Access Efficiency

Peter Vogel

95 97 2000 2002 DOWNLOAD

Here, Peter Vogel looks at the single most important change that you can make to your applications to make them run faster. It's also the one mistake that gets made the most often.

I would say that most of my Access consulting work comes from clients who have applications that are running too slowly. So far, in every case where I've been brought in, I've been able to significantly improve the speed of the application. One of the reasons that I've been so successful is that most of my clients have made the same mistake: They're retrieving too much data. I'd say that was the reason for every client, but my memory isn't what it should be, and there's the possibility that I've forgotten the one exception to this rule. This is the first area that you should concentrate on when trying to speed up your application.

I know that this isn't what you hear from experts when it comes to improving performance. Experts will tell you to make better use of indexes, use SQL in place of recordset processing, convert SQL statements in your code to queries in your database, and so on. In fact, I've provided that advice myself on more than one occasion. But these aren't fundamental improvements.

For instance, as I pointed out in last month's Working SQL column ("Efficient SQL"), indexes are often ignored by the DBMS when processing data. As far as storing your SQL statements as queries—yes, that does save you the cost of compiling your query when you execute it. Quite frankly, however, the compile time for most of the queries that you're using (especially the ones in your Recordsource properties) is probably so small that you can't see the improvement.

While all of these tips are good advice and well worth following, they aren't fundamental to making your application run faster. At the risk of giving up a good part of my consulting practice, here's the real secret to getting your application to run faster: Get less data.

The typical mistake

There are specific cases when developers retrieve too much data. I was asked to review one application that ran for four hours, used two tape drives, and generated temporary work files that filled two disk drives. Looking at the program, I discovered that it retrieved a row from a table, then retrieved some related rows from another table, then retrieved some rows from a third table that were related to the rows from the second table. The application then went back and got the next row from the very first table and repeated the process. After every row in the first table was processed, the application looked at the data that it had retrieved. It began this second pass by checking a field from the first table and discarding all the rows from all the related tables, based on that value. About 75 percent of the extracted data was thrown away in the second pass.

I rewrote the program to check the field in the first table before retrieving any other data. When I was done, the application ran in 90 minutes and used one small work file. Obviously, this is a special case, and there's not much in general rules that you can draw from this. Moving into Access, however, there are some common failures that you can guard against.

The most common mistake that I see developers make when retrieving too much data is using what I'll call an "unrestricted SQL query" in the Recordsource property of a form. An unrestricted SQL query is one that retrieves every row in a table (or most of them). The extreme form of an unrestricted query is just a table name. Equally guilty of the crime of retrieving too much data (and slowing down an application) is the SQL statement with no Where clause.

The opposite of an unrestricted query is what I call a "targeted SQL query." A targeted SQL query has a Where clause that, ideally, retrieves a single row from a table using the table's primary key. Targeted SQL statements aren't restricted to retrieving a single row. They do always contain Where clauses that carefully restrict the number of rows to retrieve.

A distant second to the unrestricted query is the "broadband query." A broadband query retrieves more fields than are necessary. However, the performance impact on your application that's inflicted by a broadband query is much less than the impact of an unrestricted SQL query. In this month's Download file (available at www.smartaccessnewsletter.com), you'll find an Access database that demonstrates the difference in speed between unrestricted, targeted, and broadband queries.

One of the ways that developers end up with this problem is by binding a form to a table. The form, of course, then displays all of the rows in that table. However, it's unlikely that the developer's users really want to see all the rows in the table. With a form that's bound to a table as a whole, developers typically use the form's Filter property to restrict the records to be displayed to the user. In this design, a form typically includes a drop-down list that lets users select which rows they want to see. The code behind this drop-down list sets the Filter property of the form in order to restrict the display of the data to the row(s) that the user actually wants.

There's lots that's wrong with this approach. To begin with, the users must wait for every row to be loaded before they can even select which row they want. After entering the criteria for the row that they really want, users must then wait while the enormous recordset retrieved when the form was loaded is reduced to just the few records actually desired. Fortunately, because the recordset is probably being held in memory, this filtering process goes quickly.

I've had developers explain to me that this is the only approach that they can take. Since the developer doesn't know, for instance, which customer a user will want to see, the developer's only alternative is to retrieve all the customer rows and then let the user select one. Of course, this means that when the form is opened, the user is presented with the first customer in the list, which is almost certainly not the one that they want.

The solution

A better design is to present the users with an empty form and let them select the row that they want. Based on the data that the user enters, you can then retrieve exactly what the customer wants. The form loads faster initially, and the users wait only to retrieve the data that they want.

