

# Switchboards and Security, Plus Two SQL Answers

Wayne Wallace



Wherein Wayne Wallace solves two problems in one question: finding out whether a user can access a form and configuring a switchboard to access only the permitted forms. Not content with what the customer asks for, he provides an even better solution. Wayne then looks at two issues in SQL: non-normalized data and logical Notes.

I was asked to convert a non-Access database made by a large development company. Their interface was set up so that when a user signs in, the program checks the user's rights and opens the switchboard with only those items that the user can access. Also, the pushbuttons on the interface were sorted, leaving no gaps on the main form. Is it possible to create this kind of interface for an Access database, and, if the answer is positive, is there a short way to describe the procedure?

There are really two problems here:

- Determining the user's permissions
- Managing the switchboard

And I don't have a short way to describe anything.

I'll start with determining the user's permissions. For this discussion, I'll assume that you're re-creating the application in Jet, which lets me avoid discussing SQL Server security. This actually isn't much of a limitation. From a switchboard, you're really looking at what forms your users can open. So I'm going to focus on managing the security for your application's forms, which are a constant between Jet and SQL Server applications.

Of course, Access security can cover more than the forms in your application. It doesn't do your users much good, for instance, to give them rights to open a form and deny them rights to the table that the form depends on. So, I'm also going to assume that you've coordinated the security settings among all of your Access objects so that controlling access to your forms is a reasonable thing to do.

With the problem sufficiently limited so that I can give you an answer, the first step is to retrieve the security permissions for a particular form. DAO provides the easiest mechanism:

1. From the Container's collection of the CurrentDB, access the Forms container.

2. Using the Container's Documents collection, retrieve the Form that you want to check.
3. Retrieve the form's AllPermissions property.

Rights to an object can either be assigned directly to a user, or assigned indirectly by assigning the rights to a group and assigning the user to the group. The AllPermissions property returns the combination of those two sets of rights as a long value:

```
Dim lngPerms As Long
lngPerms = CurrentDB.Containers("Forms")._
Documents("MyForm").AllPermissions
```

The long value that's returned can be checked by And-ing the value with specific values in DAO's PermissionsEnum (a set of constants whose names all begin with "dbsec"). Where a particular right is present, the result of the And-ing will be the constant that the Permissions was And-ed with. To use the dbsec constants you'll need to set a reference to the Microsoft DAO library in your Tools | References menu (this isn't set by default in all versions of Access). This code checks my lngPerms variable to see if the dbSecFullAccess permission has been granted:

```
If (lngPerm And dbSecFullAccess) = _
dbSecFullAccess Then
```

You could also check the security for the tables (or queries) that a form depends upon and add the buttons to your switchboard based on that test.

Let's look at the second part of your problem—setting up the switchboard. The simplest way to handle this is, at design time, to drop all the buttons that you need on the form with the code behind each button to open the appropriate form. Figure 1 shows my initial layout, which is ugly, but I'll fix that later. With the buttons placed on the page, set each button's Visible to property to False.

At runtime, in the Form Load event, you can check to see whether a user is allowed to use a form, and then make its button visible. Now you'll need to distribute your buttons on the form by setting the button's Top and Left properties.

The following sample code does just that. I've assumed a single column of buttons that are 1440 twips

(one inch) in from the left-hand edge and are each 500 twips (about one third of an inch) in height. As I make each button visible, I increment a counter of visible buttons and use that counter to set the button's distance from the top of the form:

```
Dim lngPerms As Long
Dim ingButton As Integer

ingButton = 1
lngPerms = CurrentDb.Containers("Forms"). _
Documents("Form1").AllPermissions
If lngPerms And dbSecFullAccess = dbSecFullAccess Then
    Me.cmdForm1.Visible = True
    Me.cmdForm1.Left = 1440
    Me.cmdForm1.Top = ingButton * 500
    ingButton = ingButton + 1
End If
lngPerms = CurrentDb.Containers("Forms"). _
Documents("Form2").AllPermissions
If lngPerms And dbSecFullAccess = dbSecFullAccess Then
    Me.cmdForm2.Visible = True
    Me.cmdForm2.Left = 1440
    Me.cmdForm2.Top = ingButton * 500
    ingButton = ingButton + 1
End If
lngPerms = CurrentDb.Containers("Forms"). _
Documents("Form2").AllPermissions
If lngPerms And dbSecFullAccess = dbSecFullAccess Then
    Me.cmdForm2.Visible = True
    Me.cmdForm2.Left = 1440
    Me.cmdForm2.Top = ingButton * 500
    ingButton = ingButton + 1
End If
```

