTime DATE,
CONSTRAINT referraltodealer_custid_fk FOREIGN KEY (CustomerID)
REFERENCES Customer (CustomerID));
```

As you can see, there really isn't much involved in setting up a date field. You simply identify the field as a date field and move on; you do not need to specify a size, as the size is predetermined. The real issue—in fact, the critical technical problem in this case—comes when trying to populate the table with both the date and time in this single field. I address that issue in the next section. I should mention here that there is no requirement that the ReferralId, CustomerID, and DealerID all be number fields. You can make them character fields or VARCHAR fields as well. I have found, however, that it minimizes the problems students have with the assignment if you make them all similar type fields. By making them all number fields, or VARCHAR fields, students will be less likely to define a field as, say, a number in one table and the same field as a CHAR in another table, which may subsequently lead to trouble when they begin querying the database. Note, too, that it is not always necessary to identify the size of number fields with both precision and scale. In this case, I chose to identify only precision. Since none of the numbers have decimals, it will not create a problem. Finally, note the foreign key constraint reference. It first begins with the CONSTRAINT keyword, then follows with the constraint name (referraltodealer_custid_fk), the FOREIGN KEY keyword, the field name of the foreign key, and then a REFERENCES statement followed by the field it references in parentheses, which tells the database which table and field represent the corresponding primary key reference.

Populating Tables

Once the tables have been built, it is necessary to populate them with data. While this is a relatively easy task using Access, it is more difficult to accomplish using Oracle because of the requirement that both time and date be stored in a single field. In fact, storing the time and date in a single field is critical to successful completion of the case, and it is not intuitively obvious how to do this using Oracle.

It is useful here to point out to students that when creating the tables, we utilized SQL's data definition language (DDL) capabilities. When populating tables, or when modifying data, we are taking advantage of SQL's data manipulation language (DML) strengths.

There are several ways to populate tables using SQL. The simplest way is to create SQL DML code and then modify it for each line. The second way, using substitution variables, requires students to write an interactive SQL script that creates user prompts, which they can then use to input data. By using substitution variables, students learn to appreciate first-hand the purpose and need for data input controls, as well as the increase in efficiency afforded by the technique.

To populate the Customer table using substitution variables, I present and discuss the approach below.

```
INSERT INTO Customer (CustomerID, FirstName, LastName, Street, City,
    PostCode, PhoneWork, PhoneHome, EmailAddress)
VALUES(&CustomerID, '&FirstName', '&LastName', '&Street', '&City',
    '&PostCode', '&PhoneWork', '&PhoneHome', '&EmailAddress')
```

The script begins with the INSERT INTO command, and is followed by the name of the table you wish to populate. A list of the field names within parentheses, all separated by commas, follows. After this, we have to identify the particular data values to be inserted, but in this case, rather than identifying the data values themselves, we identify the field preceded by an ampersand. CHAR, VARCHAR, and DATE fields all need to be surrounded by single quotes as shown above. NUMBER fields do not need to be surrounded with single quotes.

When the user runs the script, Oracle returns user prompts asking for each particular data value as shown below.

```
SQL> /
Enter value for customerid: 342512
Enter value for firstname:
```

The example above shows the first data item entered at the prompt, followed by the prompt for the second data item (firstname). After keying in each value, the user presses enter then continues this routine until finished with the record. The entire process for entering the second Customer record is presented in Figure 2.

When finished, Oracle responds with "1 row created." To add another record, all the user needs to do is to run the script again. This way, data input can be accomplished efficiently, while the substitution variables help minimize input errors.

To illustrate how to populate tables without using substitution variables, and how to handle the critical date/time issue, I present the SQL code below for inserting the first record of the ReferralToDealer table.

FIGURE 2
Substitution Variable Code and Input for the Customer Table

```
SQL> ed
Wrote file afiedt.buf

  1  INSERT INTO Customer (CustomerID, FirstName, LastName, Street, City, PostCode,
  2   PhoneWork, PhoneHome, EmailAddress)
  3  VALUES(&CustomerID, '&FirstName', '&LastName', '&Street', '&City', '&PostCode',
  4*  '&PhoneWork', '&PhoneHome', '&EmailAddress')
SQL> /
Enter value for customerid: 342525
Enter value for firstname: Daniel
Enter value for lastname: Lowell
Enter value for street: 225 Burbank Dr.
Enter value for city: Atlanta
Enter value for postcode: 30314
old   3: VALUES(&CustomerID, '&FirstName', '&LastName', '&Street', '&City', '&PostCode',
new   3: VALUES(342525, 'Daniel', 'Lowell', '225 Burbank Dr.', 'Atlanta', '30314',
Enter value for phonework: (404) 567-2245
Enter value for phonehome: (404) 514-8898
Enter value for emailaddress: low008@aol.com
old   4:  '&PhoneWork', '&PhoneHome', '&EmailAddress')
new   4:  '(404) 567-2245', '(404) 514-8898', 'low008@aol.com')