Is there a cost here? Yes, of course. If a user brings up a customer row, then asks for another customer, and then another, it's possible that the total wait time for four or five retrievals of individual rows will be greater than the wait time to load all the rows. I suspect that you'll find, however, that the application that loads quickly and takes a minimal amount of time between rows is perceived to be faster than an application that takes a long time to load but moves more quickly between rows.

It's also a mistake to think that if you retrieve all the rows when opening your form it will make moving between those rows free. If the users are making updates, the users will have to wait as they leave each row to send their updates back to the server. Unless

Jet vs. SQL Server

Getting too much data is a problem with both Jet and SQL Server back ends. However, if you're retrieving too much data on some of your forms with Jet and then switch over to SQL Server, you'll probably notice that the problem gets worse. To understand why, you need to look at where your data is processed with the two DBMSs.

With Jet, every user gets his or her own copy of the database engine: Jet runs on each user's computer. Since your data is probably kept in an MDB file on a network server, this means that all of the data must be brought to the user's computer in order to be processed by the database engine. If you issue a request for the customers who live in New York, then all of the customer records must be brought to the user's computer to be examined so that the non-New York customer records can be discarded. If there's an index on the appropriate field, Jet may just retrieve the index and use that to determine which rows to retrieve. Once the index is processed, though, Jet must issue a series of requests back to the file server with the MDB file to get the matching rows.

With SQL Server (or any other client/server database), all that goes up to the database server is the SQL statement. All that comes back is the record(s) that you asked for. If you ask for all the customers who live in New York, the processing of the records happens on the database server, and only the requested records come back to the user's computer. Since the database engine is probably running on the same computer as the hard disk holding the data file, this processing can be very fast.

If SQL Server is so fast, why does the problem of retrieving too much data get worse when working with SQL Server? The answer is that, with Jet, all the records have to be brought down to the user's computer anyway. If you request too many records, the impact is reduced because all of the records are on the client computer. The difference between targeted and unrestricted queries is reduced.

On the other hand, with SQL Server, if you bring too many records down to the client, you're throwing away the major advantage of using a server database. Instead of sending up a single SQL statement and getting back the records that you want, you're clogging the network with unnecessary data. Users notice that some forms take a very long time to load (unrestricted SQL queries), and some forms load very quickly (targeted SQL queries). With Jet, they may not notice the difference because of the way that the data is processed.

the users are going to process a series of rows in the order that they were retrieved, they'll either have to enter the data to filter the recordset down to the row they want or scroll through the data to find the row they want.

Using unrestricted queries

Unless there's a compelling reason, you should never use unrestricted queries. Compelling reasons include:

- *Filling list boxes and drop-down lists*—If you have many of these objects on your form, then it probably explains why your form takes so long to load.
- *The user needs to see a list of data in order to see the data in context*—In a scheduling application, for instance, users may want to see appointments before and after the one they're interested in. You still don't want to retrieve every row in the table, though.
- *The user actually intends to scroll through almost every row in a table*—This is very unlikely.

The only other occasion that I feel justifies using unrestricted queries and the Filter property is for data analysis applications. In these kinds of applications, users retrieve a large amount of data and then spend their time resorting and filtering it in order to analyze the data. This is the best use for the Filter property.

The key thing to notice here is that it all comes back to the user's intention. You may feel that you're providing the user with extra functionality by using an unrestricted query in your Recordsource. You may even tell your users, "Now you can scroll through all the data if you want." But if the users have no intention or desire to use that feature, you're just slowing them down and providing no benefit at all.

This attitude of checking with your users even applies to drop-down lists. Before you bind the 12th list box on the form to your States table, check to see whether your users really need a list box with the names of the states. Yes, your users may enter a bad state name, but it's not going to happen very often. By putting that bound drop-down list on your form, you're slowing down every user, every time he or she uses your form in order to prevent a problem that occurs very infrequently. If you assume that your users know their state names, it might be faster for them if you just provide a text box with some editing code behind it that checks that the state name is accurate. I strongly suspect that no one searches through a 52-state list to find "Wyoming," anyway.

Implementing targets

So, how do you implement targeted queries? As I've already suggested, the first step is to present the users with a form that lets them ask for the data they're actually interested in. This form will have a blank Recordsource property to prevent it from retrieving any data at all. Once the users enter their criteria, you can generate and set the Recordsource from your code, like this:

```
Set Me.Recordsource = "Select * From Customers " & _
                "Where CustId = '" & Me.txtCustId & "'"
```

This is the least efficient solution because the Jet database engine will be passed a raw SQL statement that it will have to parse, check, compile, and optimize before executing. However, as I said before, this is probably not an expensive proposition, especially when compared to using an unrestricted query. A more efficient solution is to use a parameterized query and just set the query's parameters before calling it.