The results appear in [Figure 2](#).

I'd be the first person to admit that this is ugly code, but I've written it to be obvious rather than to be easily maintained or enhanced. For instance, by giving each button on the switchboard the same name as the form that it will open, you can turn the repeated code into a function that accepts a form name and a position and returns a new position:

```
Function ShowButton(strForm As String, _
    intButton As Integer) As Integer
Dim lngPerms As Long

lngPerms = CurrentDb.Containers("Forms"). _
Documents(strForm).AllPermissions

If lngPerms And dbSecFullAccess = _
    dbSecFullAccess Then
    Me.Controls(strForm).Visible = True
    Me.Controls(strForm).Left = 1440
    Me.Controls(strForm).Top = ingButton * 500
    ShowButton = intButton + 1
End If
```

Your Form Load event now just needs to run through

the controls on the form calling this routine (I've assumed that there's nothing but buttons on the switchboard):

```
Dim ctl As Control
Dim intButton As Integer

ingButton = 1
For Each ctl In Me.Controls
    ingButton = ShowButton(ctl.Name, intButton)
Next
```

### Okay, any other solutions?

You might want to consider using a toolbar instead of using buttons. To create a toolbar, you'll need to add a reference to the Microsoft Office library. Toolbars are designed to be more configurable than controls on an Access form. With a toolbar you won't, for instance, have to add a group of invisible buttons and then move them around as I had to with buttons on the switchboard. Instead, you can just create a toolbar button for each form that the user is allowed to open. You could even use a graphic and a ToolTip to identify the button, though I'll just use text in my sample code.

First, I'll add a toolbar to put the list of available forms on. Adding a toolbar that already exists will generate an error. To handle that, I check that the toolbar isn't already there after calling the CommandBars Add method that adds my "Form List" toolbar and before I do any more work (like making the CommandBar visible):

```
Dim cbr As CommandBar
Dim cbc As CommandBarControl

On Error Resume Next
Set cbr = Application.CommandBars.Add("Form List", _
    msoBarTop, , True)

If Err.Number = 0 Then
    cbr.Visible = True
```

The True parameter at the end of the Add method flags my toolbar as temporary—the toolbar will be discarded when I shut Access down. This lets me rebuild the toolbar for each user without worrying about what's already on the bar.

With the toolbar in place, this sample code adds two buttons to open two forms. For each button, I specify that the button has to display a caption (with the Style property), the text to display (with the Caption property),

Figure 1. The switchboard at design time.

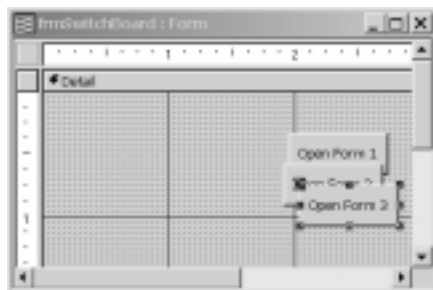


Figure 2. The switchboard after it has configured itself.



and the routine to run (with the OnAction property):

```

Set cbc = cbr.Controls.Add(msoControlButton)
cbc.Style = msoButtonCaption
cbc.Caption = "Open Form 1"
cbc.OnAction = "=OpenForm1()"

Set cbc = cbr.Controls.Add(msoControlButton)
cbc.Style = msoButtonCaption
cbc.Caption = "Open Form 2"
cbc.OnAction = "=OpenForm2()"
Else
Err.Clear
End If

```

Figure 3 shows my very simple toolbar.