1 row created.
```

INSERT INTO ReferralToDealer (ReferralID, CustomerID, DealerID, RefDateTime)
VALUES (10345, 342512, 24145, TO_DATE('10-Nov-99 10:00:00' , 'DD-MON-RR HH24:MI:SS'))

As in the example above, the statement begins with the INSERT INTO command, followed by a list of the fields in the table. It differs in the VALUES statement, however, in that it lists the specific values for each corresponding field. In this case, the first three values corresponding to ReferralId, CustomerId, and DealerId, are not surrounded by quotation marks because they are all number fields. Note that in order to place both the referral date and referral time in one field, it is necessary to use the TO_DATE command. After the TO_DATE command, we open parentheses and then identify the date and time using the exact format above. Dates must be input in Day-Mon-Year format. The time is input according to a 24-hour clock in an hour-min-sec format. After surrounding the date and time values with single quotes, it is necessary to identify the format of the input for Oracle using 'DD-MON-RR HH24:MI:SS'. Failure to specify a 24 hour clock (with a "24" after HH) will result in confusion between AM and PM in the database. The actual input is illustrated in Figure 3.

After executing the SQL statement, Oracle will return a message saying "1 row created." To modify the statement, reopen the editor, change the data values, and then execute the SQL statement again. Each subsequent row can be entered in the same manner. Note that it is not possible to follow one INSERT INTO statement with multiple VALUES statements. The data from all other tables can be input using either of these two approaches.

PART 1: CASE QUESTIONS CONCERNING AUTOMOBILE DEALER COMPLIANCE

In the first part of the case, the auto manufacturer wishes to query the database to determine how well its local dealers are complying with the requirements that they respond in a timely manner to customer referrals. The manufacturer had been getting complaints from some potential customers that they were not getting contacted in a timely manner, and so wanted to determine the length of time from the referral to the customer contact.

Case Question 1: Determine response times for dealer-reported responses.

Determining the dealer-reported response times involves joining the *DealerResponseToReferral* table with the *ReferralToDealer* table and subtracting the referral date

(*continued on next page*)

FIGURE 3
Code to Populate Tables with Date and Time Field

```
SQL> ed
Wrote file afiedt.buf

  1  INSERT INTO ReferralToDealer (ReferralID, CustomerID, DealerID, RefDateTime)
  2* VALUES (10345, 342512, 24145, TO_DATE('10-Nov-99 10:00:00','DD-MON-RR HH24:MI:SS'))
SQL> /

1 row created.
```

and time from the response date and time. Since the results will be in days, it is necessary to multiply the difference by 24 in order to place the answer in hours. While not specifically required, it is not a bad idea to have students retrieve the referral date and time and the response date and time as well, in order to check the difference calculation. The following SQL code will produce the desired results.

```
SELECT ReferralToDealer.ReferralID, DealerResponseToReferral.DealerID,
TO_CHAR(RefDateTime, 'DD-MON-RR HH24:MI:SS') AS ReferralDate,
    TO_CHAR(ResDateTime, 'DD_MON-RR HH24-MI:SS') AS ResponseDate,
(DealerResponseToReferral.ResDateTime – ReferralToDealer.RefDateTime)*24
    AS Response_Time
FROM DealerResponseToReferral, ReferralToDealer
WHERE DealerResponseToReferral.ReferralID = ReferralToDealer.ReferralID
```

In the SQL statement above, I request the database to return the ReferralID, the DealerID, the referral date and time, the response date and time, and the difference between the referral and response times. Since there are multiple tables, I used the tablename. fieldname convention in identifying the fields in the SELECT statement. By default, Oracle displays only the date when you select a date/time field, so in order to view both the date and time, it is necessary to use the TO_CHAR command, which converts the field into characters that it displays. The TO_CHAR command is immediately followed by field name and format instructions for the date and time within parentheses. Outside the parentheses, I use the AS command to apply a field alias. The AS command allows you to rename a field (or table) for convenience. The FROM clause lists the tables from which the data are to be retrieved, and finally, the WHERE clause formally joins the two tables based on the common field between them.

At this point, it is obvious that the long table names are unwieldy and inconvenient. Fortunately, Oracle allows you to use table aliases to reduce the unnecessary keying required by long table names. The following code is identical to the code above with the exception that it illustrates the use of table aliases:

```
SELECT r.ReferralID, d.DealerID,
TO_CHAR(RefDateTime, 'DD-MON-RR HH24:MI:SS') AS ReferralDate,
    TO_CHAR(ResDateTime, 'DD_MON-RR HH24-MI:SS') AS ResponseDate,
(d.ResDateTime – r.RefDateTime)*24 AS Response_Time
FROM DealerResponseToReferral d, ReferralToDealer r
WHERE d.ReferralID = r.ReferralID
```

Figure 4 illustrates the code and results from SQL Plus:

Generally, table aliases are used to reduce keystrokes; there is no requirement that they be a single letter, but the shorter they are, the better. Notice in the example above, I replaced the *DealerResponseToReferral* table references with the letter *d*, and the *ReferralToDealer* table references with the letter *r*. All that is required to enact the alias is to place the alias immediately after the table name in the FROM clause. To minimize the keying required by the long table names, I will illustrate all remaining solutions using table aliases.

Case Question 2: Find the average response time by dealer for self-reported times.

The following SQL query will produce the desired results:

FIGURE 4
SQL Code and Results Displaying Both Date and Time in Response Time

```
SQL> SELECT r.ReferralId, d.DealerID,
  2     TO_CHAR(RefDateTime, 'DD-MON-RR HH24:MI:SS') AS ReferralDate,
  3     TO_CHAR(ResDateTime, 'DD-MON-RR HH24:MI:SS') AS ResponseDate,
  4     (d.ResdateTime-r.RefDateTime)*24 AS Response_Time
  5  FROM DealerResponseToReferral d, ReferralToDealer r
  6  WHERE d.ReferralID=r.ReferralID;

REFERRALID   DEALERID REFERRALDATE        RESPONSEDATE        RESPONSE_TIME
---------- ---------- ------------------ ------------------ -------------
     10345      24145 10-NOV-99 10:00:00 12-NOV-99 11:30:00          49.5
     10352      16287 12-NOV-99 13:15:00 12-NOV-99 16:00:00          2.75
     10363      37269 15-NOV-99 10:00:00 15-NOV-99 16:30:00           6.5
     10379     405718 11-DEC-99 11:35:00 11-DEC-99 16:00:00    4.41666667
     10382      24145 12-NOV-99 11:00:00 15-NOV-99 12:20:00    73.3333333
     10394      16287 15-NOV-99 14:20:00 16-NOV-99 09:00:00    18.6666667
     10407     405718 25-NOV-99 10:00:00 25-NOV-99 13:00:00             3

7 rows selected.
```

FIGURE 5
SQL Code and Output Displaying Average Dealer Response

```
SQL> SELECT d.DealerID, AVG(d.ResDateTime - r.RefDateTime)*24 AS AvgResponse
  2  FROM DealerResponseToReferral d, referralToDealer r
  3  WHERE d.ReferralID = r.ReferralID
  4  GROUP BY d.DealerID
  5  /

  DEALERID AVGRESPONSE
---------- -----------
     16287  10.7083333
     24145  61.4166667
     37269         6.5
    405718  3.70833333
```

