Appendix J

Web Database Development with ColdFusion

Preview

This appendix examines the basics of web database development with ColdFusion, an important web application server tool for creating web database front ends. This appendix also explores some of the reasons why and how web application development differs from more traditional database application development.

Data Files and Available Formats

	MS Access	Oracle	MS SQL	My SQL	MS Access	Oracle	MS SQL	My SQL
Orderdb	✓							

Data Files Available on cengagebrain.com

J-1 Using a Web-to-Database Production Tool: ColdFusion

To understand how web-to-database interfaces work, you need to know how they are created and to see them in action. In this section, you have a chance to try Adobe ColdFusion, one of a new breed of products known as web application servers. A **web application server** is a middleware application that expands the functionality of web servers by linking them to a wide range of services, such as databases, directory systems, and search engines. The web application server also provides a consistent run-time environment for web applications.

ColdFusion application middleware can be used to:

- Connect to and query a database from a webpage.
- Present database data in a webpage using various formats.
- Create dynamic web search pages.
- Create webpages to insert, update, and delete database data.
- Define required and optional relationships.
- Define required and optional form fields.
- Enforce referential integrity in form fields.
- Use simple and nested queries and form select fields to represent business rules.

Note

Although ColdFusion has a wide range of features, the purpose of this section is to show you how to create and use a simple, yet useful web-to-database interface. You can learn additional ColdFusion features by tapping into its detailed and well-organized online documentation (*www.adobe.com*).

ColdFusion has several important characteristics:

- It is a powerful and stable software product that can be used to produce and support even the most complex web-to-database access solutions.
- In spite of its power, it is a developer- and user-friendly product. ColdFusion has a strong and growing corporate presence. Using ColdFusion, you can get some hands-on experience with the web-to-database environment, while improving the marketability of your knowledge.
- Adobe offers a free 30-day evaluation version of the latest ColdFusion software, which can be downloaded from *www.adobe.com*. Because ColdFusion includes a complete set of online documentation with full working demo applications that illustrate all of the functionality of the product, you will incur no documentation charges.

ColdFusion is, of course, not the only player in the web application server market, but products from different vendors tend to have many similar features.

web application server
A middleware application that expands the functionality of web servers by linking them to a wide range of services, such as databases, directory systems, and search engines.

Web application servers provide features such as:

- An integrated development environment with session management and support for persistent application variables.
- Security and authentication of users through user IDs and passwords.
- Computational languages to represent and store business logic in the application server.
- Automatic generation of HTML pages integrated with Java, JavaScript, VBScript, ASP, and so on.
- Performance and fault-tolerant features.
- Database access with transaction management capabilities.
- Access to multiple services, such as file transfers (FTP), database connectivity, email, and directory services.

All of these products offer the ability to connect web servers to multiple data sources and other services. These products vary in terms of the range of available features, robustness, scalability, ease of use, compatibility with other web and database tools, and extent of the development environment.

J-1a How ColdFusion Works

ColdFusion has been described as a full-fledged web application server that provides hooks to databases, email systems, directory systems, search engines, and so on. To do its job, ColdFusion provides a server-side markup language (HTML extensions, or **tags**) known as the **ColdFusion Markup Language (CFML)**, which is used to create ColdFusion application pages known as *scripts*. A **script** is a series of instructions executed in *interpreter* mode. The script is a plain-text file that is not compiled like COBOL, C++, or Java. ColdFusion scripts contain the code that is required to connect, query, and update a database from a web front end.

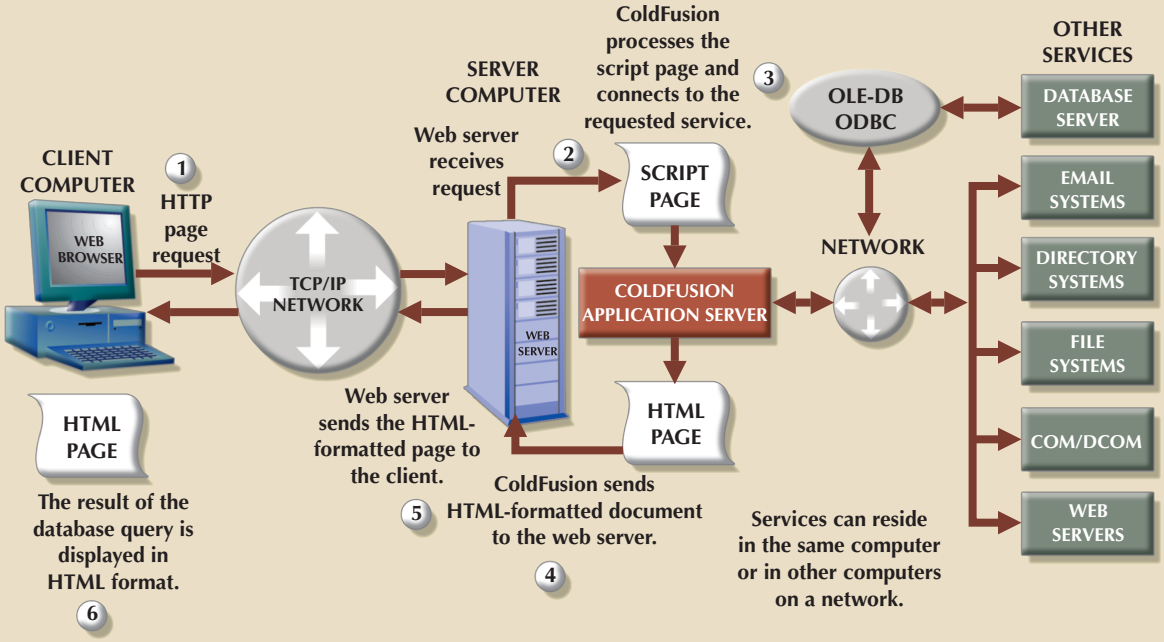
ColdFusion scripts contain a combination of HTML, ColdFusion tags, and when necessary, Java, JavaScript, or VBScript. ColdFusion tags start with <CF and may include an opening and closing component such as <CFQUERY> (begin a query) and </CFQUERY> (end a query). ColdFusion scripts are saved in files with .cfm extensions. When a client browser requests a .cfm page from the web server, the ColdFusion application server executes the .cfm script instructions and sends the resulting output—in HTML format—to the web server. The web server then sends the document to the client computer for display. Figure J.1 shows the application server components and actions.

tag
In markup languages such as HTML and XML, a command inserted in a document to specify how the document should be formatted. Tags are used in server-side markup languages and interpreted by a web browser for presenting data.

ColdFusion Markup Language (CFML)
A server-side markup language (HTML extensions or tags) that is used to create ColdFusion application pages known as *scripts*.

script
A programming language that is not compiled, but is interpreted and executed at run time.

FIGURE J.1 HOW COLDFUSION WORKS



J-1b The Orderdb Sample Database

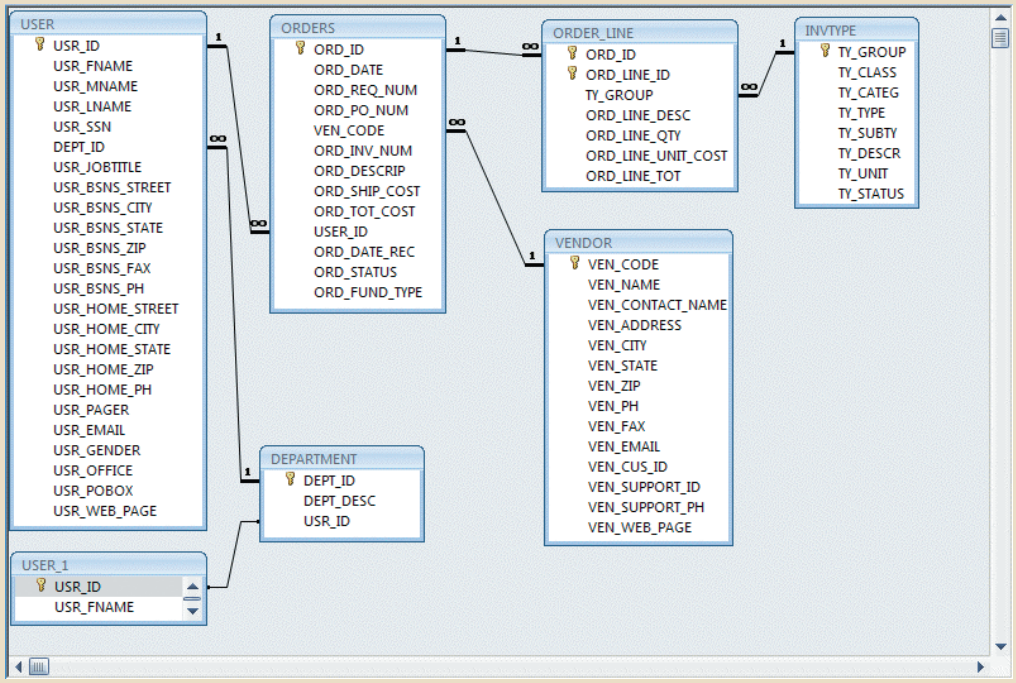
To illustrate how ColdFusion can be used to provide the web-to-database interface, a small Microsoft Access database named Orderdb will be used. The following sections will guide you through the creation of several ColdFusion scripts designed to select, insert, update, and delete data from the Orderdb database.

Note

To focus your efforts on the use of CFML to access databases, these exercises assume that you are familiar with basic HTML tags and the HTML editing process. The examples shown in this appendix can be created using any standard text editor such as Notepad.exe.

The Orderdb database, whose relational diagram is shown in Figure J.2, was designed to track the purchase orders placed by users in a multidepartment company.

FIGURE J.2 THE ORDERDB DATABASE'S RELATIONAL DIAGRAM



Note

The database and script files used in this appendix are located on cengagebrain.com. The database name is **orderdb.mdb.** Note that this database has been saved in MS Access 2000 format. ***Please do not change the database format to a newer version***. Unless you have the latest MS Access ODBC/OLE drivers, changing the format could render the database inaccessible to the ColdFusion scripts. The orderdb.mdb is accessed using the ODBC data source name **"RobCor"**.

As you examine Figure J.2, note that the database contains the following tables: USER, DEPARTMENT, VENDOR, INVTYPE, ORDERS, and ORDER_LINE. The relationships between the tables are derived from the following business rules:

- A department employs many users.
- A department may be managed by one of those users.
- Each user belongs to one department, and each department can have many users.
- Each department may have a department manager. (That is, a department manager is optional.)
- Each order is placed to only one vendor, and each vendor can receive many orders.
- Each order contains one or more order lines.
- Each order line refers to one inventory type.

USER_1 is a virtual component created by MS Access when *multiple relationships* between USER and DEPARTMENT are set. MS Access created the USER_1 virtual table to represent the "USER *manages* DEPARTMENT" relationship. This is a one-to-one *optional* relationship, thus allowing the USR_ID field in the DEPARTMENT table to be null. (This relationship will be used to illustrate how you can manage optional relationships within a web interface.)

J-1c Creating a Simple Query with CFQUERY and CFOUTPUT

Let's begin by creating a simple script to produce a query that will list all of the vendors in the VENDOR table. This script will perform two tasks:

1. Query the database, using standard SQL to retrieve a data set that contains all records found in the VENDOR table.
2. Format all of the records generated in Step 1 in HTML so they are included in the page that is returned to the client browser.

   Script J.1 contains the required code.

Note

If you install the ColdFusion demo, you can run this script (and all subsequent scripts) by pointing the browser to the web server address. For example, if your computer is the web server, you can use ***http://127.0.0.1/robcor/rc-1.cfm*** as your web address. If your web server is a different computer selected by your instructor, use the address supplied by your instructor. You can also access a menu for all ColdFusion scripts by going to ***http://127.0.0.1/robcor***.

As you examine Script J-1 (rc-1.cfm), note that its ColdFusion tags are CFQUERY (to query a database) and CFOUTPUT (to display the data returned by the query). Note that the CFML and HTML tags are shown in different colors. Let's take a closer look at these two CFML tags.

- <*CFQUERY*> tag (lines 4–6). This tag sets the stage for the database connection and the execution of the enclosed SQL statement. You should include all query statements *before* or *within* the document's HTML header (<HEAD>) section. Using that procedure, the page will display the output on the client side *after all queries have been executed*. If you do not use that procedure, the browser will be perceived as "slow" because the page will start to display output and then pause to wait for additional data to arrive from the Web server. The CFQUERY tag requires the following parameters:
  - *NAME* = "*queryname*". This is a mandatory parameter, whose name uniquely identifies the record set returned by the database query—in this case, *venlist*. You can have multiple queries, each with a unique name, in a single script.
  - *DATASOURCE* = "*datasourcename*". This, too, is a mandatory parameter that uses the database name as defined in ODBC. Keep in mind that the database name is case sensitive. Therefore, if the database name is "*RobCor*", do not use "ROBCOR" or "robcor". You use the ColdFusion Administrator interface to define all data sources. Data sources can use ODBC, a native driver such as Oracle SQL*Net, or Microsoft OLE-DB (object linking and embedding).