My problem is that I have a table of customers with four columns: 1QSales, 2QSales, 3QSales, and 4QSales. The total of the purchases made by the customer in the first quarter appears in 1QSales, the total of the purchases in the second quarter in 2QSales, and so on. My boss wants me to tell him which quarter has the best sales. I seem to be writing a lot of code...

I'm going to borrow a line from Peter Vogel's "Working SQL" columns: The problem isn't in your code, it's in your table layout. Your table violates the first rule of relational database design: A table shouldn't contain repeating fields. You've hit the problem head on.

The simplest way to solve your problem is to use a SQL statement that generates totals for all of your columns:

```

Select Sum(1QSales), Sum(2QSales),
Sum(3QSales), Sum(4QSales)
From Customer

```

Now you'll have to write some code to check which column has the largest number. This code isn't pretty, but after it's finished running it will return the name of the column with the highest total:

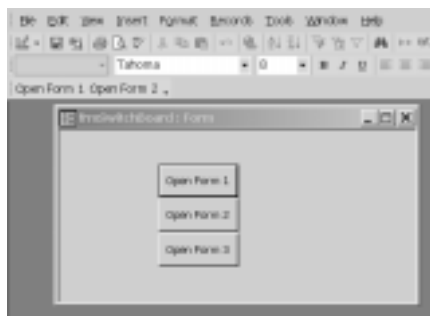
```

Function FindLargest() As Integer
Dim curHold As Currency
Dim rec As ADODB.Recordset
Dim intHigh As Integer

Set rec = CurrentDB.Connection.Execute( _
"Select Sum(1QSales), Sum(2QSales), " & _
"Sum(3QSales), Sum(4QSales) " & _
"From Customer"
intHigh = 0
curHold = rec(intHigh)

```

Figure 3. The form with a toolbar.



```

For ing = 1 to 3
If rec(ing) > curHold Then
curHold = rec(ing)
intHigh = ing
End If
Next
FindLargest = rec(intHigh).Name
End If

```

So how could your database design have eliminated this problem? A better design would move the sales data into a new table with three columns:

- CustomerId: The customer's unique identifier.
- SalesQuarter: The name of the quarter for the purchases (1QSales, 2QSales, and so on).
- TotalPurchases: The total purchases for this customer for the quarter.

The primary key for this table is a combination of CustomerId and SalesQuarter. With this design, you can determine the result just using SQL:

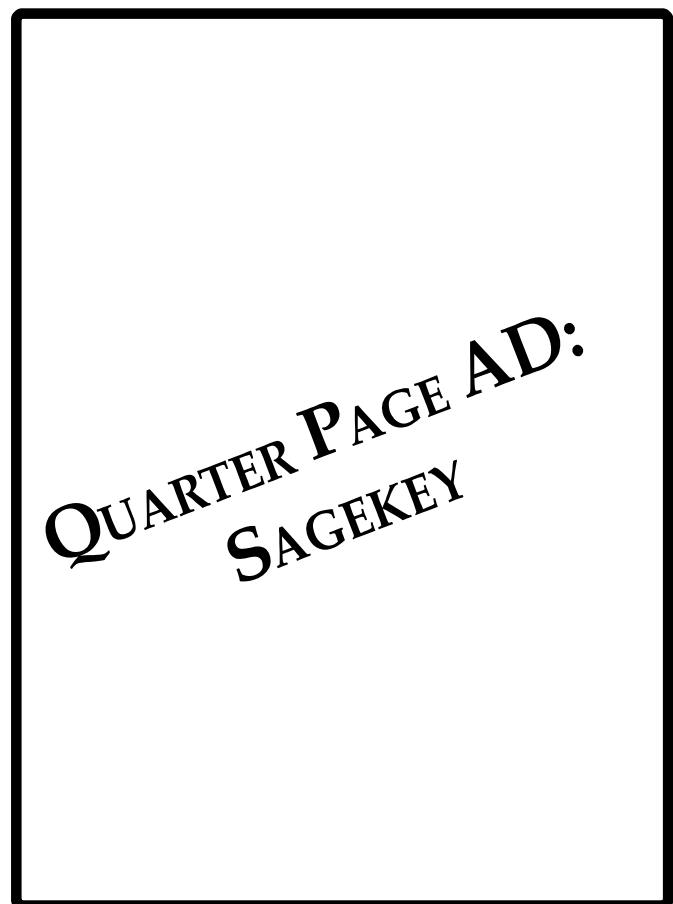
```

Select SalesQuarter, Sum(TotalPurchases) As TotalP
From Customer
Order by TotalP Desc