```
SELECT d.DealerID, AVG(d.ResDateTime – r.RefDateTime)*24 AS AvgResponse
FROM DealerResponseToReferral d, referralToDealer r
WHERE d.ReferralID = r.ReferralID
GROUP BY d.DealerID
```

Figure 5 displays the query and results:

In order to find the average response time by dealer, it is necessary to use the AVG function, one of Oracle's built-in functions. In this query, I select the DealerID, and then calculate the difference between the referral date/time and the response date/time. Taking the average of the difference is simply a matter of preceding the parentheses with AVG. I then apply a field alias called *AvgResponse* to the field.

Since the question asks for dealer response averaged by dealers, it is necessary to include a GROUP BY clause at the end of the SQL statement in order to calculate averages for each dealer ID. If you include other fields in the select statement besides the field(s) you are grouping on (in this case, DealerID) and the field to be averaged, you will get an error message from Oracle saying that the field is not part of an aggregate function. It is

important to understand that when using Oracle's group functions (Sum, Count, Avg, Min, and Max), you may not include fields in the SELECT statement that are neither the grouping field(s) nor the evaluated field. I mention this because students are likely to have trouble with the group functions, and this is the most common error they make.

Case Question 3: Determine response times for dealers' emailed responses

This question requires students to retrieve dealers' email response times in the same manner as they retrieved self-reported response times in Question 1. The solution is nearly identical to the solution in Question 1 except that the *EmailResponseToReferral* table is referenced rather than the *DealerResponseToReferral* table. Since the solution required is similar to Question 1, Figure 6 illustrates the query and presents the results.

Note that before keying in the query, I set the column width for the Response_Time field to be equal to three digits, and two decimal places (Col Response_Time FORMAT 999.90) The COL statement allows me to format the decimal places for the Response_Time field to two digits, presenting cleaner results than the results presented in case Questions 1 and 2.

Case Question 4: Find the average response time for dealers' emailed responses

This question asks students to find the average email response time grouped by dealer, and is very similar to Question 2 above. Again, the only aspect of the solution that must change is the reference to the *EmailResponseToReferral* table rather than the *DealerResponseToReferral* table. In the solution below, I changed the table reference, as well as the field alias for the average email response field. The solution and results are presented in Figure 7.

Case Question 5: Compare dealers' self-reported response times to dealers' emailed response times

This question requires students to refer to both the *DealerResponseToReferral* table and the *EmailResponseToReferral* table. The business reason for making the comparison is to compare dealers' reported response times with some objective measure of their actual response times (using the email time stamps as proxies for the true response times). If dealers' self-reporting response times are substantially earlier than the email response times, then it may be that dealers are not complying with the manufacturer's policies and are misrepresenting their actual behavior. Of course it may be that dealers are not careful to report accurate times so that a dealer self-reported response may be reported as before, or after, the email response time. The easiest way to make the comparison is for students to subtract the ResDateTime field reported in the *EmailResponseToReferral* table from the ResDateTime field reported in the *ResponseToReferral* table and calculate the number of hours by which the two times differ. If the resulting numbers are large positive values, then the dealer may be misrepresenting the timeliness of its response.

Figure 8 displays the query and results:

In the query above, I select the ReferralId and DealerId from the *DealerResponseToReferral* table, and the ResDateTime fields from both the *DealerResponseToReferral* and *EmailResponseToReferral* tables. These date/time selections are defined in the TO_CHAR statements, so that the date and time of each response will display. Line 4 above performs the calculation of the difference in reported response times, and line 7 sorts the results in descending order first by the Time_Difference field, then by the DealerId field. Note that the results indicate that for dealer 16287, reported response times were 69.6 hours and eight hours before the corresponding actual email responses to the customer. If email responses

FIGURE 6
SQL Code Used to Reformat Response Time Column and Calculate Dealer Response Times

```
SQL> col Response_Time FORMAT 999.90
SQL> SELECT r.ReferralID, e.DealerID,
  2  TO_CHAR(r.RefDateTime,'DD-MON-RR  HH24:MI:SS') AS ReferralDate,
  3  TO_CHAR(e.ResDateTime, 'DD-MON-RR  HH24:MI:SS') AS ResponseDate,
  4  (e.ResDateTime - r.RefDateTime)*24 AS Response_Time
  5  FROM EmailResponseToReferral e, ReferralToDealer r
  6  WHERE e.ReferralID = r.ReferralID
  7  /

REFERRALID   DEALERID REFERRALDATE         RESPONSEDATE         RESPONSE_TIME
---------- ---------- -------------------- -------------------- -------------
     10345      24145 10-NOV-99  10:00:00 12-NOV-99  11:30:00         49.50
     10352      16287 12-NOV-99  13:15:00 15-NOV-99  13:36:00         72.35
     10379     405718 11-DEC-99  11:35:00 11-DEC-99  17:00:00          5.42
     10382      24145 12-NOV-99  11:00:00 15-NOV-99  12:20:00         73.33
     10394      16287 15-NOV-99  14:20:00 16-NOV-99  17:00:00         26.67
     10407     405718 25-NOV-99  10:00:00 25-NOV-99  13:00:00          3.00

6 rows selected.
```

accurately capture the true response times, then this dealer is misreporting its self-reported response times.

Case Question 6: Find the average time difference between dealers' self-reported and emailed responses

Here again it is necessary to use Oracle's built-in functions to find the solution. Since the question seeks to determine the average time difference by dealer, it is necessary to group by dealer while using the AVG function. The following query and results illustrates one approach to the question (Figure 9).

In Figure 9, I demonstrate how to join three tables and group on multiple fields. Line 1 selects the Dealer ID field from the *DealerResponseToReferral* table, the dealer name from the *Dealer* table, and then calculates the average response difference (in hours) between the ResDateTime fields from the *DealerResponseToReferral* and the *EmailResponseToReferral* tables. Note that, in order to convert the calculation to positive numbers, I used the absolute value command (ABS) before the AVG function. Since three tables were joined, two primary key/foreign key links were specified in line 4. Finally, line 5 indicates that the data were grouped on two fields. Oracle allows grouping on multiple fields so long as the fields pertain to the same data item (in this case, the average time difference). Oracle groups according to the unique combination of fields within the GROUP BY statement.

PART 2: CASE QUESTIONS CONCERNING MANUFACTURER COMPLIANCE

In this section of the case, the verification emphasis shifts from the auto manufacturer, which in Part 1 was attempting to verify that dealers responded to customer inquiries within its predetermined parameters, to the dealers, who are attempting to verify that the manufacturer is making customer referrals to the nearest dealer to the customer. The method used in the case to determine geographical proximity is a comparison of the dealers' postal

FIGURE 7
SQL Code Used to Calculate Average Email Response Times