SCRIPT J.1 A SIMPLE QUERY USING CFQUERY AND CFOUTPUT (RC-1.CFM)

```
1  <HTML>
2  <HEAD>
3  <TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
4  <CFQUERY NAME="venlist" DATASOURCE="RobCor">
5      SELECT * FROM VENDOR ORDER BY VEN_CODE
6  </CFQUERY>
7  </HEAD>
8  <BODY BGCOLOR="LIGHTBLUE">
9  <H1>
10 <CENTER><B>Simple Query using CFQUERY and CFOUTPUT</B></CENTER>
11 <CENTER><B>(Vertical Output)</B></CENTER>
12 </H1>
13 <BR>
14 <HR>
15 <CFOUTPUT>
16     Your query returned #venlist.RecordCount# records
17 </CFOUTPUT>
18 <CFOUTPUT QUERY="venlist">
19 <PRE><B>
20  VENDOR CODE:       #VEN_CODE#
21  VENDOR NAME:       #VEN_NAME#
22  CONTACT PERSON:    #VEN_CONTACT_NAME#
23  ADDRESS:           #VEN_ADDRESS#
24  CITY:              #VEN_CITY#
25  STATE:             #VEN_STATE#
26  ZIP:               #VEN_ZIP#
27  PHONE:             #VEN_PH#
28  FAX:               #VEN_FAX#
29  E-MAIL:            #VEN_EMAIL#
30  CUSTOMER ID:       #VEN_CUS_ID#
31  SUPPORT ID:        #VEN_SUPPORT_ID#
32  SUPPORT PHONE:     #VEN_SUPPORT_PH#
33  VENDOR WEB PAGE:   #VEN_WEB_PAGE#
34 <HR></B></PRE>
35 </CFOUTPUT>
36 <FORM ACTION="rc-0.cfm" METHOD="post">
37     <INPUT TYPE="submit" VALUE="Main Menu ">
38 </FORM>
39 </BODY>
40 </HTML>
```

- *SQL statement* (line 5) is another mandatory parameter. In this case, the parameter is defined by the following query, but you could use any ODBC SQL-compliant statement:

  SELECT * FROM VENDOR ORDER BY VEN_CODE

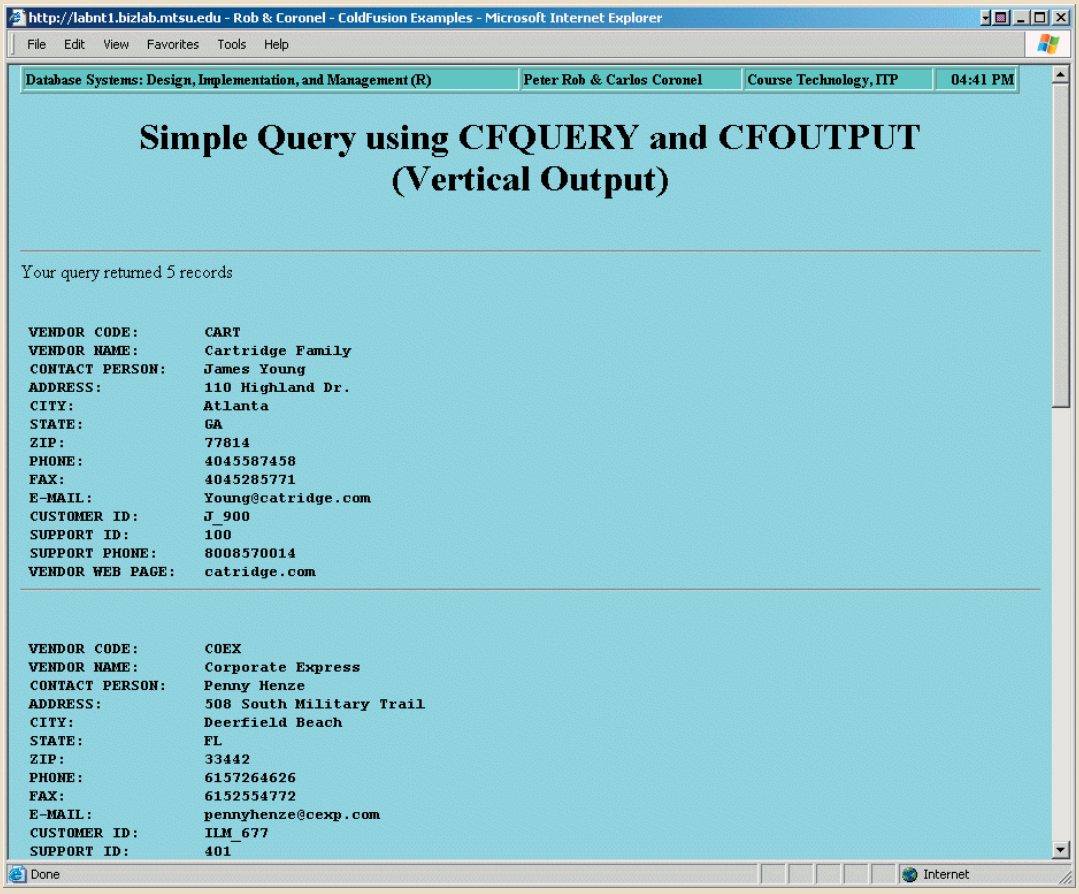
- <*CFOUTPUT*> tag (lines 15–17 and 18–35). This tag is used to display the results from a CFQUERY or to call other ColdFusion variables or functions. Its parameters are:
  - QUERY = "*queryname*". This is an optional parameter (see line 18). If you use a query name for a query that returns 10 rows, this tag will execute all commands between the opening and closing CFOUTPUT tags 10 times—one per row. In short, this tag works like a loop that is executed as many times as the number of rows in the named query set.
  - You can include any valid HTML tags or text within the opening and closing CFOUTPUT tags. ColdFusion uses pound signs (#) to reference query fields in the resulting query set or to call other ColdFusion functions or variables. When ColdFusion encounters a name enclosed within pound signs, it evaluates this named variable to verify that it is a field name of the named query, an internal variable, or a function. Following this evaluation, ColdFusion will replace the named variable with the value that corresponds to the query, the internal variable, or the function. In this case, line 16 is a call to a ColdFusion internal variable. When the query is

executed, this variable's value is the number of rows in the query output. The name of the query must precede the *RecordCount* keyword, and the two components must be separated by a period. For example, *#venlist.RecordCount#* is used to name the variable in line 16.

– Lines 19–34. These lines are repeated as a loop, one for each record returned in the named query. In this example, the loop is defined by *QUERY* = "*venlist*". Note that in lines 20–33, the field names (enclosed in pound signs) will be replaced by the actual values of the fields that are returned by the query.

The output produced by rc-1.cfm is shown in Figure J.3.

FIGURE J.3 THE RC-1.CFM SCRIPT OUTPUT (VERTICAL LISTING)



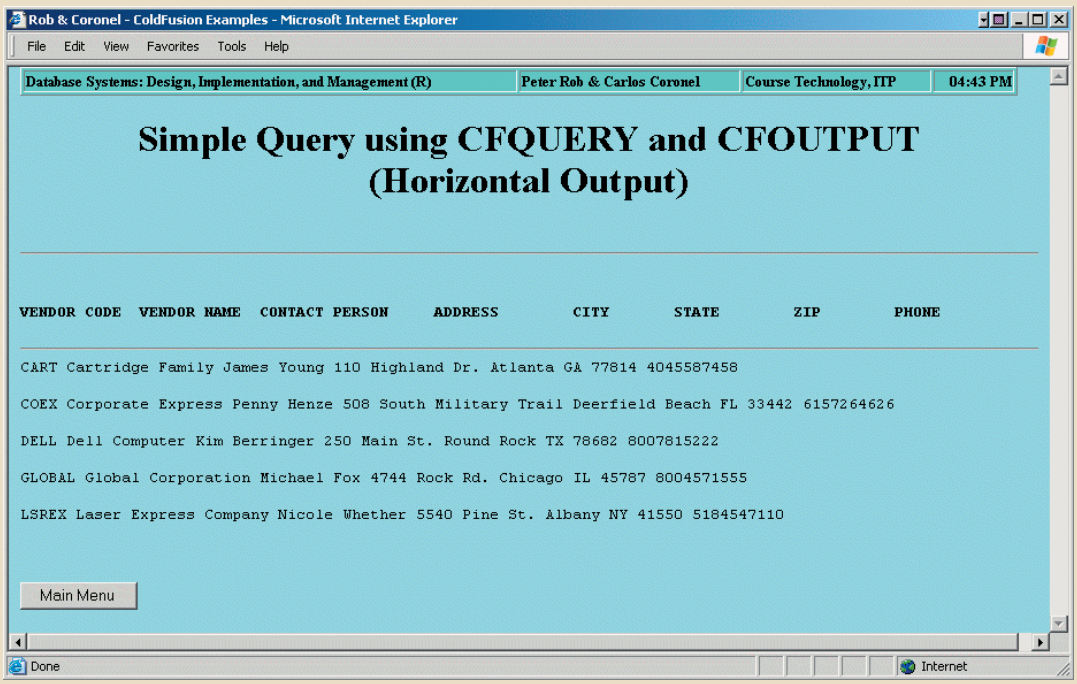
A variation of the just-described approach is found in Script J.2 (rc-2.cfm), in which the output is preformatted (using the HTML <PRE> tag) one row per record, using the CFOUTPUT tag.

SCRIPT J.2 CFQUERY WITH TABULAR CFOUTPUT (RC-2.CFM)

```
<HTML>
<CFQUERY NAME="VENDATA" DATASOURCE="RobCor">
        SELECT * FROM VENDOR ORDER BY VEN_CODE
</CFQUERY>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<B><CENTER>Simple Query using CFQUERY and CFOUTPUT</CENTER></B>
<B><CENTER>(Horizontal Output)</CENTER></B>
</H1>
<BR>
<HR>
<PRE>
<B>
VENDOR CODE  VENDOR NAME  CONTACT PERSON     ADDRESS        CITY       STATE        ZIP        PHONE
</B>
<HR>
<CFOUTPUT QUERY="VENDATA">
#VEN_CODE#    #VEN_NAME#    #VEN_CONTACT_NAME# #VEN_ADDRESS#  #VEN_CITY# #VEN_STATE#  #VEN_ZIP#  #VEN_PH#<BR>
</CFOUTPUT>
</PRE>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu ">
</FORM>
</BODY>
</HTML>
```

The output of script rc-2.cfm is shown in Figure J.4.

FIGURE J.4 THE RC-2.CFM SCRIPT OUTPUT (HORIZONTAL LISTING)



J-1d Creating a Simple Query with CFQUERY and CFTABLE

As you can see in Figure J.4, the output produced by CFOUTPUT is not aligned. To give the output a more polished look, the vendor list can be displayed in tabular format, based on the CFTABLE tag. That tag will automatically create a tabular output in which each row in the data set is placed in a row in the table. The source code for this example is stored in Script J.3 (rc-3.cfm).

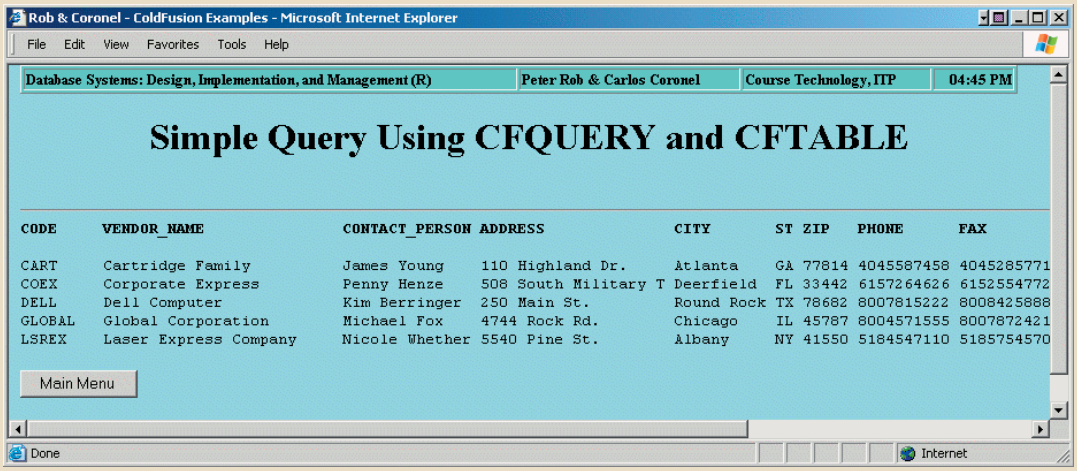
SCRIPT J.3 CFQUERY WITH CFTABLE (RC-3.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
<CFQUERY NAME="VENDATA" DATASOURCE="RobCor">
        SELECT * FROM VENDOR ORDER BY VEN_CODE
</CFQUERY>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<B><CENTER>Simple Query Using CFQUERY and CFTABLE</CENTER></B>
</H1>
<BR>
<TABLE BGCOLOR="Silver" BORDERCOLOR="Fuchsia" FRAME="BORDER">
<TR><HR></TR>
<TR>
<CFTABLE QUERY="VENDATA" STARTROW="1" COLSPACING="1" COLHEADERS>
<CFCOL HEADER="<B>CODE</B>" WIDTH="8" TEXT="#VEN_CODE#">
<CFCOL HEADER="<B>VENDOR_NAME</B>" WIDTH="25" TEXT="#VEN_NAME#">
<CFCOL HEADER="<B>CONTACT_PERSON</B>" WIDTH="14" TEXT="#VEN_CONTACT_NAME#">
<CFCOL HEADER="<B>ADDRESS</B>" WIDTH="20" TEXT="#VEN_ADDRESS#">
<CFCOL HEADER="<B>CITY</B>" WIDTH="10" TEXT="#VEN_CITY#">
<CFCOL HEADER="<B>STATE</B>" WIDTH="2" TEXT="#VEN_STATE#">
<CFCOL HEADER="<B>ZIP</B>" WIDTH="5" TEXT="#VEN_ZIP#">
<CFCOL HEADER="<B>PHONE</B>" WIDTH="10" TEXT="#VEN_PH#">
<CFCOL HEADER="<B>FAX</B>" WIDTH="10" TEXT="#VEN_FAX#">
<CFCOL HEADER="<B>E-MAIL</B>" WIDTH="10" TEXT="#VEN_EMAIL#">
<CFCOL HEADER="<B>CUSTOMER_ID</B>" WIDTH="8" TEXT="#VEN_CUS_ID#">
<CFCOL HEADER="<B>SUPPORT_PHONE</B>" WIDTH="6" TEXT="#VEN_SUPPORT_ID#">
<CFCOL HEADER="<B>WEB PAGE</B>" WIDTH="14" TEXT="#VEN_WEB_PAGE#">
</CFTABLE>
</TR>
<TR>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu ">
</FORM>
</TR>
</TABLE>
</BODY>
</HTML>
```

The output of script rc-3.cfm is shown in Figure J.5.
The rc-3.cfm script's CFTABLE ColdFusion tag contents are as follows:

- <*CFTABLE*> tag (line 16). This tag, used to display the results from a CFQUERY (lines 4–6) in a tabular format, requires the following parameters:
  - *QUERY* = "*queryname*". A required parameter that uses the name of the query that generated the data set to be displayed in tabular format.
  - *STARTROW* = "*1*". An optional parameter that is used to tell ColdFusion which query row will be the table's first row. This parameter is particularly useful when your query returns many rows and you do not want to display them all in one long page. Instead, you can display them in multiple pages, each page displaying the number of rows defined by the parameter. For example, if you want to list 10 rows per page, the starting row will be 1 for the first page, 11 for the second page, 21 for the third, and so on.
  - *COLSPACING* = "*1*". An optional parameter that is used to define the number of spaces to be placed between columns.
  - *COLHEADERS*. An optional parameter that will make the first row in the table a row header. This row header contains the name of each column, using the values defined in the CFCOL tags.
- The CFCOL tag (lines 17–29) is used to define each table column, using the following parameters:
  - *HEADER* = "*header text*". This is the header text that will appear in the table's header row for each of the displayed columns. The header text can include HTML tags.
  - *WIDTH* = "*x*". This parameter defines the column width.
  - *TEXT* = "*#queryfieldname#*". This is the actual value to be placed in the selected column. For example, line 17 will cause ColdFusion to replace *#VEN_CODE#* with the actual values retrieved by the query for this field.

FIGURE J.5 THE RC-3.CFM SCRIPT OUTPUT (FORMATTED HORIZONTAL LISTING)



J-1e Creating a Dynamic Search Page

At this point, you have seen how the CFQUERY, CFOUTPUT, and CFTABLE tags are used to send the SQL statement to the database to retrieve a data set. Given the script files used thus far, the query will always retrieve the same records from the VENDOR table. (Because the SQL statement is "hard-coded" inside the page, it cannot be changed unless the SQL code is manually edited each time it is to be used to generate a different query output.) Such a static output display clearly limits the query's usefulness.

To create a practical dynamic query environment, you can create a dynamic search query in which the query search condition can be changed at the user's option—without requiring script page editing. To demonstrate the process, two fields, vendor code and vendor state, will be used to search for user-specified vendor records. (Naturally, you can create a search form that uses as many fields as you think are necessary.)

To perform a dynamic query over the web, you must complete these two steps:

1. Create a script that will generate a form that will be used to enter the criteria used in the search. In other words, the form allows the user to enter the parameter values that are to be used in the query statement.
2. Create a script that will execute the query and display the results based on the parameters that are passed to it by the script created in Step 1.

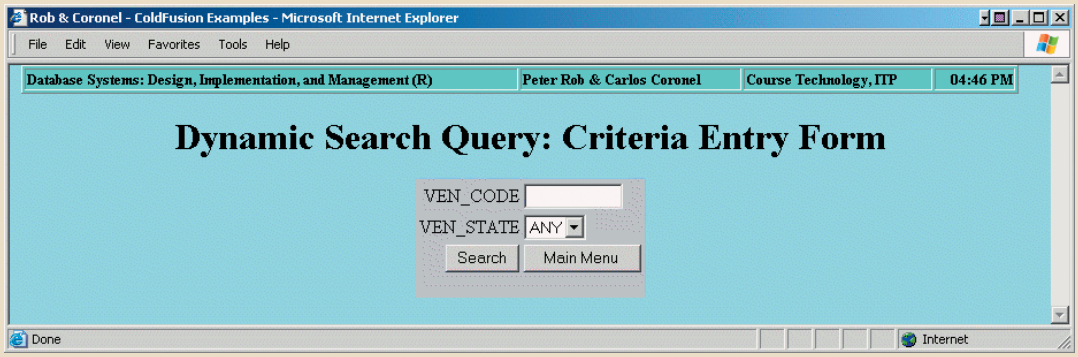
   The script required to complete Step 1 is listed in Script J.4A (rc-4a.cfm).

SCRIPT J.4A DYNAMIC SEARCH QUERY: CRITERIA-ENTRY FORM (RC-4A.CFM)

