vb123.com

Smart Access

Solutions for Microsoft® Access™ Developers

Simplifying Complex SQL

Peter Vogel

95 97 2000 2002 2003 DOWNLOAD

Access developers often find SQL bewildering: While simple things are easy in SQL, as you move up to more complicated problems, SQL statements can quickly become intimidating. Peter Vogel looks at some strategies for solving tough problems with SQL.

WHEN working with complicated problems, you often find yourself working with complicated SQL. One of the difficulties that developers have when writing SQL is the steep learning curve. While simple things are easy in SQL, once you move beyond retrieving groups of records things get difficult very quickly. SQL can become complicated very quickly.

To succeed with SQL, the most important thing you need to learn isn't new SQL commands but a new approach. Most Access developers use a "procedure-oriented" approach to problem solving: Do this, then do this, then do this. This approach works well when writing VBA code. Unfortunately, SQL is *not* procedural but is set-based. Many developers, when working with SQL, develop a procedural algorithm ("If I were doing this in VBA, I would...") and then try to convert that procedural algorithm to a set-based algorithm. This only complicates the problem. The first step in handling complicated SQL is to simplify the way that you think about the problems you want to solve with SQL.

A set-based approach to solving problems begins by generating all the rows that contain the solution and then removing those rows that aren't part of the solution (I'll call this second phase "winnowing the results"). Generating all possible rows is easy in SQL: If you join two tables without a Join or corresponding On clause (or a Where clause), you'll get every combination of every row in the two tables. That's what this SQL statement does:

```
Select Team.Name, Team_1.Name
  From Team, Team As Team_1
```

The classic example of the winnowing approach is generating a schedule

January 2006

Volume 14, Number 1

95 Applies to Access 95
97 Applies to Access 97
2000 Applies to Access 2000
2002 Applies to Access 2002
2003 Applies to Access 2003

DOWNLOAD Accompanying files available online at www.pinnaclepublishing.com

for teams to play each other in a league. Joining the table of teams to itself will generate a list where every team plays every other team. However, this list will also have each team playing itself. So, while the list contains the solution, it also contains rows that aren't part of the solution. To get to the solution, all that's necessary is to winnow the offending rows. In this case, that means removing those rows where the names of the two teams in a match are the same. This query implements that solution:

```
Select Team.Name, Team_1.Name
  From Team, Team As Team_1
Where Team.Name <> Team_1.Name
```

In this case, the Where clause that eliminates the offending rows can be moved into the From clause and be specified in an On clause (this is called a non-equijoin):

```
Select Team.Name, Team_1.Name
  From Team Inner Join Team As Team_1
    On Team.Name <> Team_1.Name
```

It's always risky to predict performance based on modifying SQL statements because so much depends on how the SQL optimizer in your database server processes requests. However, moving the test into the From clause should give faster processing.

Divide and conquer

One of the more convenient features of SQL is that the output of any SQL statement is a table that can then be used as input to another SQL statement (in fact, most SQL processors don't discriminate between queries and tables). As a result, many developers handle complicated SQL by breaking the problem down into several SQL statements and basing each query on an earlier statement. In addition to generating simple SQL at each step, this strategy also provides a kind of "SQL debugging." The results of each query can be reviewed, allowing the developer to see what is being accomplished at each step in the process.

For instance, this query generates some information based on calculations that use other fields in the query (I'll assume that this query is saved as qryMPG):

```
Select VehicleType, Miles, FuelConsumed,
         Miles/FuelConsumed As MPG
  From Vehicle_Performance
```

This query then calculates the average MPG for a specific vehicle type by treating the qryMPG query as its input:

```
Select VehicleType, Avg(MPG)
  From qryMPG
Where VehicleType = "SUV"
Group By VehicleType
```

Often developers are worried about the performance implications of implementing this process. The assumption is that the SQL processor will start by executing the query that's based only on actual tables, store the intermediary table somewhere (probably in memory), and then execute the queries that are based on that initial query. This process would not only be slow but would also consume tremendous amounts of resources in holding the intermediary tables (especially because the queries that run early in the process tend to produce lots of rows that queries later in the process winnow).