```
SQL> SELECT e.DealerID, AVG(e.ResDateTime - r.RefDateTime)*24 AS AvgEmailResp
  2  FROM EmailResponseToReferral e, referralToDealer r
  3  WHERE e.ReferralID = r.ReferralID
  4  GROUP BY e.DealerID;

  DEALERID AVGEMAILRESP
---------- ------------
     16287   49.5083333
     24145   61.4166667
    405718   4.20833333
```

FIGURE 8
SQL Code Used to Calculate the Difference between Dealer Response and Email Response

```
SQL> ed
Wrote file afiedt.buf

  1  SELECT d.ReferralID, d.dealerid,
  2  TO_CHAR(d.ResDateTime, 'DD-MON-RR HH24:MI:SS') AS DealerTime,
  3  TO_CHAR(e.ResDateTime, 'DD-MON-RR HH24:MI:SS') AS EmailTime,
  4  (e.ResDateTime-d.ResDateTime)*24 AS Time_Difference
  5  FROM DealerresponseToReferral d, EmailResponseToReferral e
  6  WHERE d.ReferralID=e.ReferralID
  7* ORDER BY Time_Difference DESC, d.dealerid DESC
SQL> /

REFERRALID   DEALERID DEALERTIME         EMAILTIME          TIME_DIFFERENCE
---------- ---------- ------------------ ------------------ ---------------
     10352      16287 12-NOV-99 16:00:00 15-NOV-99 13:36:00            69.6
     10394      16287 16-NOV-99 09:00:00 16-NOV-99 17:00:00               8
     10379     405718 11-DEC-99 16:00:00 11-DEC-99 17:00:00               1
     10407     405718 25-NOV-99 13:00:00 25-NOV-99 13:00:00               0
     10345      24145 12-NOV-99 11:30:00 12-NOV-99 11:30:00               0
     10382      24145 15-NOV-99 12:20:00 15-NOV-99 12:20:00               0

6 rows selected.
```

codes with those of their customers. While the case asks students to consider alternative ways of determining the proximity of dealers and customers in Part 2B (BJK), and presents a postal code map for that purpose, the initial case questions themselves focus on determining the numerical difference between postal codes and using this difference as a proxy for the geographical distance.

Case Question 1: For each referral, determine the difference between the postal codes of the customer and the dealer receiving the referral.

The first question asks students to determine the difference between the customer's postal code and the postal code of the dealer to which he or she was referred. To find the solution, students will need to query the *Customer* table to find the customer's postal code, the *Dealer* table, to find the dealer's postal code, and the *ReferralToDealer* table to find

FIGURE 9
Average Differences between Dealer and Email Responses

```
SQL> ed
Wrote file afiedt.buf

  1  SELECT d.DealerID, dlr.Name, ABS(AVG(d.ResDateTime-e.ResDateTime)*24)
  2     AS "Average Response Difference"
  3  FROM DealerResponseToReferral d, EmailResponseToReferral e, Dealer dlr
  4  WHERE d.ReferralID = e.ReferralID and e.DealerID = dlr.DealerID
  5* GROUP BY d.DealerID, dlr.Name
SQL> /

  DEALERID NAME                 Average Response Difference
---------- -------------------- ---------------------------
     16287 Buckhead Auto                              38.8
     24145 Paul Light                                    0
    405718 Town Touring                                 .5
```

the referral linking the customer to the dealer. Figure 10 presents one way of retrieving and formatting the desired information:

The above query selects the referral id, the customer's first and last name, the dealer's name, the customer and dealer postal codes, and then calculates the numerical difference between the respective postal codes.

I took the opportunity with this query to illustrate certain useful formatting and querying techniques. First, the four COL statements are used to format the results. The COL keyword

FIGURE 10
Calculation of Postal Code Differences Showing Column Formatting and Concatenation Techniques

```
SQL> COL Customer FORMAT a17
SQL> COL Dealer FORMAT a15
SQL> COL "Cust Zip" FORMAT a9
SQL> COL "Dealer Zip" FORMAT a10
SQL>  SELECT r.ReferralID, c.FirstName || ' ' || c.LastName AS Customer,
  2   d.Name AS Dealer, c.PostCode AS "Cust Zip", d.PostCode AS "Dealer Zip",
  3   c.PostCode-d.PostCode AS ZipDiff
  4   FROM Customer c, Dealer d, ReferralToDealer r
  5   WHERE c.CustomerID = r.CustomerID AND d.DealerID = r.DealerID
  6   ORDER BY ZipDiff ASC;