```
<HTML>
<CFQUERY NAME="STATELIST" DATASOURCE="RobCor">
    Select VEN_STATE from VENDOR Order by VEN_STATE
</CFQUERY>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Dynamic Search Query: Criteria Entry Form</B></CENTER>
</H1>
<FORM ACTION="rc-4b.cfm" METHOD=POST>
<TABLE ALIGN="CENTER" BGCOLOR="Silver">
    <TR>
    <TD ALIGN="right">VEN_CODE</TD>
    <TD>
    <INPUT TYPE ="text" NAME="VEN_CODE" SIZE="10" MAXLENGTH="10"></TD>
    </TR>
    <TR>
    <TD ALIGN="right">VEN_STATE</TD>
    <TD><SELECT NAME="VEN_STATE" SIZE=1>
        <OPTION SELECTED VALUE="ANY">ANY
         <CFOUTPUT QUERY="STATELIST">
         <OPTION VALUE="#STATELIST.VEN_STATE#">#VEN_STATE#
         </CFOUTPUT>
         </SELECT>
    </TD>
    </TR>
    <TR>
    <TD ALIGN="right" VALIGN="middle">
        <INPUT TYPE="submit" VALUE="Search">
        </FORM>
    </TD>
    <TD ALIGN="right" VALIGN="middle">
        <FORM ACTION="rc-0.cfm" METHOD="post">
            <INPUT TYPE="submit" VALUE="Main Menu ">
        </FORM>
    </TD>
    </TR>
</TABLE>
</BODY>
</HTML>
```

The rc-4a.cfm script output is shown in Figure J.6.

FIGURE J.6 THE RC-4A.CFM SCRIPT OUTPUT (STATE SEARCH CRITERIA ENTRY FORM)



As you examine Figure J.6, note that the rc-4a.cfm script generates a form in which the user enters the search parameters. Let's follow the rc-4a.cfm script to see how it works.

- In lines 2–4, all existing values are retrieved from the *VEN_STATE* field in the VENDOR table. This query effectively tells you what states are represented in the vendor table. If no vendor is listed for a given state, that state will not be shown in the returned data set. This query is named "*STATELIST*", and it will be used later in your form.
- In lines 12–32, the form is defined. Note that when the user clicks the "submit" button, the rc-4b.cfm script (shown later) will be called to receive the form's variables, *VEN_CODE* and *VEN_STATE*.
- Line 17 presents the first input text box to let the user enter a value for the *VEN_CODE* form variable. This value will be used in the SQL statement to search for all records with matching VEN_CODE values.
- Lines 21–26 create a drop-down SELECT box to let the user pick the state to be used in the vendor table query. This selection will later be passed to the rc-4b.cfm script.
- In lines 23–25, the CFOUTPUT tag is used to build the selection options, using the states that occur in the VENDOR table. The default ("SELECTED") option shown in line 22 gives the user the ability to search without selecting any particular state. If line 22 is not included, there will be no way to limit the search to only a vendor code, nor will the user be able to display all vendors from all states. To limit the search, the VEN_STATE form field must have the option to contain a null "value." In short, line 22's default condition provides crucial flexibility.

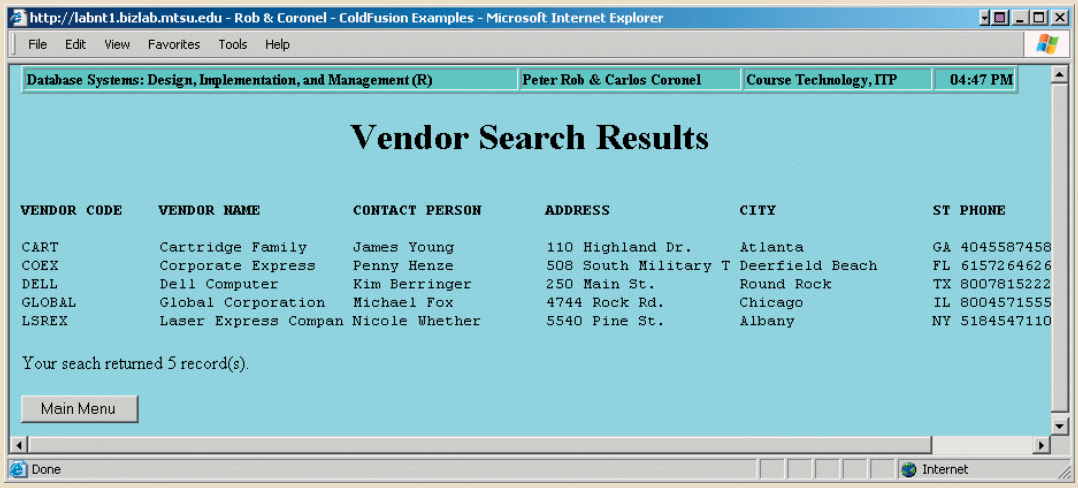
When the user clicks the Search button, the rc-4b.cfm script is executed, and the two form variables (VEN_CODE and VEN_STATE) are passed to the rc-4b.cfm script. The rc-4b.cfm script is shown in Script J.4B.

SCRIPT J.4B THE VENDOR SEARCH RESULTS (RC-4B.CFM)