With SQL Server, as I discuss in the sidebar ("Jet vs. SQL Server," on page 8), having some forms with unrestricted queries and others with targeted queries can create greater disparities in runtime than the same code with a Jet database. Fortunately, with SQL Server and Access ADPs, you do have one property that can reduce the impact of unrestricted queries: Server filter. This property is applied to your unrestricted Recordsource on the server, reducing the data brought

Continues on page 20

users and permissions using DAO, ADOX, and DDL SQL scripts.

Several chapters are devoted to working with SQL Server. The information included is sufficient to help an Access developer get started with SQL Server. Enough basic training is provided to show you how to create database objects such as tables, views, stored procs, and the like. Though a brief T-SQL tutorial is provided, if you plan to do anything fancy, you'll want to invest in a good book on SQL Server.

The last set of chapters deals with creating a Web interface for your Microsoft Access applications. Data Access Pages (DAPs) were first introduced in Access 2000 to help developers create Web pages for displaying and editing your Access data. Deploying DAPs was problematic in Access 2000, but, according to the *ADH*, the Access 2002 version is much more mature. In addition to explaining how to use the Access user interface to create deployable DAPs, the book includes a tutorial on creating basic ASP pages, even touching on the new ASP.NET technology. One complaint: The chapter on integrating XML with Access 2002 covers all the technical aspects of importing and exporting data but lacks real-life scenarios where you might want to use this technology.

I usually like to end book reviews with a "What's the verdict?" section, but that question just doesn't apply to the *Access Developer's Handbook*. It's not a question of whether or not the book is a great value. There really is something for everyone in the *Access 2002 Developer's Handbook*. How much you personally can expect to leverage from these volumes depends on how much you may already know. For the serious Access developer, these books are a must-have. ▲

Danny J. Lesandrini, a Microsoft Certified Professional in Access, Visual Basic, and SQL Server, has been programming with Microsoft Access since 1995. He maintains a Web site containing Access-related code solutions at http://datafast.cjb.net and replies to all questions and comments sent to him via e-mail. datafast@attbi.com.

Access Efficiency...

Continued from page 9

down to the client. However, it's probably just as easy to reset the Recordsource as to reset the Server filter property.

So what should you do? First, if a form's response time is good enough, leave it alone! You have enough to do. Where there's a problem, stay away from the Filter property unless you absolutely need it. If you have a poorly performing form, make these changes:

1. Delete the entry in the form's Recordsource property.
2. Add a constant to your form that holds the SQL equivalent to the old Recordsource property.

```
Const strOldRecordSource As String = _
        "Select * From Customers"
```

3. Wherever your code uses the Filter property, replace that code with code that updates the Recordsource property with your constant and the string that you used to put in the Filter:

```
Me.Filter = strOldRecordSource & " Where " & _
        "…old text for the Filter property…"
```

If the original Recordsource already contained a Where clause, then you'll need to use this code:

```
Me.Filter = strOldRecordSource & " And " & _
        "…old text for the Filter property…"
```

4. If you're using the Where parameter in the DoCmd.OpenForm method, move that parameter's entry to the OpenForm's last parameter (the OpenArgs parameter). In the Form_Load event update the Recordsource:

```
Me.Filter = strOldRecordSource & " And " & _
        Me.OpenArgs
```

Alternatively, you could call me in to consult. My rates are very reasonable. In fact, call me in anyway—that way, you'll have someone to blame if this doesn't work out. ▲

DOWNLOAD **ACCEFF.ZIP at www.smartacessnewsletter.com**

Peter Vogel (MBA, MCSD) is the editor of *Smart Access* and a principal in PH&V Information Services. PH&V specializes in system design and development for systems that use Microsoft technologies. Peter has designed, built, and installed intranet and component-based systems for Bayer AG, Exxon, Christie Digital, and the Canadian Imperial Bank of Commerce. He's also the editor of Pinnacle's *XML Developer* newsletter and wrote *The Visual Basic Object and Component Handbook* (Prentice Hall, currently being revised for .NET). In addition to teaching for Learning Tree International, Peter wrote its Web application development, ASP.NET, and technical writing courses, along with being technical editor of its COM+ course. His articles have appeared in every major magazine devoted to VB-based development, can be found in the Microsoft Developer Network libraries, and will be included in Visual Studio .NET. Peter also presents at conferences around the world. peter.vogel@phvis.com.