REFERRALID CUSTOMER          DEALER          Cust Zip  Dealer Zip    ZIPDIFF
---------- ----------------- --------------- --------- ---------- ----------
     10407 Cathy Allen       Town Touring    30303     30303               0
     10379 Cathy Allen       Town Touring    30303     30303               0
     10382 Ryan Hong         Paul Light      30344     30342               2
     10345 Ryan Hong         Paul Light      30344     30342               2
     10394 Daniel Lowell     Buckhead Auto   30314     30305               9
     10352 Daniel Lowell     Buckhead Auto   30314     30305               9
     10363 Terrel Thomas     Bob Motoring    30360     30349              11

7 rows selected.
```

followed by the field name, the FORMAT command, and the desired width, formats the specified column to the desired width. For example, the "COL Customer FORMAT a17" statement formats the resulting field named customer to 17 alpha characters wide. Second, in the first line of the SELECT statement, I join the LastName field from the customer table together to the FirstName field from the customer table in a process known as "concatenation." To concatenate fields in Oracle, you use the concatenation operator, the two vertical bars (| |), after the first field and before the second field you wish to join. In between the two concatenator symbols, I placed two single quotation marks with a space between them. This places a space between the two fields in the query results. You can concatenate any fields and place any combination of spaces or characters between them following this general approach. I then renamed the concatenated field "Customer." Third, the field alias for the customer postal code is surrounded by double quotation marks because there is a space in the alias. Note that the alias for dealer name (d.Name in line 2 is given the alias "Dealer") has no quotation marks, while the alias for both the customer postal code and the dealer postal codes are surrounded by double quotation marks because they both have spaces in their aliases, "Cust Zip" and "Dealer Zip," respectively. Finally, the results were sorted in ascending order by the postal code difference.

The results indicate that some customer referrals are to dealers within the same postal codes, while others differ by what appear to be relatively large numbers. By themselves, however, there is no way to put the results in perspective. Is 11 a large difference, or is it a relatively small difference?

Case Question 2: For each referral, determine the minimum postal code difference over all dealers.

This question asks students to compare, for each referral, the customer postal code with every dealer's postal code, and then to find the dealer that minimizes the difference for that particular referral. In effect, it places the results of the previous query in context.

A simple modification of the previous query will create the combination of all dealers' postal codes compared to the customer's postal code for each referral. Since there were seven referrals and six dealers, there will be 6 × 7 rows in the results. In order to get the database to relate all combinations of referral and dealer, it is only necessary to drop the join between the dealer and the referral in the WHERE statement. Leaving the clause WHERE c.CustomerID = r.CustomerID joins the referral to the customer. Since there is now no corresponding join between the dealer and the referral, all the dealers will be related in the results to each customer referral. Figure 11 displays the SQL query and the results.

The results in Figure 11 answer the question because the minimum distance for each referral can be obtained by scanning the results, but it is hardly an elegant solution. The query can, and should, be further refined to present only the minimum values in a much more efficient manner.

To modify the query further, drop all the fields from the SELECT statement except the ReferralID and the calculation of the difference. To find the minimum value of all dealer/referral combinations, simply express the calculation field as MIN(ABS(c.PostCode – d.PostCode) and give it an alias. Finally, add a GROUP BY statement after the WHERE statement since we want to retrieve the minimum postal code difference for each referral. Figure 12 illustrates the modified query with results.

Achieving this result answers the case question directly, and is certainly a more direct answer than the earlier result.

FIGURE 11
Preliminary SQL Code Used to Determine Minimum Postal Code Differences Across All Dealers