```
<HTML>
<CFQUERY NAME="SearchVendor" DATASOURCE="RobCor">
    SELECT VEN_CODE, VEN_NAME, VEN_CONTACT_NAME, VEN_ADDRESS, VEN_CITY, VEN_STATE, VEN_PH
    FROM VENDOR
    WHERE 0=0
    <CFIF #FORM.VEN_CODE# IS NOT "">
        AND VENDOR.VEN_CODE LIKE '%#FORM.VEN_CODE#%'
    </CFIF>
    <CFIF #FORM.VEN_STATE# IS NOT "ANY">
        AND VENDOR.VEN_STATE LIKE '%#FORM.VEN_STATE#%'
    </CFIF>
    ORDER BY VEN_CODE
</CFQUERY>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Vendor Search Results</B></CENTER>
</H1>
<BR>
<CFTABLE QUERY="SearchVendor" STARTROW="1" COLSPACING="1" COLHEADERS>
<CFCOL HEADER="<B>VENDOR CODE</B>" WIDTH="14" TEXT="#VEN_CODE#">
<CFCOL HEADER="<B>VENDOR NAME</B>" WIDTH="20" TEXT="#VEN_NAME#">
<CFCOL HEADER="<B>CONTACT PERSON</B>" WIDTH="20" TEXT="#VEN_CONTACT_NAME#">
<CFCOL HEADER="<B>ADDRESS</B>" WIDTH="20" TEXT="#VEN_ADDRESS#">
<CFCOL HEADER="<B>CITY</B>" WIDTH="20" TEXT="#VEN_CITY#">
<CFCOL HEADER="<B>STATE</B>" WIDTH="2" TEXT="#VEN_STATE#">
<CFCOL HEADER="<B>PHONE</B>" WIDTH="20" TEXT="#VEN_PH#">
</CFTABLE>
<CFIF #SEARCHVENDOR.RECORDCOUNT# IS 0>
    <H2>No records were found matching your criteria </H2>
<CFELSE>
    <CFOUTPUT>Your seach returned #SearchVendor.RecordCount# record(s).</CFOUTPUT>
</CFIF>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu ">
</FORM>
</BODY>
</HTML>
```

The rc-4b.cfm script output is shown in Figure J.7.

FIGURE J.7 THE RC-4B.CFM SCRIPT OUTPUT (VENDOR SEARCH RESULTS: ALL STATES)

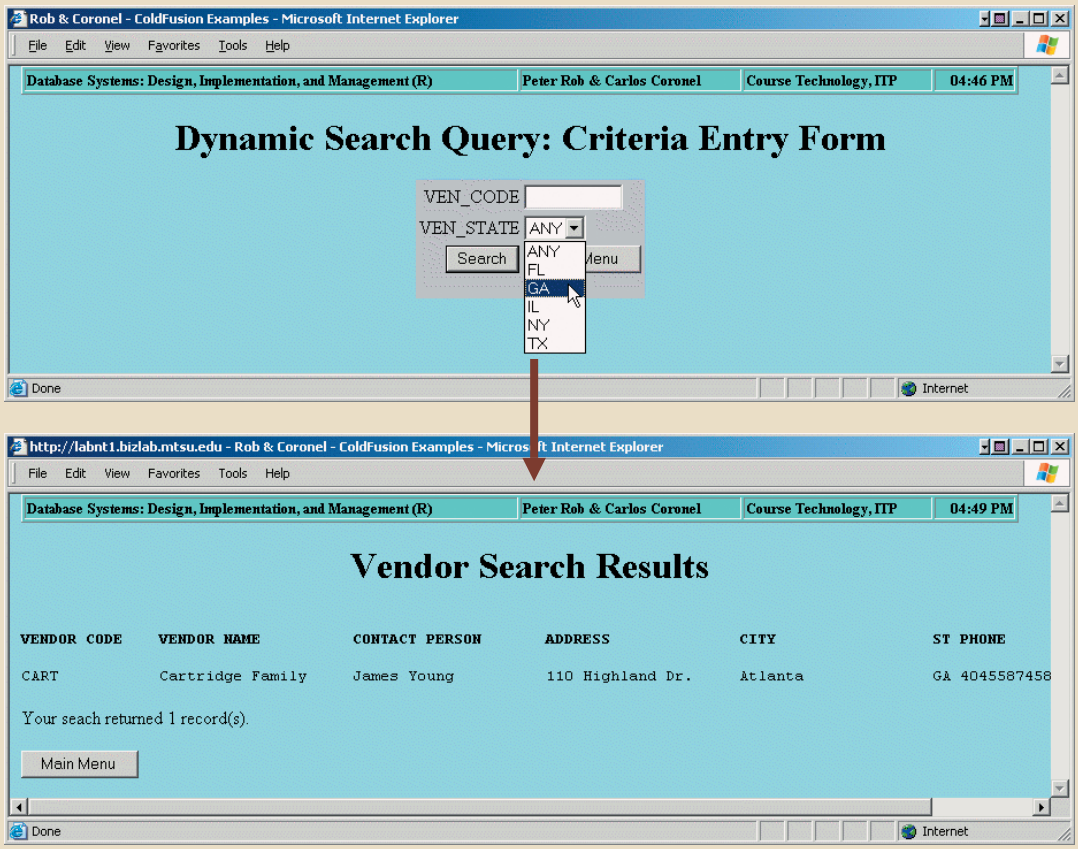


As you examine Script J.4B (rc-4b.cfm,) note how it references the form variables received from the calling script (rc-4a.cfm) and how it uses the CFIF tag to dynamically build SQL statements. Let's take a closer look at this script.

- Lines 2–13 are used to create the SQL statement and to query the database.
- Line 5 creates a dummy WHERE clause that will be used to anchor the query's conditional criteria. The "0=0" conditional criterion is the default condition that will list all records. This line is required to form the basis for additional conditions, using SQL's logical operators. Naturally, if no additional conditions are specified, the default value ensures that the query will list all records.
- Lines 6–8 use a CFIF tag to evaluate the VEN_CODE form field passed from the calling page. (In this case, the rc-4a.cfm script created the calling page.) If the field specified in line 6 is not null (""), the condition specified in line 7 is added to the query.
- Lines 9–11 use the CFIF tag to evaluate the VEN_STATE form field passed from the calling page (rc-4a.cfm). If this field is not "ANY", line 10 adds the "state" condition to the query. (Remember that "ANY" is the default selection for the rc-4a.cfm script's VEN_STATE field.) This default selection ensures that it is not necessary to search for any specific state. Therefore, the user can select a state to limit the query output to that state, or by not selecting a state, the user can generate the query output for all states.
- Lines 22–30 indicate that the CFTABLE and CFCOL tags are used to display the query result set in a tabular format.
- Finally, a CFIF statement is used to evaluate the number of records in the resulting record set and to display the appropriate message. For example, line 31 ensures that the "*No records were found matching your search criteria*" message is displayed in line 32 if the record set returns zero (0) rows. If the record count is not zero, line 34 generates a message that indicates the number of records returned in the query result set.

Figure J.8 shows the output for a dynamic search based on Vendors for the condition VEN_STATE = "GA".

FIGURE J.8 THE VENDOR LIST FOR THE CONDITION VEN_STATE = "GA"



The dynamic search process clearly makes the web a viable end-user information-generation tool. However, to conduct *transactions*, you must be able to modify the database's table contents. (For example, if you make a withdrawal from inventory, you must be able to update the inventory table; if you make a purchase, you must be able to generate an invoice record; and so on.) Therefore, the focus now will be on the basic procedures that may be used to insert, update, and delete data through web interfaces. Before you can develop web-based transaction applications, you need to know why and how the shortcomings of HTML and browsers affect data manipulation activities. Those shortcomings are a function of the web's basic structure, which is often described as a so-called *stateless system*.

stateless system
A system in which a web server does not know the status of the clients communicating with it. The web does not reserve memory to maintain an open communi-cations state between the client and the server.

J-1f The Web as a Stateless System

The web is said to be a stateless system. Simply put, the label **stateless system** indicates that at any given time, a web server does not know the status of any of the clients communicating with it. That is, there is no open communication line between the server and each client accessing it, which, of course, is impractical in a *worldwide* web! Instead, client and server computers interact in very short "conversations" that follow the request-reply model. For example, the browser is concerned only with the *current* page, so there is no way for the second page to know what was done in the first page. The only time the

client and server computers communicate is when the client requests a page—when the user clicks a link—and the server sends the requested page to the client. Once the client receives the page and its components, the client/server communication is ended. Therefore, although you may be browsing a page and *think* that the communication is open, you are actually just browsing the HTML document stored in the local cache (temporary directory) of the client browser. The server does not have any idea what the end user is doing with the document, what data is entered in a form, what option is selected, and so on. On the web, if you want to act on a client's selection, you need to jump to a new page (go back to the web server), thus losing track of whatever was done before!

Not knowing what was done before (or what a client selected before getting to this page) makes adding business logic to the web cumbersome. For example, suppose that you need to write a program that performs the following steps: display a data-entry screen, capture data, validate data, and save data. That entire sequence can be completed in a single COBOL program because COBOL uses a working storage section that holds in memory all variables used in the program. Now imagine the same COBOL program—but *each* section (PERFORM statement) is now a separate program! That is precisely how the web works. In short, the web's stateless nature means that extensive processing required by a program's execution cannot be done directly through a single webpage; the client browser's processing ability is limited by the lack of processing ability and the lack of a working storage area to hold variables used by all pages on a website.

Keep in mind that a web browser's function is to display a page on the client computer. The browser—through its use of HTML—does not have computational abilities beyond formatting output text and accepting form field inputs. Even when the browser accepts form field data, there is no way to perform immediate data entry validation. Therefore, to perform such crucial processing in the client, the web defers to other web programming languages such as Java, JavaScript, and VBScript. The browser most resembles a dumb terminal that displays only data and can perform only rudimentary processing such as accepting form data inputs. Most of the processing takes place at the server end—in this case, the web application server.

To circumvent the above-mentioned shortcomings of the web environment, most web application servers have session management capabilities that allow a web server to maintain status information for each client session in the server's memory. Each web server vendor has its own way to maintain that information. In the case of ColdFusion, session variables are declared using the <CFSET session.variablename=value> command syntax. For example, to declare a user-type session variable, you would say, <CFSET session.usertype = "ADMIN">. Then the variable would be available to all pages in the same client session. The client session starts the first time the client's browser connects to the web server; it ends when the client accesses a page outside the website, closes his/her browser, or stays idle for a given time-out period.

J-1g Inserting Data

In this section, you will create a data entry form to insert data in the DEPARTMENT table. In the following example, the DEPARTMENT table contains three fields: department ID and department description, which are required, and an optional manager user ID that references the USER table. Of course, *if the user enters a user ID, that ID must match a user ID in the USER table*. Given that basic scenario, let's see how ColdFusion can be used to establish basic server-side data validation for the required fields and how ColdFusion can implement and manage data entry for an optional relationship.

At least two pages are needed to accomplish the just-described tasks. The first page, generated by a script named rc-5a.cfm, creates a form to get the data. The second page, generated by a script named rc-5b.cfm, inserts the data in the table. Data validation will

take place at the server side. Let's first look at script J.5A (rc-5a.cfm). The script is followed by its output in Figure J.9.

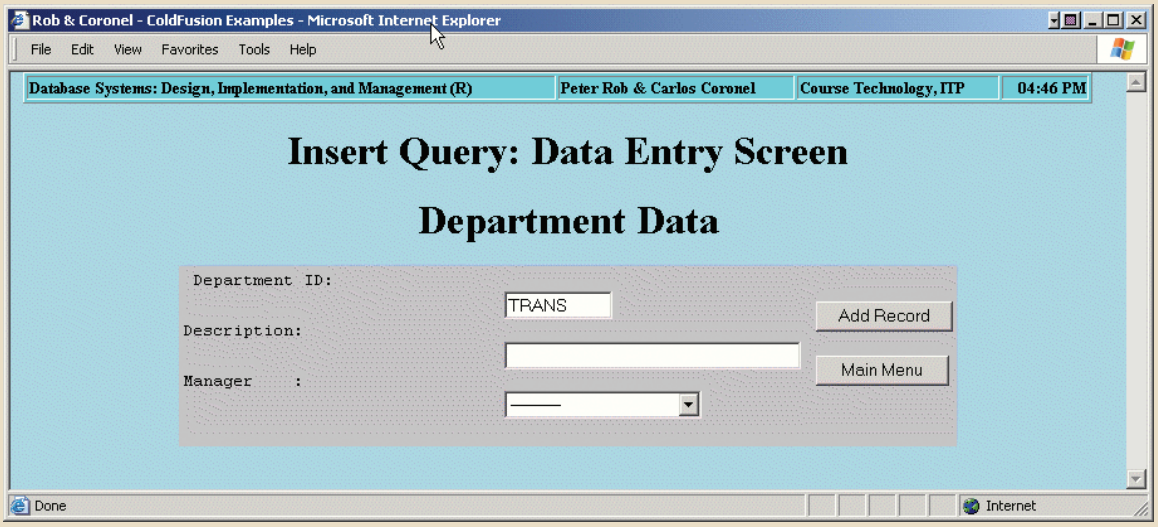
SCRIPT J.5A INSERT QUERY—DATA ENTRY (RC-5A.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
<CFQUERY NAME="USRLIST" DATASOURCE="RobCor">
SELECT USR_ID, USR_LNAME, USR_FNAME, USR_MNAME
      FROM USER
      WHERE USR_ID NOT IN (SELECT USR_ID FROM DEPARTMENT WHERE USR_ID > 0)
      ORDER BY USR_LNAME, USR_FNAME, USR_MNAME
</CFQUERY>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Insert Query: Data Entry Screen</B></CENTER></H1>
<FORM ACTION="rc-5b.cfm" METHOD="post">
<CENTER><H1>Department Data</H1></CENTER><!--- the following code defines required fields --->
    <INPUT TYPE="hidden" NAME="DEPT_ID_required" VALUE="You must enter a DEPT_ID">
    <INPUT TYPE="hidden" NAME="DEPT_DESC_required" VALUE="You must enter a description">
    <TABLE ALIGN="CENTER" BGCOLOR="Silver">
        <TR>
            <TD>
            <PRE>
 Department ID:
                <INPUT TYPE="text" NAME="DEPT_ID" SIZE="10" MAXLENGTH="10"><BR>Description:
                <INPUT TYPE="text" NAME="DEPT_DESC" SIZE="35" MAXLENGTH="35"><BR>Manager    :
                <SELECT NAME="USR_ID" SIZE="1"><!--- select user from list --->
                    <OPTION VALUE="">----------
                    <CFOUTPUT QUERY="USRLIST">
                        <OPTION VALUE="#UsrList.USR_ID#">[#USR_ID#] #USR_LNAME#, #USR_FNAME#, #USR_MNAME#
                    </CFOUTPUT>
                </SELECT></PRE>
            </TD>
            <TD>
                <INPUT TYPE="submit" VALUE="Add Record">
</FORM>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE=" Main Menu ">
</FORM>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

The rc-5a.cfm script produces a data entry form to enable the end user to enter the new department data. To show you how the form works, let's add a new Transportation department (TRANS) and assign a manager to run the department. The rc-5a.cfm script is designed to perform a data validation check in the USR_ID field, using a query and a select box. To see how the script accomplishes those tasks, let's look at some key lines, as follows:

- Lines 4–9 execute a nested query to find all user IDs for employees who are not already department managers. Performing this portion of the data validation procedure ensures that there are no duplicate user IDs in the DEPARTMENT table. Therefore, it will not be possible to have a manager manage more than one department. Also, because a department might not yet have a manager assigned to it, the USR_ID might not have a value in it. To perform the requisite checks, start with a nested query, using the *USR_ID > 0* condition. This condition will be true only for those records in which a manager (USR_ID) exists.

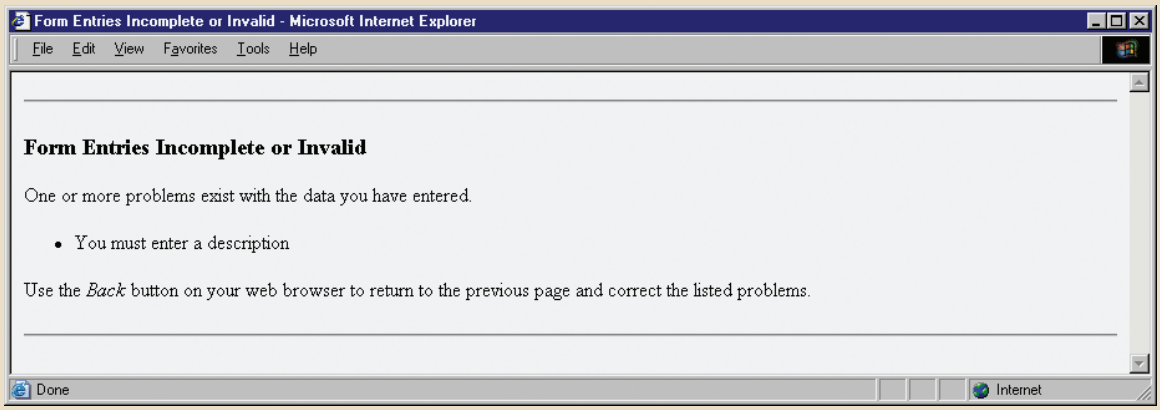
FIGURE J.9 INSERT QUERY—DATA ENTRY SCREEN (RC-5A.CFM)



- Lines 14–34 define the form that will be used to enter the data. Note that the rc-5b.cfm script will be called when the user clicks the Add Record button.
- Lines 16 and 17 are special form tags ColdFusion uses to perform data validation at the server side. In this case, the entries will be validated for the two required fields, the department identification code (DEPT_ID) and the department description (DEPT_DESC). This task is performed by using an INPUT form tag with the following parameters:
  - *TYPE* = "*hidden*". This parameter ensures that the field will not be displayed on the screen.
  - *NAME* = "*fieldname_required*". This parameter specifies the field to be checked, followed by the word *_required*. Other parameter options are:
    - ***_integer*** to check for integer values only.
    - ***_date*** to check for valid dates only in the format mm/dd/yy.
    - ***_range*** to check for a value within a range. The range is given in the value = parameter; for example, value = "MIN=10 MAX=20".
  - *VALUE* = "*error message*". This parameter contains the error message to be displayed when the constraint is violated (in this case, when the field is empty). When the parameter type = _range, this field contains the maximum and minimum values used in the validation.
- Lines 25–30 create a drop-down select box to show all of the *available* users who can be selected as department manager. Note in particular the following lines:
  - Line 26 defines a null option, represented on the screen by a dotted line to indicate that the department does not have a manager assigned. This line implements the optionality of the USR_ID field. If this line is not included, there will be no way to assign a null to the USR_ID field.
  - Lines 27–29 generate a list of all users who are eligible to manage the new department. Remember that the USRLIST query returns only the USR_ID of users who are not already in the DEPARTMENT table. In other words, the USRLIST query lists only those users *who are still available to become department managers*. (Remember that the business rule specifies that a manager can manage only one department.)

When the user clicks the Add Record button, the form is sent to the web server for processing. There, ColdFusion will evaluate the required fields, sending an error message if one of the required fields is empty. (See Figure J.10.)

FIGURE J.10 SERVER-SIDE VALIDATION ERROR MESSAGE



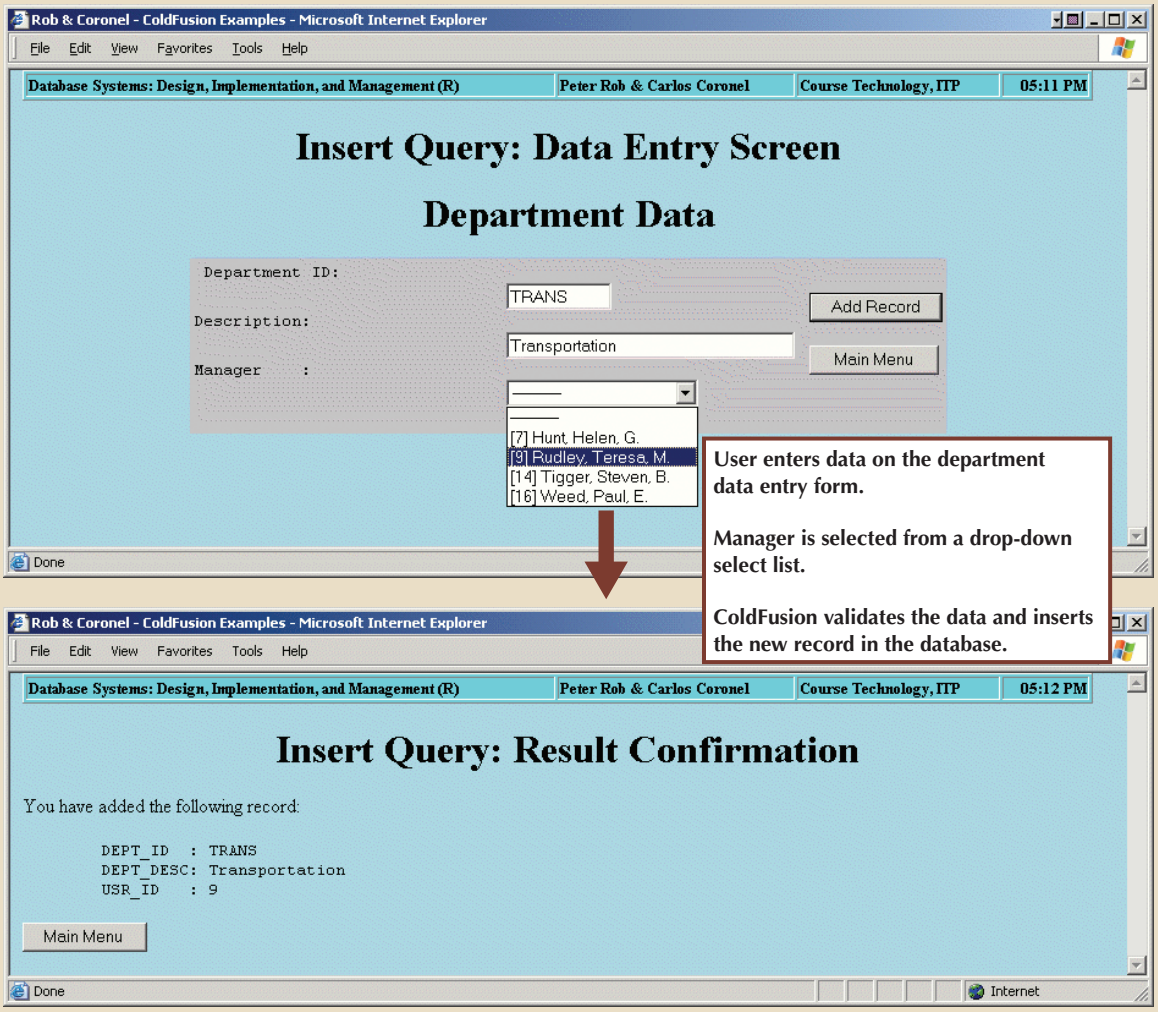
If the server-side data validation yields the conclusion that the data entry is valid, the second script, rc-5b.cfm, is executed. This script receives the form fields from the rc-5a.cfm script and uses the CFINSERT tag to add the record to the database. Once the record has been added, the rc-5b.cfm script presents a confirmation screen to the end user.

Script rc-5b.cfm (Script J.5B) is shown next. The execution of the rc-5b.cfm script is shown in Figure J.11.

SCRIPT J.5B INSERT QUERY—RESULT CONRMATION (RC-5B.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel - ColdFusion Examples</TITLE>
<!--- inserting record in table --->
<CFINSERT DATASOURCE="RobCor" TABLENAME="DEPARTMENT">
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Insert Query: Result Confirmation</B></CENTER></H1>
You have added the following record:
<CFOUTPUT>
<PRE>
    DEPT_ID  : #DEPT_ID#
    DEPT_DESC: #DEPT_DESC#
    USR_ID   : #USR_ID#
</PRE>
</CFOUTPUT>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu ">
</FORM>
</BODY>
</HTML>
```