The good news here is that most SQL processors (including Jet) will actually combine the queries into a single uber-query and execute that. In other words, the actual query that's executed will look something like this:

```
SELECT Vehicle_Performance.VehicleType,
   Avg([Miles]/[FuelConsumed]) AS AverageOfMPG
  FROM Vehicle_Performance
WHERE (((Vehicle_Performance.VehicleType)="SUV"))
GROUP BY Vehicle_Performance.VehicleType
```

The key point here is that you don't have to worry about this—at least, not normally (see below for an unusual exception). If you've been using this approach, while you shouldn't stop worrying about performance, you don't need to feel concerned that you're being spectacularly inefficient, either. One caveat: Not all queries can be merged, and the SQL processor may end up running each query individually and storing the result.

If you look at the queries that you've been generating, you'll probably find that you're following the process that I recommended: The earlier queries in the process generate all the possible rows and the later queries winnow those results.

In fact, there are benefits in application maintenance by following this strategy. You can often build a very complex query that produces the base data required by several other queries. These other queries become much simpler to read and modify because much of their complexity has been removed to the base query. There's no free lunch, of course: There is some risk in that you've created a single point of failure in the base query. An unfortunate change to the base query can cause several other queries to fail.

Incorporating VBA

Another strategy that many developers follow is to use VBA functions in their SQL statements, something that only Jet allows. Developers typically have three reasons for calling VBA functions that they've written from SQL statements:

- To use native VBA functionality that isn't available in SQL.
- To implement procedural code.
- To make SQL statements easier to read (that is, instead of using hard-to-read IFF functions in-line with their SQL statements, VBA code can use the more easily read and debugged VBA If...Else...End If construct).

Continues on page 4

Simplifying Complex SQL...

Continued from page 2

Assuming that there's no other way to solve your problem other than using VBA, this can be an excellent strategy for Access developers who want to take advantage of their facility with procedural code. CalculateMPG in this SQL statement handles conversions between different units of measure (kilometers, miles, gallons, liters) in calculating miles per column:

```
Select VehicleType,
    CalculateMPG(Miles, DistanceUOM,
                 FuelConsumed, FuelUOM)
   From Vehicle_Performance
```

There are two standard problems that developers have when calling VBA functions from SQL. The first results from Jet's attempts to optimize query processing. This CountNumber function, for instance, is being used to number each line in the query:

```
  Select CountNumber, VehicleType,
       Miles/FuelConsumed As MPG
       From Vehicle_Performance
```

The code in CountNumber is very simple:

```
Function CountNumber() As Integer
Static Counter As Integer

  Counter = Counter + 1
  CountNumber = Counter

End Function
```

However, the result produced will look like this with the same result for CountNumber on each row:

CountNumber	VehicleType	MPG
1	SUV	20
1	SUV	22.5
1	Tractor	30

What's happened is that Jet has attempted to optimize processing of the query. Jet has decided that the CountNumber query only needs to be run once (that the function CountNumber's result is invariant from one row to another) and ran the function once, then repeated the result on each line. Jet makes this decision based on the parameters passed to the function: If the function doesn't accept any fields from the query, Jet assumes that the function only needs to be called once.

If your function does produce a different result every time it's called (that is, you've written a counting function or a random number generator), you'll need to pass a field to the function even if you don't need any field in the function. Rewriting the CountNumber function to produce the correct result would look like this:

```
Function CountNumber(SomeField As String) As Integer
```

The SQL statement would look like this:

```
Select CountNumber(VehicleType), VehicleType,
     Miles/FuelConsumed As MPG
     From Vehicle_Performance
```

The second problem results from the way that Jet handles the interaction between VBA and SQL. The problem is simple: The SQL processor/data retrieval functions don't interact with VBA. If you use a VBA function in your code, then Jet retrieves the records and only then, after the data is retrieved, processes the results. Whether or not this is a bad thing depends on where you use your VBA function.

If you use your VBA function in your Select clause (as in my previous example), Jet retrieves the record and applies the function. Still assuming that there's no other way to meet your goal other than to call a VBA function, there's no particular performance hit associated with calling a VBA function from your Select clause.