```

The quarter with the highest sales will be the first record in the query. By the way, just by adding the year to the value in the SalesQuarter column, this design lets you keep purchase history for as long as you want, not just the four quarters your current design limits you to.

Interestingly enough, you can mimic this result by



using a set of SQL statements to convert your unfortunate table design into the correct design. This query pulls out each column and actually consists of four separate queries (one for each column), joined together by the Union statement:

```
Select "Q1Sales" As SalesQuarter,
      Q1Sales As TotalPurchases
From Customer
Union
Select "Q2Sales" As SalesQuarter,
      Q2Sales As TotalPurchases
From Customer
Union
Select "Q3Sales" As SalesQuarter,
      Q3Sales As TotalPurchases
From Customer
Union
Select "Q4Sales" As SalesQuarter,
      Q4Sales As TotalPurchases
From Customer
```

You can then use this query as input to my previous query to get the result that you want.

I have a query that doesn't give me the results that I want. In my query I don't want any entries for dates that occur on a Saturday or Sunday. I'm using "Where Not(Day = 'Sunday' And Day='Saturday')". I get every record returned every time.

This happens so often that it could be classified as a "standard error." What you want are the records where the Day column isn't equal to 'Sunday' and where the Day column isn't equal to 'Saturday.' In a Where clause, the test would look like this:

```
Day <> 'Sunday' And Day <> 'Saturday'
```

This does *not* convert into:

```
Not(Day = 'Sunday' And Day = 'Saturday')
```

Let me look at why it doesn't convert before I give you the right test. If you remove the Not and the parentheses from your test, you can see that leaves this

as your Where clause:

```
Day = 'Sunday' And Day = 'Saturday'
```

This Where clause will obviously never be true because, in any single record, the Day field will never be set to both 'Sunday' and 'Saturday'. This Where clause is always false. Wrapping the Not around the clause simply reverses this result, meaning that your Where clause is always true. Since your Where clause is always true, your SQL statement returns every record.

I had this drummed into me back in my university days when I took a course in logic. When you pull the *Not* out of the *Not equals* in individual tests, you have to change the And that joins the two tests into an Or (and vice versa). The following two Where clauses are identical from a logical point of view because I change the And to an Or as I migrate the Not to the front of the test:

```
Day <> 'Sunday' And Day <> 'Saturday'
Not(Day = 'Sunday' Or Day = 'Saturday')
```

This conversion is known to us logicians as DeMorgan's law.

You can analyze this new test the same way that I did your original Where clause. What's inside the parentheses in the second of these Where clauses is:

```
Day = 'Sunday' Or Day = 'Saturday'
```

This test is true when your Day is equal to 'Sunday' or when Day is equal to 'Saturday'. By putting the statement inside a Not, the statement is false when Day is equal to 'Sunday' or 'Saturday'—and this is what you want. ▲

 [AA0603.ZIP at www.vb123.com/kb](http://www.vb123.com/kb/AA0603.ZIP)

Wayne Wallace is a builder-coder for a large pharmaceutical distributor in Canada. When he's not working with Access, he plays pick-up hockey and teaches night school.