FIGURE J.11 INSERT QUERY—RESULT CONRMATION SCREEN



To see how the rc-5b.cfm script uses the CFINSERT tag to add the record to the database, check line 5. Note that this tag uses the following two parameters:

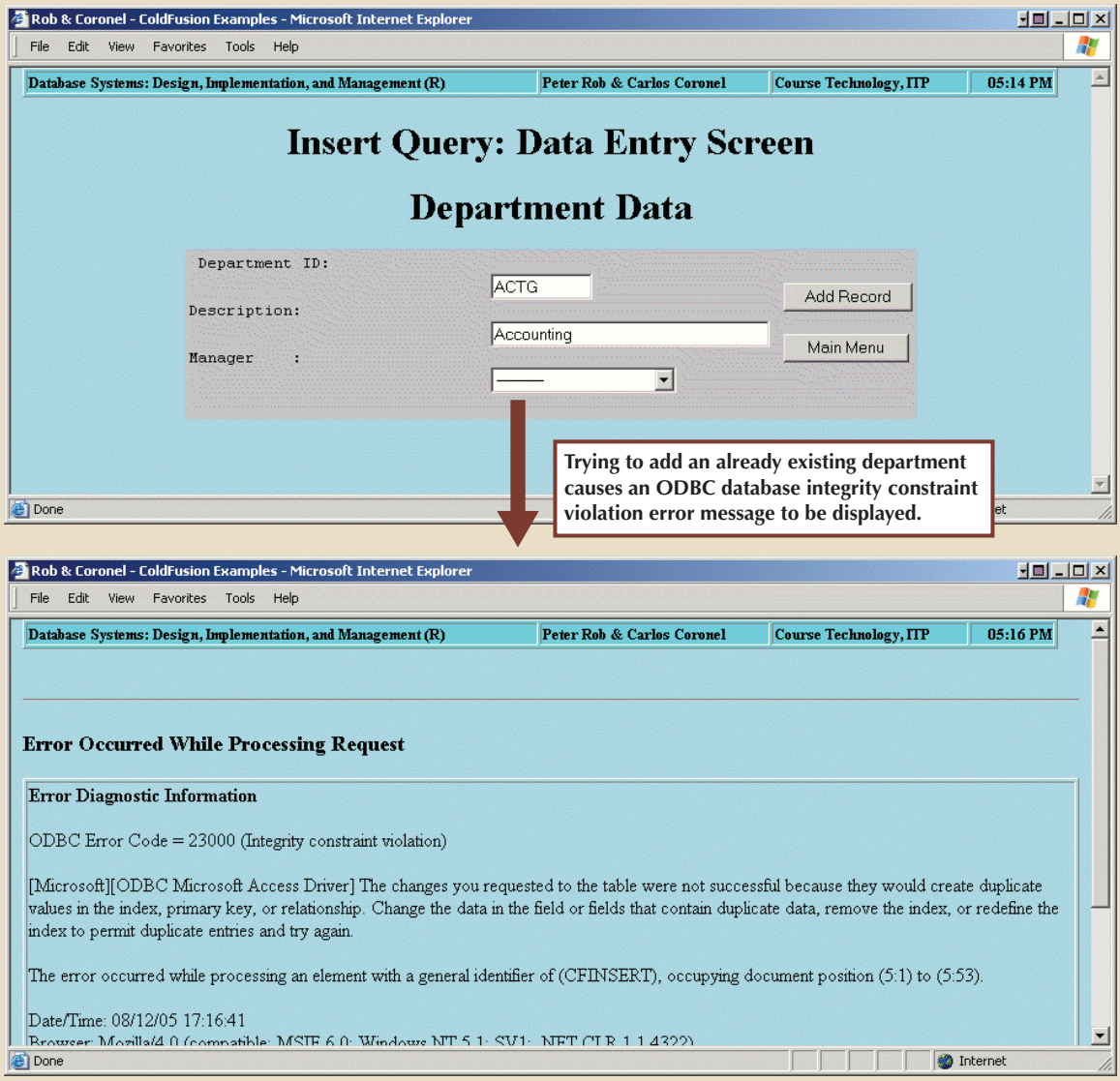
- *DATASOURCE* = "*datasource_name*" (the name of the ODBC database connection).
- *TABLENAME* = "*table_name*" (the name of the table to be updated).

How does the CFINSERT tag know what fields to insert in the table? And where does it get the values to insert into the table columns? The answer to both questions is found by examining the rc-5b.cfm script CFINSERT tag. That tag uses the field names that were defined on the form generated by the rc-5a.cfm script. Recall that the rc-5a.cfm script produced a form containing the DEPT_ID, DEPT_DESC, and USR_ID fields. The CFINSERT tag compares those form field names with the names of the table columns in order to do the insert. To avoid an error condition, the form that was created in the calling page must have form field names that match the table column names.

The MS Access database enforces entity integrity for the Orderdb database's DEPARTMENT table. Therefore, entering an *existing* department ID on the input form automatically triggers an ODBC database integrity violation error, as shown in Figure J.12.

(Remember that entity integrity is violated when a primary key value is duplicated; your error message may look slightly different depending on your version of ColdFusion.)

FIGURE J.12 ODBC INTEGRITY VIOLATION ERROR



J-1h Updating Data

Data updates require multiple pages. For example, if you want to produce a simple update in the DEPARTMENT table's records, the update process requires *three* different pages.

- The first page, produced by the rc-6a.cfm script, lets the end user select the record to be updated. When the user clicks this page's Edit button, the second page, produced by the rc-6b.cfm script, is called and the first page's search field value is passed to the second page. (To keep the process as simple as possible, the primary key, DEPT_ID, will be used to ensure a unique match. If you use secondary search fields, you may

find more than one record. You would need to use an *additional* page to select one of the multiple records to generate a unique match.)

- The second page (rc-6b.cfm) reads the selected record, then displays a data entry form to let the end user modify the data. When the end user clicks the Update button, this page calls the third script and passes the second page's form fields to the third page.
- The third and last page, generated by the rc-6c.cfm script, updates the data in the database and presents a confirmation message to the end user.

Although this process seems simple enough, several issues must be addressed, as you will see next. Let's begin by taking a close look at Script J.6A (rc-6a.cfm). The rc-6a.cfm script output is shown in Figure J.13.

SCRIPT J.6A UPDATE QUERY—RECORD SELECTION (RC-6A.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
<CFQUERY NAME="Deptlist" DATASOURCE="RobCor">
    SELECT * FROM DEPARTMENT ORDER BY DEPT_ID
</CFQUERY>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Update Query: Record Selection Screen</B></CENTER>
</H1>
<TABLE ALIGN="CENTER" BGCOLOR="Silver">
<TR VALIGN="TOP">
<TD>
<FORM ACTION="rc-6b.cfm" METHOD="post">
    <SELECT NAME="DEPT_ID" SIZE=1>
    <CFOUTPUT QUERY="Deptlist">
        <OPTION VALUE="#DEPT_ID#">[#DEPT_ID#] - #DEPT_DESC#
    </CFOUTPUT>
    </SELECT>
    <INPUT TYPE=HIDDEN NAME="DEPT_ID_required" VALUE="DEPT_ID is required">
</TD>
<TD>
<INPUT TYPE="submit" VALUE=" Edit ">
</FORM>
</TD>
<TD>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu ">
</FORM>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

FIGURE J.13 UPDATE QUERY—RECORD SELECTION SCREEN

The rc-6a.cfm script produces the form that lets the end user select the record to be updated. This record selection process requires the completion of the following steps:

- Lines 4–6 execute a query ("Deptlist") to retrieve all DEPARTMENT table records. This record set will be used later to create a drop-down selection box.
- Lines 17–19 use a CFOUTPUT tag to produce a list of available options. In this example, ColdFusion uses an OPTION tag for each department in the "Deptlist" query.
- Lines 15–25 produce the record selection form. This form uses an HTML form tag to pick the department to be updated.
- Line 21 defines the "DEPT_ID" form field to be a required field. This definition ensures that the end user will not generate "variable not defined" errors when the next page is called.