```
SELECT r.ReferralID, c.FirstName || ' ' || c.LastName AS Customer,
  2 d.Name AS Dealer, c.PostCode AS "Cust Zip", d.PostCode AS "Dealer Zip",
  3 ABS(c.PostCode-d.PostCode) AS ZipDiff
  4 FROM Customer c, Dealer d, ReferralToDealer r
  5 WHERE c.CustomerID = r.CustomerID
  6 ORDER BY r.ReferralID ASC, ZipDiff ASC;
```

REFERRALID	CUSTOMER	DEALER	Cust Zip	Dealer Zip	ZIPDIFF
10345	Ryan Hong	Paul Light	30344	30342	2
10345	Ryan Hong	Neal Pope Motorcar	30344	30341	3
10345	Ryan Hong	Bob Motoring	30344	30349	5
10345	Ryan Hong	Afford Auto	30344	30339	5
10345	Ryan Hong	Buckhead Auto	30344	30305	39
10345	Ryan Hong	Town Touring	30344	30303	41
10352	Daniel Lowell	Buckhead Auto	30314	30305	9
10352	Daniel Lowell	Town Touring	30314	30303	11
10352	Daniel Lowell	Afford Auto	30314	30339	25
10352	Daniel Lowell	Neal Pope Motorcar	30314	30341	27
10352	Daniel Lowell	Paul Light	30314	30342	28

REFERRALID	CUSTOMER	DEALER	Cust Zip	Dealer Zip	ZIPDIFF
10352	Daniel Lowell	Bob Motoring	30314	30349	35
10363	Terrel Thomas	Bob Motoring	30360	30349	11
10363	Terrel Thomas	Paul Light	30360	30342	18
10363	Terrel Thomas	Neal Pope Motorcar	30360	30341	19
10363	Terrel Thomas	Afford Auto	30360	30339	21
10363	Terrel Thomas	Buckhead Auto	30360	30305	55
10363	Terrel Thomas	Town Touring	30360	30303	57
10379	Cathy Allen	Town Touring	30303	30303	0
10379	Cathy Allen	Buckhead Auto	30303	30305	2
10379	Cathy Allen	Afford Auto	30303	30339	36
10379	Cathy Allen	Neal Pope Motorcar	30303	30341	38

REFERRALID	CUSTOMER	DEALER	Cust Zip	Dealer Zip	ZIPDIFF
10379	Cathy Allen	Paul Light	30303	30342	39
10379	Cathy Allen	Bob Motoring	30303	30349	46
10382	Ryan Hong	Paul Light	30344	30342	2
10382	Ryan Hong	Neal Pope Motorcar	30344	30341	3
10382	Ryan Hong	Bob Motoring	30344	30349	5
10382	Ryan Hong	Afford Auto	30344	30339	5
10382	Ryan Hong	Buckhead Auto	30344	30305	39
10382	Ryan Hong	Town Touring	30344	30303	41
10394	Daniel Lowell	Buckhead Auto	30314	30305	9
10394	Daniel Lowell	Town Touring	30314	30303	11
10394	Daniel Lowell	Afford Auto	30314	30339	25

REFERRALID	CUSTOMER	DEALER	Cust Zip	Dealer Zip	ZIPDIFF
10394	Daniel Lowell	Neal Pope Motorcar	30314	30341	27
10394	Daniel Lowell	Paul Light	30314	30342	28
10394	Daniel Lowell	Bob Motoring	30314	30349	35
10407	Cathy Allen	Town Touring	30303	30303	0
10407	Cathy Allen	Buckhead Auto	30303	30305	2
10407	Cathy Allen	Afford Auto	30303	30339	36
10407	Cathy Allen	Neal Pope Motorcar	30303	30341	38
10407	Cathy Allen	Paul Light	30303	30342	39
10407	Cathy Allen	Bob Motoring	30303	30349	46

```
42 rows selected.

SQL>
```

FIGURE 12
Modified SQL Code Used to Determine Minimum Postal Code Differences Across All Dealers

```
SQL> SELECT r.ReferralID,  MIN(ABS(c.PostCode-d.PostCode)) AS MinZipDiff
  2  FROM Customer c, Dealer d, ReferralToDealer r
  3  WHERE c.CustomerID = r.CustomerID
  4  GROUP BY r.ReferralID
  5  ORDER BY r.ReferralID ASC
  6  /

REFERRALID MINZIPDIFF
---------- ----------
     10345          2
     10352          9
     10363         11
     10379          0
     10382          2
     10394          9
     10407          0

7 rows selected.
```

Case Question 3: For each referral, determine whether the referred dealer corresponds to the minimum postal code difference.

This question is the final case question involving hands-on querying, and asks students to compare the existing difference in postal codes from Part 2, Question 1 with the minimum possible postal code difference for each referral from Part 2, Question 2. There are several possible approaches to addressing the question.

The solution I present here requires that students first create a new table from the results of the previous query in Part 2, Question 2. The advantage to this approach, besides allowing you to introduce sub-queries, is that this allows students to name the new table and avoids having to write a much more complicated sub-query later on. Figure 13 illustrates how to create a table from the results of the previous query.

FIGURE 13
SQL Code Used to Save Minimum Postal Code Differences as a New Table