However, this is a case where knowing more SQL keywords can be helpful and knowing what's available on the database engine that you're using is critical. It's almost always true that a function that executes natively on the database engine will outperform any processing done back in your Access application. If the same goal can be met by both a command native to your database server and a VBA function, you'll be better off using the native command. Writing a stored procedure that will execute on the server, while it involves learning yet another language, and calling it from your SQL statement will also give better performance than calling a VBA function.

However, if you use your VBA function in the Where clause of your SQL statement, things turn ugly. Because the SQL processor can't handle the VBA function, Jet drops the Where clause. And not just part of the Where clause, either: Jet drops *all* of the Where clause. The result is that Jet retrieves all of the rows specified in the Join clause and, once the rows are returned to the Access application, winnows the results at the application by applying the Where clause to the returned result. This is the worst of all possible worlds: More records are returned than are necessary and extra processing is applied at the application.

It gets worse. Coupling the way that Jet handles VBA functions with the way that Jet merges queries can create a perfect storm of bad performance—and even generate queries that inexplicably stop running. My previous qryMPG query, for instance, calculated miles per gallon by dividing the Miles field by the FuelConsumed field. However, if either of those two fields is Null, the calculation will fail. So a reliable version of the query would look like this:

```
Select VehicleType, Miles, FuelConsumed,
        Miles/FuelConsumed As MPG
  From Vehicle_Performance
Where Miles Is Not Null
  And FuelConsumed Is Not Null
```

Now assume that a query built on top of this query

uses a VBA function in its Where clause:

```
Select VehicleType, MPG
   From qryMP
Where CheckType(VehicleType) = True
```

When the Jet processor combines these two queries, you'd think that this query is the result:

```
Select VehicleType,
       Miles/FuelConsumed MPG
  From Vehicle_Performance
Where Miles Is Not Null
  And FuelConsumed Is Not Null
  And CheckType(VehicleType) = True
```

However, Jet will strip out the Where clause because it contains a VBA function, leaving this query:

```
Select VehicleType,
       Miles/FuelConsumed MPG
  From Vehicle_Performance
```

Since rows with null values are no longer being excluded, the query will now die when doing the calculation as soon as a row with null values is retrieved. Assuming that there's no other way to winnow your rows other than to use the VBA function, there really is no solution to this problem other than to create a temporary table to hold the results of the first query and then base the second query on the temporary table.

Warnings

Warning: Just in case the warnings I placed around using VBA aren't sufficient, I'll repeat them here. You're always better off using native SQL commands or using your database engine's stored procedure language than processing with VBA back at the application.

And, as fond as I am of SQL, there's a whole range of problems that are better solved using procedural code. Any query that's positional in nature (find the highest value, find the top six values, find the second largest value) is not well suited for SQL unless you take advantage of vendor-specific extensions. At one point, I developed a humongous SQL statement to return the second highest value in a table. The query took five seconds to run. Executing a much simpler query from code that just returned a recordset of all the rows in the table and then, still in code, moving to the second row took less than a second.

You now have three tools for simplifying code. The most important tool is the winnowing process: Generate a recordset that contains the answer and then winnow out the records that don't belong. Breaking your queries down into more understandable queries that build on each other is one way of implementing that winnowing process. For many procedural programmers, incorporating VBA can further reduce the complexity

Continues on page 11

Simplifying Complex SQL...

Continued from page 5

of a SQL statement. And, of course, you can combine these techniques provided that you're careful about any VBA functions used in Where clauses.

Now, go forth and query. ▲

DOWNLOAD 601VOGEL.ZIP at www.pinnaclepublishing.com

Peter Vogel (MBA, MCSD) is the editor of *Smart Access* and a principal in PH&V Information Services. PH&V specializes in the design and development for systems that use Microsoft tools. Clients include Volvo, the Canadian Imperial Bank of Commerce, and Microsoft. He also wrote *Professional Custom Controls and Web Parts for ASP.NET 2.0* and *The Visual Basic Object and Component Handbook*. In addition to teaching for Learning Tree International, Peter wrote their ASP.NET and Technical Writing courses, along with being technical editor of their COM+ course. His articles have appeared in every major magazine devoted to development with Microsoft tools, can be found in the Microsoft Developer Network libraries, and are included in Visual Studio .NET and Office 2003 release packages. Peter also presents at conferences around the world..