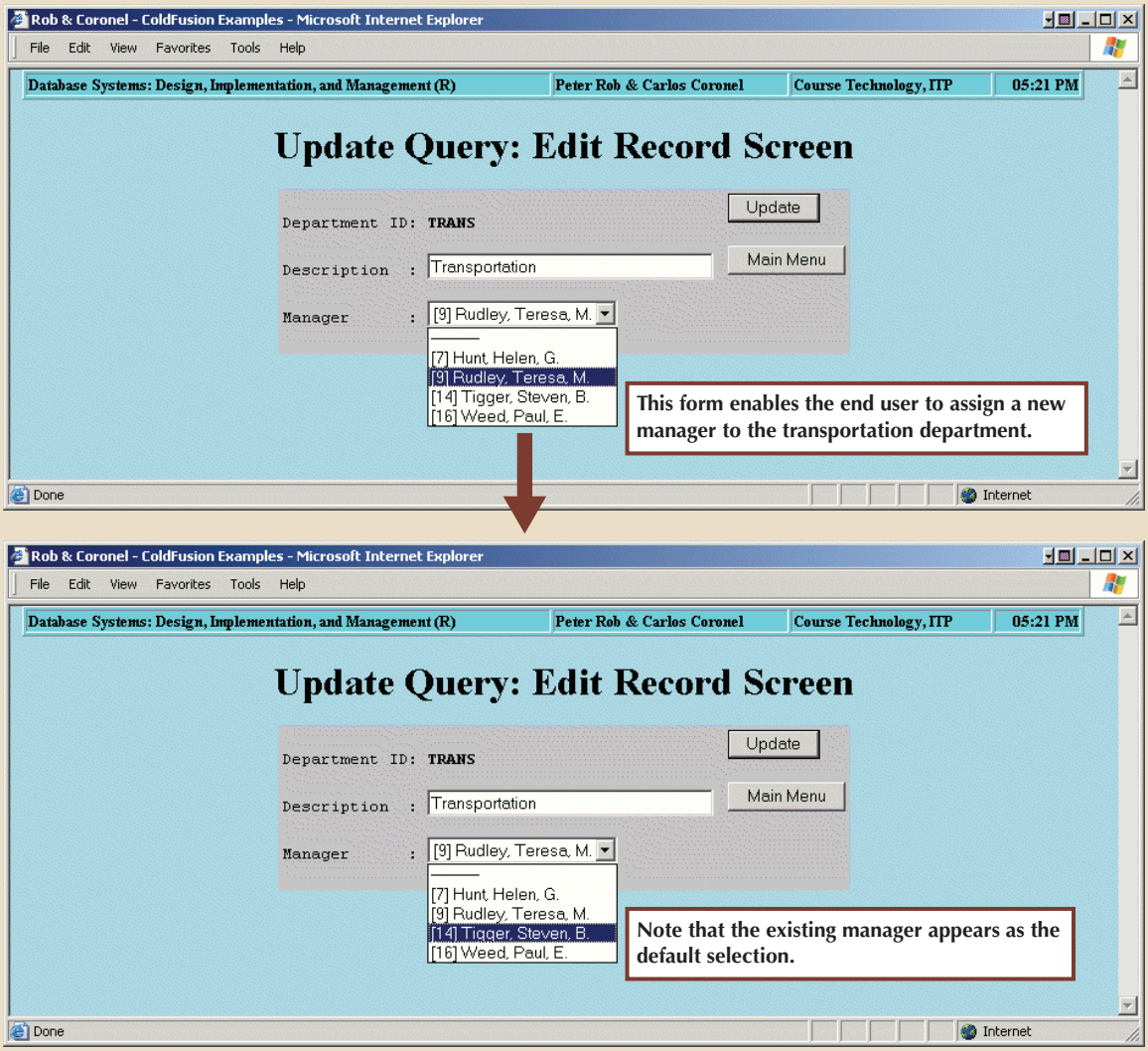
When the end user clicks the form's Edit button, the rc-6b.cfm script is called. The rc-6b.cfm script will receive the DEPT_ID form field as a parameter, using it to find the matching department table record. It will then prepare and display the data edit form.

The script rc-6b.cfm is shown next in Script J.6B. The rc-6b.cfm script output is shown in Figure J.14.

SCRIPT J.6B UPDATE QUERY—EDIT RECORD (RC-6B.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
<CFQUERY NAME="DeptData" DATASOURCE="RobCor">
    SELECT * FROM DEPARTMENT WHERE (DEPARTMENT.DEPT_ID='#form.DEPT_ID#')
</CFQUERY>
<CFQUERY NAME="USRLIST" DATASOURCE="RobCor">
   SELECT USR_ID, USR_LNAME, USR_FNAME, USR_MNAME
      FROM USER
      WHERE USR_ID NOT IN (SELECT USR_ID FROM DEPARTMENT
                           WHERE USR_ID > 0 AND DEPT_ID <> '#form.DEPT_ID#')
      ORDER BY USR_LNAME, USR_FNAME, USR_MNAME
</CFQUERY>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Update Query: Edit Record Screen</B></CENTER>
</H1>
<FORM ACTION="rc-6c.cfm" METHOD="post">
<TABLE ALIGN="CENTER" BGCOLOR="Silver" BORDERCOLOR="Blue">
<TR>
<TD>
<CFOUTPUT QUERY="DeptData">
<PRE>
<INPUT TYPE="hidden" NAME="DEPT_ID" VALUE="#DEPTDATA.DEPT_ID#">
Department ID: <B>#DEPT_ID#</B><BR>
Description  : <INPUT TYPE="text" NAME="DEPT_DESC" VALUE="#DEPT_DESC#"SIZE=35 MAXLENGTH=35><BR>
Manager      : <SELECT NAME="USR_ID" SIZE=1><!--- select user from list --->
               <OPTION <CFIF #DEPTDATA.USR_ID# EQ "">SELECTED</CFIF> VALUE="">----------
</CFOUTPUT>
<CFOUTPUT QUERY="USRLIST">
               <OPTION <CFIF #DEPTDATA.USR_ID# EQ #USRLIST.USR_ID#>SELECTED</CFIF> VALUE="#UsrList.USR_ID#">
               [#USR_ID#] #USR_LNAME#, #USR_FNAME#, #USR_MNAME#
</CFOUTPUT>
                </SELECT>
</PRE>
</TD>
<TD VALIGN="TOP">
<INPUT TYPE="submit" VALUE=" Update ">
</FORM>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu">
</FORM>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

FIGURE J.14 UPDATE QUERY—EDIT RECORD SCREEN



As you compare the rc-6b.cfm script and the sequence shown in Figure J.14, note that the script generates the following actions:

- Lines 4–6 read the Department data, using the *"#form.DEPT_ID#"* parameter received from the rc-6a.cfm script.
- Lines 7–13 specify and execute a key query. When a new department is to be inserted, the "USRLIST" query lists all users who are not already managers of a department. This query applies only to *new* record inserts; it cannot be used to edit an *existing* record.

To see why it is necessary to modify the original USRLIST query, let's suppose that USR_ID = 13 is the current ACTG department's manager. If you try to *edit* the ACTG department record using the *original* USRLIST query, the list of "available users" will yield all users who are *not* already in the DEPARTMENT table. Therefore, because the ACTG department's current manager *is* listed in the DEPARTMENT table's USR_ID column, the current manager will not appear on the list of users who are available as ACTG department manager. In short, if you try to edit (update) a record using the original USRLIST query, you will be forced to select a manager other than the current one. That is clearly not what's intended!

To avoid the just-described dilemma, the nested query criteria of the USRLIST query must be modified to exclude the "to be edited" department's DEPT_ID from the subquery. Therefore, line 11 specifies the nested query criteria to be WHERE USR_ID > 0 AND DEPT_ID <> '#form.DEPT_ID#'.

Given that modification, the ACTG department's current manager (USR_ID = 13) will appear on the list of available user IDs.

- Lines 19–40 produce the user edit form. This form allows the end user to modify only two DEPARTMENT table fields: department description (DEPT_DESC) and department manager (USR_ID).
  - *Because the DEPT_ID is the DEPARTMENT table's primary key, the end user cannot be allowed to modify its value*. The reason for this restriction is simple. Suppose that the end user edits the ACTG department, that is, the DEPT_ID = 'ACTG'. If the end user is permitted to change the DEPT_ID from 'ACTG' to 'ACTNG', the department data update will pass the DEPT_ID = 'ACTNG' form field to the update script. Unfortunately, the DEPT_ID = 'ACTNG' does not exist in the DEPARTMENT table. Therefore, the database will return an error to indicate that the end user is trying to update a record that does not exist—which is true: 'ACTG' exists, but 'ACTNG' does not.
- Line 25 creates an input form variable named DEPT_ID, to which the "#DEPTDATA.DEPT_ID#" value is assigned. In other words, the script assigns the current record value to the edit mode. Note that this variable is hidden, so it will not be shown on the screen. This value assignment ensures that the DEPT_ID is passed to the rc-6c.cfm script. (Remember that all form variables that are defined with an INPUT or SELECT form tag are passed to the called program.)
- Line 26 ensures that the current department ID is shown on the screen. Note that the end user cannot edit this value.
- Line 27 allows the end user to modify the department's description. Note that the INPUT tag sets the default value for this field to "#DEPT_DESC#", thus ensuring that the previous field's contents are displayed. The end user can then modify the displayed values.
- Lines 28–35 allow the end user to choose a manager for the selected department. These lines create a select box that lists all valid options for the manager field. The valid options are:
  - All users who are not managers.
  - If the department has a manager, the existing manager.
  - A null manager option to indicate that no manager has yet been assigned to the selected department.

  Given those options, the end user can set the manager to null, leave the current manager unchanged, or choose another manager. If the department being edited already has a manager, that manager must appear as the default (SELECTED) option.
- Line 29 allows the manager ID to be set to null. (Note that VALUE = "".) This line also contains a CFIF tag to evaluate the current value of the department's USR_ID field. If the USR_ID is "" (that is, the selected department does not have a manager), this null will be the SELECTED option. Otherwise, this option will appear on the list of options but will not be preselected. This null option must be available to ensure that a department's manager can be removed.
- Lines 31–34 use the CFOUTPUT tag to create OPTION entries for each of the user IDs in the "USRLIST" query. (Remember that the CFOUTPUT tag will loop through

each record in the named query.) For each added option, a CFIF tag compares the existing USR_ID in the department table (#DEPTDATA.USR_ID#) with the USR_ID being added (#USRLIST.USR_ID#). If the two are equal, the "SELECTED" keyword is added to the OPTION tag.

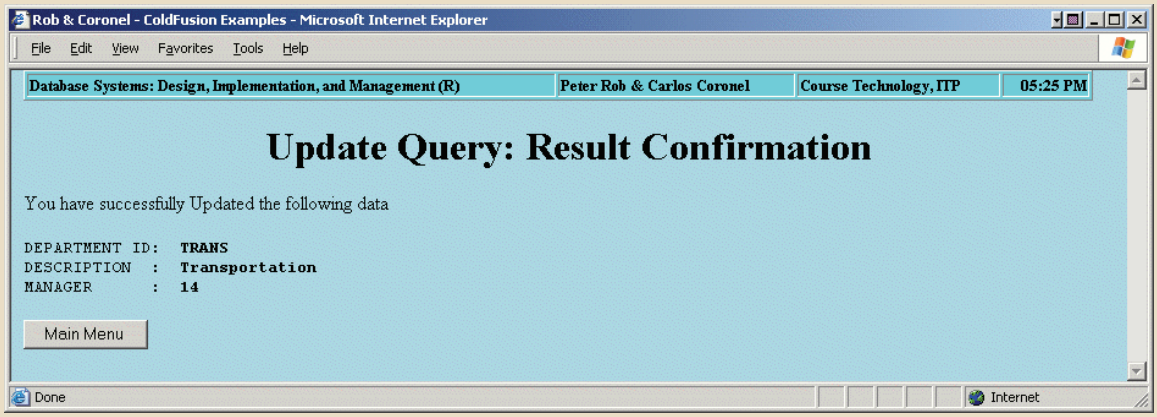
When the user clicks the form's Update button, Script J.6B (rc-6b.cfm) calls Script J.6C (rc-6c.cfm), passing its variable values.

The rc-6c.cfm script's output is shown in Figure J.15.