```
SQL> ed
Wrote file afiedt.buf

  1  CREATE TABLE MinDiff AS
  2  SELECT r.ReferralID,  MIN(ABS(c.PostCode-d.PostCode)) AS MinZipDiff
  3  FROM Customer c, Dealer d, ReferralToDealer r
  4  WHERE c.CustomerID = r.CustomerID
  5  GROUP BY r.ReferralID
  6* ORDER BY r.ReferralID ASC
SQL> /

Table created.
```

In this query, I begin with the CREATE TABLE command, followed by the name I wish to give to the new table—in this case "MinDiff." Rather than following the command with the new fields and data types—all the data definition commands discussed earlier—I follow it with "AS," and then the query from Part 2, Question 2 above. The entire SELECT statement becomes, in effect, a second query within the original query, a sub-query. Sub-queries may be used in the SELECT statement, following the FROM statement, or in the WHERE statement as well. Oracle always evaluates the sub-query first, then the host query.

In this case, the SELECT statement and the other commands in the sub-query generate results having two fields, the referral id, and the calculated field representing the minimum possible differences in dealer/referral postal codes called MinZipDiff. Once those results are generated, Oracle then executes the CREATE TABLE command, which takes those results and creates a new table named "MinDiff" from them.

The next step is to reproduce the results from Part 2, Question 1, and then join the tables used in generating those results to the newly created MinDiff table. Because the MinDiff results (the results from Question 2) have been saved to a new table, they can now be referenced easily. Figure 14 illustrates the query and results that satisfy the requirements for Question 3.

The SELECT statement requests the ReferralId field from the *ReferralToDealer* table, the dealer name from the *Dealer* table, then calculates the existing difference between customer and dealer postal codes, naming the calculated field "ZipDiff." The next selection is the MinZipDiff field from the newly created *MinDiff* table, followed by another calculation of the difference between the existing zip code difference (ZipDiff) and the MinZipDiff field, which is given the alias "ZipResult." The FROM statement lists the four tables from which the data are drawn with their table aliases, and the WHERE statement joins the four tables on their common fields.

The results indicate that there is no difference between the minimum possible postal code differences and the existing postal code differences for each referral. Apparently, the

FIGURE 14
SQL Code and Output Showing Comparison of Actual and Possible Minimum Postal Code Differences

```
SQL> ed
Wrote file afiedt.buf

  1  SELECT r.ReferralID, d.Name AS Dealer, ABS(c.PostCode  -  d.PostCode) AS ZipDiff,
  2  m.MinZipDiff, ((c.PostCode-d.PostCode) - m.MinZipDiff) AS ZipResult
  3  FROM ReferralToDealer r, Customer c, Dealer d, MinDiff m
  4  WHERE r.CustomerID=c.CustomerID AND d.DealerID = r.DealerID
  5* AND m.ReferralID=r.ReferralID
SQL> /

REFERRALID DEALER              ZIPDIFF MINZIPDIFF  ZIPRESULT
---------- ------------------ ------- ---------- ----------
     10345 Paul Light               2          2          0
     10352 Buckhead Auto            9          9          0
     10363 Bob Motoring            11         11          0
     10379 Town Touring             0          0          0
     10382 Paul Light               2          2          0
     10394 Buckhead Auto            9          9          0
     10407 Town Touring             0          0          0

7 rows selected.
```

auto manufacturer is abiding by the agreement to refer customers to the nearest dealership, at least when the nearest dealer is defined by difference in postal codes.

III. CONCLUSION

This paper offered a follow-up note to BKJ's case focused on having students develop queries verifying mutual compliance between an auto manufacturer and its dealers. The original case presents query strategies and solutions developed for MS Access using a QBE approach. This follow-up note offers query strategies and solutions developed for Oracle using SQL-Plus. Teaching the case using SQL rather than a QBE approach makes it a more challenging assignment for students, but it also offers several distinct pedagogical benefits: it reinforces the logic necessary to developing successful queries, it exposes students to the most widely-used business database language, it illustrates for students the data definition, manipulation, and querying capabilities inherent in SQL, and it acquaints students with a major database management system actually used for large-scale, robust accounting systems.

REFERENCES

Borthick, A. F. 1996. Helping accountants learn to get the information managers want: The role of the accounting information systems course. *Journal of Information Systems* 10 (Fall): 75–85.

———, D. R. Jones, and R. Kim. 2001. Developing database query proficiency: Assuring compliance for responses to website referrals. *Journal of Information Systems* 15 (Spring): 35–56.

Gelinas, U. J., S. G. Sutton, and J. E. Hunton. 2005. *Accounting Information Systems*. 6th edition. Mason, OH: Thomson/Southwestern.

Romney, M. B., and P. J. Steinbart. 2003. *Accounting Information Systems*. 9th edition. Upper Saddle River, NJ: Prentice Hall.