SCRIPT J.6C UPDATE QUERY—RESULT CONRMATION (RC-6C.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
<CFUPDATE DATASOURCE="RobCor" TABLENAME="Department">
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Update Query: Result Confirmation</B></CENTER>
</H1>
<CFOUTPUT>
You have successfully Updated the following data
<PRE>
DEPARTMENT ID:  <B>#DEPT_ID#</B>
DESCRIPTION  :  <B>#DEPT_DESC#</B>
MANAGER      :  <B>#USR_ID#</B>
</PRE>
</CFOUTPUT>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu ">
</FORM>
</BODY>
</HTML>
```

FIGURE J.15 UPDATE QUERY—RESULT CONRMATION SCREEN



The rc-6c.cfm script uses the CFUPDATE tag to update the DEPARTMENT table. The parameters for this tag are:

- *DATASOURCE = "datasource_name"* (the name of the ODBC database connection).
- *TABLENAME = "table_name"* (the name of the table to be updated).

The CFUPDATE tag uses the form fields passed from the calling page (DEPT_ID, DEPT_DESC, and USR_ID) to update the named table. As was true with the other .cfm

pages, the form created in the calling page must name its form fields to match the table column names. Failure to adhere to that naming requirement will generate a "variable not found" error.

The most basic web-based data management process requires at least three actions: create a new record, modify an existing record, and delete a record. The first two have been discussed, so it's time to examine the "delete" action.

J-1i Deleting Data

The "delete" query examined in this section enables the end user to delete a department record. As was true for the update query, the delete query's implementation requires three pages.

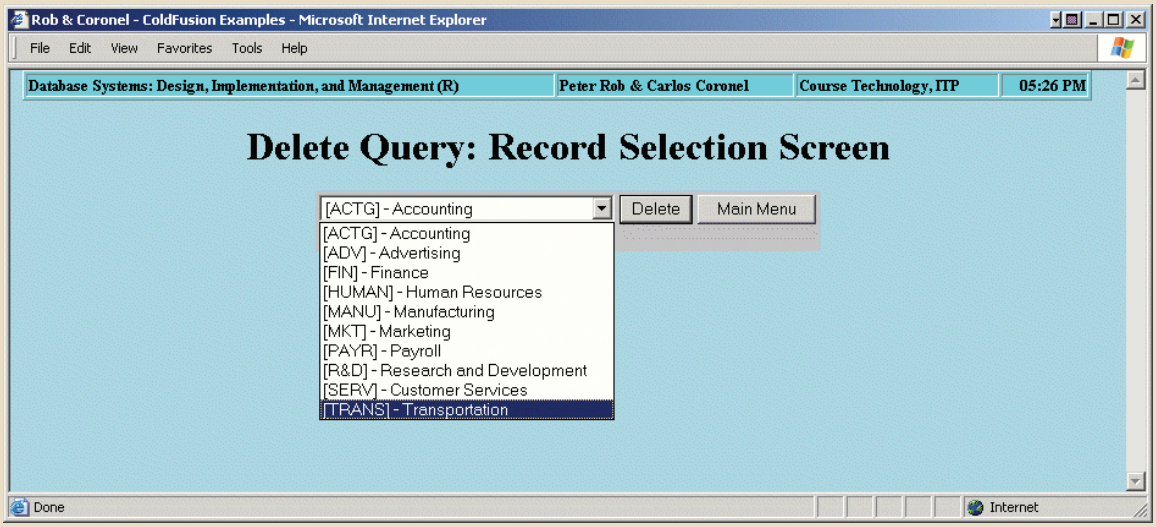
- The first page (the rc-7a.cfm script shown in Script J.7A) allows the end user to select the record to be deleted. When the user clicks the form's Delete button, the rc-7b.cfm script is invoked and the DEPT_ID form field value is passed to it.
- The second page (the rc-7b.cfm script shown in Script J.7B) reads the selected record and displays its data on the screen. This query also performs a referential integrity check to ensure that the end user cannot delete a department that still contains users. When the user clicks the Delete button, this page calls the third page, passing the DEPT_ID form's field value to that page.
- The last page (the rc-7c.cfm script) deletes the department row from the database table, using the DEPT_ID form field value passed from its calling program, rc-7b.cfm.

Let's look at the first of these three scripts. The rc-7a.cfm script output is shown in Figure J.16.

SCRIPT J.7A DELETE QUERY—RECORD SELECTION (RC-7A.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
<CFQUERY NAME="Deptlist" DATASOURCE="RobCor">
    SELECT * FROM DEPARTMENT ORDER BY DEPT_ID
</CFQUERY>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1>
<CENTER><B>Delete Query: Record Selection Screen</B></CENTER>
</H1>
<TABLE ALIGN="CENTER" BGCOLOR="Silver">
<TR VALIGN="TOP">
<TD>
<FORM ACTION="rc-7b.cfm" METHOD="post">
    <SELECT NAME="DEPT_ID" SIZE=1>
    <CFOUTPUT QUERY="Deptlist">
        <OPTION VALUE="#DEPT_ID#">[#DEPT_ID#] - #DEPT_DESC#
    </CFOUTPUT>
    </SELECT>
    <INPUT TYPE=HIDDEN NAME="DEPT_ID_required" VALUE="DEPT_ID is required">
</TD>
<TD>
    <INPUT TYPE="submit" VALUE="Delete">
</FORM>
</TD>
<TD>
<FORM ACTION="rc-0.cfm" METHOD="post">
    <INPUT TYPE="submit" VALUE="Main Menu">
</FORM>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

FIGURE J.16 DELETE QUERY—RECORD SELECTION SCREEN



The rc-7a.cfm script allows the end user to select a record to be deleted. To trace its operations, let's examine the following lines:

- Lines 4–6 perform a query ("Deptlist") to retrieve all records from the DEPARTMENT table. This query result set will be used to display a record selection form.
- Lines 15–25 define the record selection form. When the user clicks the form's Delete button, the rc-7a.cfm script calls script rc-7b.cfm. As before, the rc-7a.cfm script form passes its DEPT_ID form field to the rc-7b.cfm script.
- Lines 16–20 create the SELECT form control.
- Lines 17–19 use a CFOUTPUT tag to dynamically create the OPTION list.
- Line 21 uses the INPUT tag to define the DEPT_ID field as a required field. ColdFusion uses this command to perform server-side validation on this field. If this field is left blank, ColdFusion will return the error message text entered in the "VALUE" parameter. Therefore, line 21 ensures that the end user selects a record before trying to delete it. (You can't delete a record without first specifying which one it is.) If the end user fails to select a record before clicking the Delete button, an ODBC database error message will result, indicating the attempted deletion of a nonexistent record—or, even worse, the deletion of all table rows!

The second script (rc-7b.cfm) performs the following two important functions:

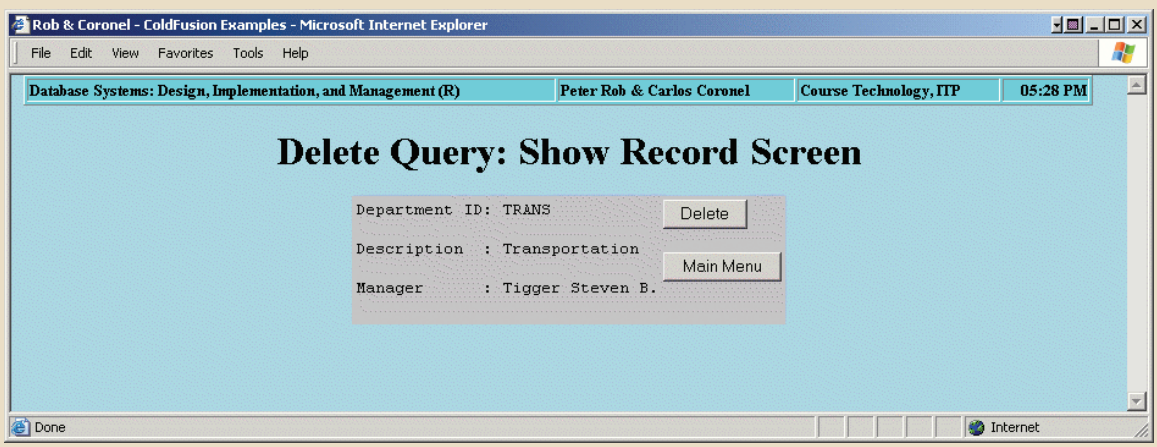
- It reads the record to be deleted and presents its data on the screen to let the end user confirm that this is the record (s)he wants to delete.
- It performs the referential integrity validation. Remember that the DEPARTMENT and USER tables maintain a 1:M relationship expressed by "each department may have one or more users." Therefore, the end user cannot be allowed to delete a department if it still contains users.

Script rc-7b.cfm is shown in Script J7.B. The rc-7b.cfm script output is shown in Figure J.17.

SCRIPT J.7B DELETE QUERY—SHOW RECORD (RC-7B.CFM)

```
<HTML>
<HEAD>
<TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
</HEAD>
<CFQUERY NAME="DeptData" DATASOURCE="RobCor">
    SELECT * FROM DEPARTMENT WHERE (DEPARTMENT.DEPT_ID='#form.DEPT_ID#')
</CFQUERY>
<CFIF #DEPTDATA.USR_ID# IS NOT "">
    <CFQUERY NAME="Usrdata" DATASOURCE="RobCor">
        SELECT USR_ID, USR_LNAME, USR_FNAME, USR_MNAME FROM USER
         WHERE (USER.USR_ID = #deptdata.usr_id#)
    </CFQUERY>
</CFIF>
<CFQUERY NAME="UsrTot" DATASOURCE="RobCor">
    SELECT COUNT(*) AS T1 FROM USER
      WHERE (USER.DEPT_ID = '#form.DEPT_ID#')
</CFQUERY>
</HEAD>
<BODY BGCOLOR="LIGHTBLUE">
<H1><CENTER><B>Delete Query: Show Record Screen</B></CENTER></H1>
<FORM ACTION="rc-7c.cfm" METHOD="post">
<CFOUTPUT QUERY="DeptData">
<INPUT TYPE="hidden" NAME="DEPT_ID" VALUE="#deptdata.DEPT_ID#">
<INPUT TYPE="hidden" NAME="DEPT_DESC" VALUE="#deptdata.DEPT_DESC#">
<INPUT TYPE="hidden" NAME="USR_ID" VALUE=#DEPTDATA.USR_ID#>
<TABLE ALIGN="CENTER" BGCOLOR="Silver" BORDERCOLOR="Blue">
<TR>
<TD>
<PRE>
Department ID: #DEPT_ID#<BR>
Description  : #DEPT_DESC#<BR>
Manager      : <CFIF #DEPTDATA.USR_ID# IS NOT "">#Usrdata.USR_LNAME# #Usrdata.USR_FNAME# #Usrdata.USR_MNAME#</CFIF>
</CFOUTPUT>
</PRE>
</TD>
<TD VALIGN="TOP">
<CFIF #USRTOT.T1# EQ 0>
    <INPUT TYPE="submit" VALUE=" Delete ">
<CFELSE>
    <SMALL><B>We cannot delete this record <BR>because there are dependent <BR>users assigned to this department</B></SMALL>
</CFIF>
</FORM>
<FORM ACTION="rc-0.cfm" METHOD="post"><INPUT TYPE="submit" VALUE="Main Menu"></FORM>
</TD>
</TR>
</TABLE>
</BODY>
</HTML>
```

FIGURE J.17 DELETE QUERY—SHOW RECORD SCREEN



Let's examine the rc-7b.cfm script to understand how it works.

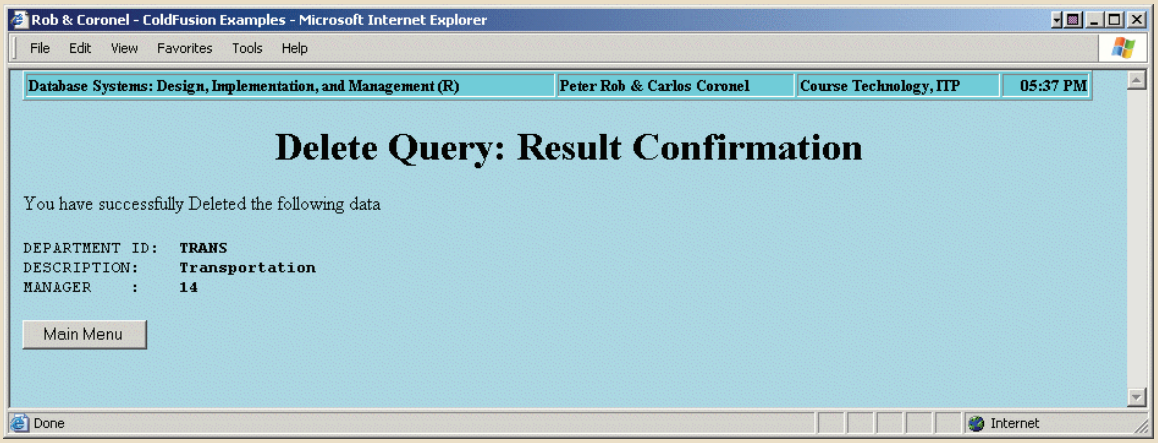
- Lines 5–7 use the CFQUERY tag to read the selected department data. The query uses the "*#form.DEPT_ID#*" form field passed from the rc-7a.cfm script.
- Lines 8–13 retrieve the department manager data from the USER table. Because this is an optional field, the user ID is checked first to see whether it is not null. If this non-null condition is met, the user data is read from the USER table, using the "#deptdata.usr_id#" value. If the user ID is null, there is no need to read the user data.
- Lines 14–17 perform referential integrity validation checks. The process starts by executing a query to see if users are still assigned to the department to be deleted. Note that the SQL query in lines 15 and 16 counts the number of users assigned to the department. (If this count yields a value greater than zero, the department contains at least one user.) Note that the count is stored in variable T1.
- Lines 21–42 define a form to display the department data and to confirm the record deletion. When the user clicks the Delete button, the rc-7c.cfm script (shown in Script J.7C) is called and the three variables (DEPT_ID, DEPT_DESC, and USR_ID) are passed to it.
- Line 23 defines the form's DEPT_ID field as "hidden," and the to-be-deleted DEPARTMENT table's DEPT_ID value is assigned to this hidden field. (Although this hidden input field does not show on the screen, it is passed to the next script.)
- Lines 24 and 25 perform the same function as line 23, defining the remaining form fields (DEPT_DESC and USR_ID) as "hidden" and assigning the corresponding department field values to the hidden form fields. These hidden form field values also are passed to the next script.
- Lines 30–32 display the department data for the record to be deleted. This action enables the end user to see the record to confirm that this is, in fact, the department record (s)he wants to delete. Note that the fields specified in lines 30 and 31 use the "DeptData" query source as specified in line 22's CFOUTPUT tag. In contrast, note that line 32's field name prefix indicates that it uses fields from the "Usrdata" query.
- Line 32 uses a CFIF tag to check whether there are user data. If the "#deptdata.usr_id#" is not null, the user data are displayed.
- Lines 37–41 check to see if any users are assigned to the department. If user records do not exist (#usrtot.t1# EQ 0), the Delete button is shown. (Check line 15 again and note that its count is now the basis for the condition check.) If the count is anything other than zero, the form displays a message to indicate that users are still assigned to this department and the Delete button is not shown.

If the record can be deleted and the user clicks the Delete button, script rc-7c.cfm (Script J.7C) is called and the (hidden) form fields are passed to it. The rc-7c.cfm script output is shown in Figure J.18.

SCRIPT J.7C DELETE QUERY—RESULT CONRMATION (RC-7C.CFM)

```
1  <HTML>
2  <HEAD>
3  <TITLE>Rob & Coronel -  ColdFusion Examples</TITLE>
4  <CFQUERY NAME="DeleteDept" DATASOURCE="RobCor">
5      DELETE FROM DEPARTMENT WHERE (DEPT_ID = '#FORM.DEPT_ID#')
6  </CFQUERY>
7  </HEAD>
8  <BODY BGCOLOR="LIGHTBLUE">
9  <H1>
10 <CENTER><B>Delete Query: Result Confirmation</B></CENTER>
11 </H1>
12 <CFOUTPUT>
13 You have successfully Deleted the following data
14 <PRE>
15 DEPARTMENT ID:  <B>#DEPT_ID#</B>
16 DESCRIPTION:    <B>#DEPT_DESC#</B>
17 MANAGER    :    <B>#USR_ID#</B>
18 </PRE>
19 </CFOUTPUT>
20 <FORM ACTION="rc-0.cfm" METHOD="post">
21     <INPUT TYPE="submit" VALUE="Main Menu ">
22 </FORM>
23 </BODY>
24 </HTML>
```

FIGURE J.18 DELETE QUERY—RESULT CONRMATION SCREEN

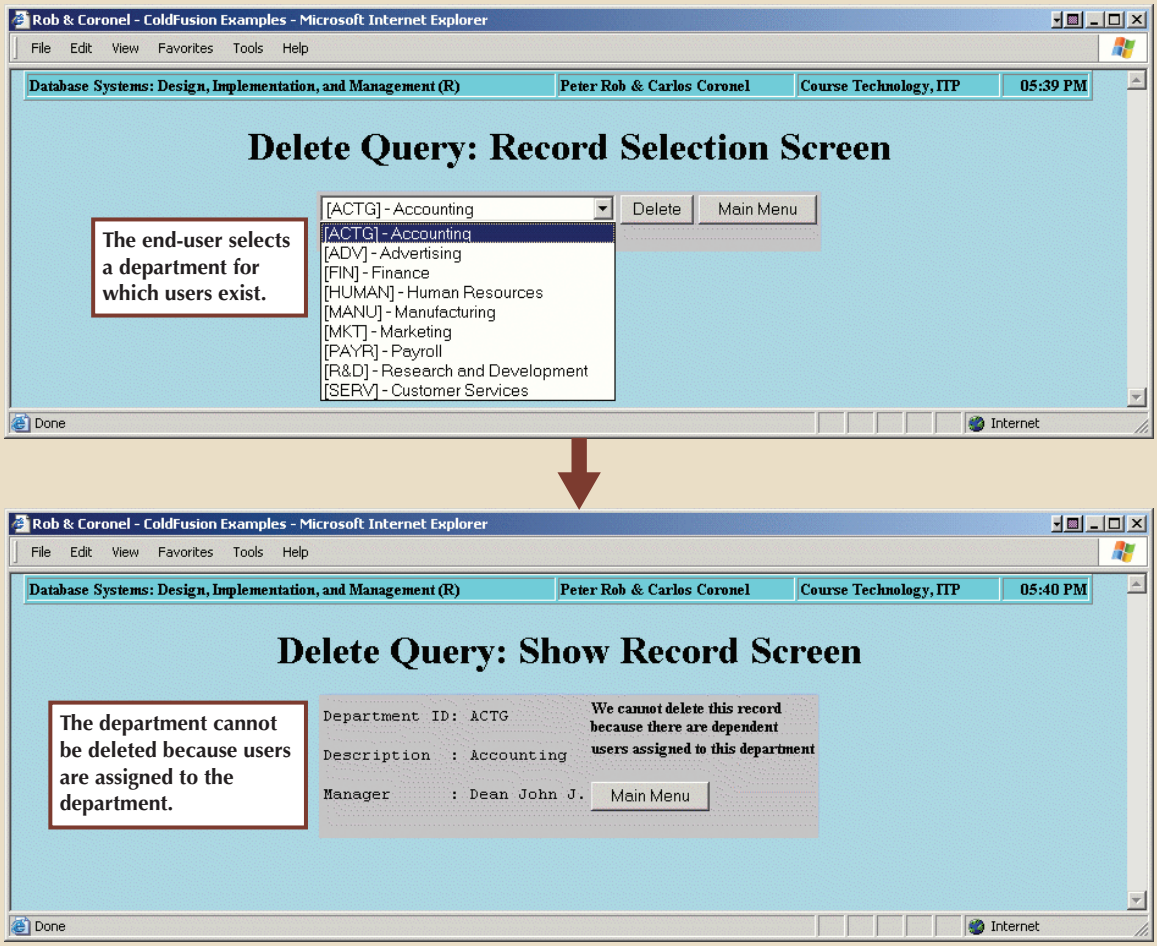


The rc-7c.cfm script deletes the department record from the database and displays a confirmation screen. To see how the script accomplishes those tasks, examine the following lines:

- Lines 4–6 perform a query used to delete the department record from the database. This query executes the SQL "Delete" statement, using the form's DEPT_ID field received from the rc-7b.cfm script.
- Lines 10–17 confirm that the record has been deleted, and they display the deleted data.

Figure J.19 shows how the delete sequences work. As you examine the screens, note that the attempt failed to delete a department that still contains users. Also note that the Delete button is not shown in the second screen. There is, after all, no reason to display a Delete button if the selected department record cannot be deleted.

FIGURE J.19 DELETE QUERY—RECORD VALIDATION



Note

The ColdFusion techniques presented in this appendix represent just the tip of the proverbial iceberg in the development of database-enabled web applications. At the time of this writing, ColdFusion provides hundreds of additional tags and functions to help you develop professional web applications properly. Although the preceding examples are far from exhaustive, they do provide a compelling illustration of the web interfaces power and flexibility.

In today's increasingly web-driven business environment, there is little doubt that you will work with databases that are web-enabled. Given the clear competitive advantages

provided by database web access, it is tempting to focus on the web side of the web-database equation. Yet it is important to realize that a web interface to a badly designed database is a recipe for database disasters. On the other hand, good database design and implementation, coupled with sound web development techniques, yield countless business tactical and strategic advantages, virtually boundless professional opportunities, and personal satisfaction.

J-2 Internet Database Systems: Special Considerations

Internet database systems involve more than just the development of database-enabled web applications. In addition, certain issues must be addressed when web interfaces are used as the gateway to corporate and institutional databases. For example, data security, transaction management, client-side data validation, and many other operational and management challenges must be met. Although many of those issues were discussed in detail in Chapter 15, Database Connectivity and Web Technologies, they are particularly relevant to web database development. Therefore, some of them are revisited here.

Whether you are talking about databases in a conventional client/server environment or in the latest Internet arena, database systems development requires sound database design and implementation. The database system must exist within a secure environment that is well suited to maintaining well-monitored and -protected data access, robust transaction management with a focus on data integrity maintenance, and solid data recovery. Finally, from the end user's and business manager's points of view, the database is not particularly useful unless its front end is characterized by user-friendly, information-capable, and transaction-supportive end-user applications.

Production database and data warehouse designs are not affected—*at the conceptual level*—by the change from the conventional client/server environment to the Internet's client/server environment. Therefore, the basic design processes and procedures need not be revisited. However, Internet database application development is quite different from that found in the traditional client/server environment. And given the vastness of the Internet, development issues such as security, backup, and transaction volume are even more critical than they were in the traditional environment.

Clearly, concurrent database access by multiple heterogeneous clients affects how transactions are defined and managed. Support for multiple data sources and types, the advent of increasing platform independence and portability, process distribution and scalability, and open standards have a major effect on how applications are developed, installed, and managed.

No doubt, database application development is most affected by the Internet. Characteristics of the Internet—particularly its web service—fundamentally change the way that applications work. The stateless nature of the web has a major impact on how database queries are presented and executed. Just think of the web's request-reply model and how different it is from the conventional programmer's view.

If database systems are to be developed and managed intelligently, today's database administrator must understand the Internet-based business environment in order to cope successfully with the issues that drive the development, use, and management of web-to-database interfaces. Since it all begins and ends with data, you will begin by looking at the incredibly diverse data types that are supported in the Internet environment.

J-2a What Data Types Are Supported?

Web development requires the concurrent management of many and quite different data types. Typically, conventional databases support data types such as Julian dates, various types of numbers (integer, floating point, and currency), and text (fixed and variable length). Most leading RDBMS vendors support extended data types such as binary and OLE (Object Linking and Embedding) objects in the conventional database environment.

Because interactive websites tend to integrate data from multiple sources—word-processed documents, spreadsheets, presentations, pictures, movies, sounds, and even holograms—some web and database designers use extended data types to store page components (in binary format) for later incorporation into the webpages. Although the page arrangement may provide better data organization from the database point of view, several issues must be addressed.

- How does someone store and extract data objects such as documents, pictures, and movies through a web browser? Remember that the web client expects every page component to be a file stored in the web server's directory. Therefore, the DBMS or the web-to-database middleware must provide special functions or subroutines that allow objects to be extracted dynamically from a database field to the web server's directory, and vice versa.
- How much overhead will be created by the storage of binary objects in the database? How robust must the DBMS be to handle binary object transactions? What are the limitations for extended or OLE data types? How many extended or OLE data type fields can tables have?
- Does the client browser support the data type of the object being accessed? Are the necessary plug-ins available? Is there a way to automatically translate documents from their native format to HTML? For example, a PowerPoint presentation can be viewed within an Internet Explorer browser, but a plug-in might be required to open it in a different browser.
- Finally, storing pictures or multimedia presentations in the database can very quickly increase the size of the database. Does the DBMS support very large databases? What about transaction speed? The concurrent insertion and extraction of binary objects in database fields can take quite a toll on database transaction performance. How many users are going to access the database? How frequently?

Web-to-database interface design must juggle all of those issues and find the right balance to ensure that the database does not become the web-based system's bottleneck.

J-2b Data Security

Security is a key issue when databases are accessible through the Internet. Most DBMS vendors provide interfaces to manage database security. When you create a database web interface, security can be implemented in the web server, in the database, and in the networking infrastructure. In many ways, building multiple firewalls is the essence of Internet database security.

At the web server level, most web clients and servers can perform secure transactions by using encryption routines at the TCP/IP protocol level. Clients and servers can exchange security certificates to ensure that the clients and servers are who they say they

are. Therefore, you must ensure that the clients and servers are properly registered and that they have compatible encryption protocols. Also, the web administrator can use TCP/IP addresses and firewalls to restrict access to the site. The firewalls ensure that only authorized data travel outside the company.

All RDBMS vendors provide security mechanisms at the database end, providing some form of login authentication for users who are trying to access the database. At the SQL level, administrators can use the GRANT and REVOKE commands to assign access restrictions to tables and/or to specific SQL commands.

Web-to-database middleware vendors usually have several security mechanisms available for interfacing with databases. For example, when using ODBC data sources, the administrator can restrict end-user access to certain SQL statements (such as SELECT, UPDATE, INSERT, or DELETE) or to some combination of those commands. And although the webpages operate in the request-reply model, the use of web interfaces does not preclude the creation of algorithms to guarantee data entity and referential integrity requirements. Data security measures must also include logs to relate data manipulation activities to specific end users. Those logs ensure that each database update is directly associated with an authorized user.

Security must also be extended to support electronic commerce, or e-commerce. That support is key to the website's ability to execute secure business transactions over the Internet. If a vendor wants to be able to take credit card orders over the Internet, the order processing, rooted in a production database environment, must have strict security mechanisms in place to safeguard the transactions. In addition, the order transaction must be able to interact *securely* with multiple sites—such as distributors and banks—making sure that the transaction information cannot be modified and that the information cannot be stolen.

J-2c Transaction Management

Although the preceding comments focus on transaction-management issues that must be addressed for e-commerce to be conducted successfully, the concept of database *transactions* is foreign to the web. Remember that the web's request-reply model means that the web client and the web server interact by using very short messages. Those messages are limited to the request for and delivery of pages and their components. (Page components may include pictures, multimedia files, and so on.) The dilemma created by the web's request-reply model is that:

- The web cannot maintain an open line between the client and the database server.
- The mechanics of a recovery from incomplete or corrupted database transactions *require* that the client must maintain an open communications line with the database server.

Clearly, the creation of mission-critical web applications mandates support for database transaction management capabilities. Given the just-described dilemma, *designers must ensure proper transaction management support at the database server level.*

Many web-to-middleware products provide transaction management support. For example, ColdFusion provides this support through the use of its CFTRANSACTION tag. If the transaction load is very high, this function can be assigned to an independent computer. By using that approach, the web application and database servers are free to perform other tasks and the overall transaction load is distributed among multiple processors.

J-2d Denormalization of Database Tables

When the web is used to interact with databases, the application design must take into account the fact that the web forms cannot use the multiple data entry lines that are typical of parent–child (1:M) relationships. Yet those 1:M relationships are crucial in e-commerce. For example, think of order and order line, or invoice and invoice line. Most end users are familiar with the conventional GUI entry forms that support multitable (parent–child) data entry through a multiple-component structure composed of a main form and a subform. Using such main-form/subform forms, the end user can enter multiple purchases associated with a single invoice. All data entry is done on a single screen.

Unfortunately, the web environment does not support this very common type of data entry screen. As illustrated in the ColdFusion script examples, the web can easily handle single-table data entry. However, when multitable data entries or updates are needed—such as order with order lines, invoice with invoice lines, and reservation with reservation lines—the web falls short. Although implementing the parent/child data entry is not impossible in a web environment, its final outcome is less than optimum, usually counterintuitive, less user-friendly, and prone to errors.

To see how the web developer might deal with the parent/child data entry, let's briefly examine how you might deal with the ORDER and ORDER_LINE relationship used to store customer orders. Using an applications middleware server such as ColdFusion to create a web front end to update orders, one or more of the following techniques might be used:

- Design HTML frames to separate the screen into order header and detail lines. An additional frame would be used to provide status information or menu navigation.
- Use recursive calls to pages to refresh and display the latest items added to an order.
- Create temporary tables or server-side arrays to hold the child table data while in the data entry mode. This technique is usually based on the bottom-up approach in which the end user first selects the products to order. When the ordering sequence is completed, the order-specific data, such as customer ID, shipping information, and credit card details, are entered. Using this technique, the order detail data are stored in the temporary tables or arrays.
- Use stored procedures or triggers to move the data from the temporary table or array to the master tables.

Although the web itself does not support the parent/child data entry directly, it is possible to resort to web programming languages such as Java, JavaScript, or VBScript to create the required web interfaces. The price of that approach is a steeper application development learning curve and a need to hone programming skills. And while that augmentation works, it also means that complete programs are stored outside the HTML code that is used in a website.

Key Terms

ColdFusion Markup Language (CFML), J-3
script, J-3
stateless system, J-16
tag, J-3
web application server, J-2

Review Questions

1. What are scripts, and how are they created in ColdFusion?
2. Describe the basic services provided by the ColdFusion web application server.
3. Discuss the following assertion: The web is not capable of performing transaction management.
4. Transaction management is critical to the e-commerce environment. Given the assertion made in Question 3, how is transaction management supported?
5. Describe the webpage development problems related to database parent/child relationships.

Problems

In the following exercises, you are required to create ColdFusion scripts. When you create these scripts, include one main script to show the records and the main options, for a total of five scripts for each table (show, search, add, edit, and delete). Consider and document foreign key and business rules when creating your scripts.

1. Create ColdFusion scripts to search, add, edit, and delete records for the USER table in the RobCor data source.
2. Create ColdFusion scripts to search, add, edit, and delete records for the INVTYPE table in the RobCor data source.
3. Create ColdFusion scripts to search, add, edit, and delete records for the VENDOR table in the RobCor data source.
4. Modify the insert scripts (rc-5a.cfm and rc-5b.cfm) for the DEPARTMENT table so that the users who can be manager of a department are only those who belong to that department.
5. Create an Order data-entry screen, using the ORDERS and ORDER_LINE tables in the RobCor data source. To do this, you can use frames and other advanced ColdFusion tags. Consult the online manual and review the demo applications.