Working SQL: Efficient SQL

Peter Vogel

In the latest edition of the "Working SQL" column, Peter Vogel looks at some of the general issues around creating SQL queries that run quickly.

THIS is a bell that I keep ringing: The key to great performance in a database application is using SQL. Every time you find yourself looping through a recordset and doing updates, you should be figuring out how to replace your code with an SQL statement. If there isn't an SQL statement that will do the job, take it as a sign that your database design is wrong and fix it so that you can replace your recordset-based processing with an SQL statement.

But, having said that, some SQL statements run more slowly than others. So how can you build SQL statements that will run as fast as possible? Related to this issue is what you can do to make all of your SQL queries run faster.

Optimizing the database

The second question—optimizing the database to support your SQL statements—is the easiest to answer, as the best practice is to follow good database maintenance. You'll want, for instance, to keep your indexes defragmented, move the log file on a separate drive from the database file, and perform other common best practices.

Many database management systems (DBMSs) offer ways to physically organize your data so that queries will run faster. For instance, where there's an index that your SQL statements use frequently (more on this later), the DBMS can cluster your tables on that index. Clustering in this sense refers to physically storing the records in order by the values in the index. This can be especially worthwhile if the index is used to sort the records into a specific order. If you have two tables that are frequently joined, many DBMSs will allow you to store related records from different tables in the same physical location on your hard disk.

There are caveats in all of these solutions. The first, of course, is that not all DBMSs offer these options. If you take advantage of them, you're locking yourself into a particular vendor. On the other hand, in my experience, changing databases isn't all that common an event, so why worry? The second caveat is that all of these solutions tend to contain the words "frequently" or "usually" in their descriptions. In other words, these solutions depend on how your current users use your current applications to access data for their current business environment. If any of these variables change, not only will these kinds of optimizations not improve the performance of your system, they'll actually slow down your applications.

For most applications, you're better off creating the most efficient SQL statement that you can and ignoring these physical options. One of the benefits of creating efficient SQL statements is that the principles for creating efficient SQL apply, with some variations, across all DBMSs. What works in one environment will port to others.

Rather than provide you with a laundry list of SQL "speed up" techniques, in the rest of this article I'll give you the understanding that you need to make your own decisions.

Understanding SQL

SQL is a non-procedural language. Unlike, for instance, Visual Basic, you don't say how you want the job done. Instead, you specify what you want as your result. The following SQL query says that I want the customer last name and total orders but only for customers in Canada, sorted by the customer's Last Name:

```
Select Customer.LastName, Count(Order.*)
From Customer Inner Join Order
  On Customer.CustomerNumber = Order.CustomerNumber
Where Customer.Country = 'Canada'
Group By Customer.LastName
Order By Customer.LastName
```

While SQL is a non-procedural language, in the end, the computer must execute a series of instructions that will retrieve the necessary data. So, when you consider an SQL statement, you should also consider the procedural code that will be required to retrieve the data that you've requested.

None of this is magic. Take my previous SQL statement, for instance. There are a limited number of ways that the data could be retrieved. Here's one possibility:

1. The customer table is read sequentially.
2. As each customer in Canada is found, the Orders table is accessed.
3. The Order table is read sequentially each time it's accessed.
4. When an order for the Customer currently being

processed is found in the Orders table, the Customer LastName is written to a new table.
5. When the end of the Customer table is reached, processing starts on the new table of last names.
6. The records are read out of the new table into yet another table, sorted in order by the customer last name so that the orders for each last name can be counted.
7. The final table is read, counting the number of records for each last name.

Written out this way, it seems remarkable that you get an answer back at all. One thing that you might notice is that the Order By clause at the end of the SQL statement is unnecessary. In order to generate the count of orders for each last name, a new table with the data sorted by last name is created—you're going to get the data in order by your Group By clause with or without the Order By clause.

I've assumed that I've correctly guessed how the data will be processed. The conversion from the non-procedural SQL to the procedural code that will retrieve the data is handled by the DBMS's query optimizer. This component of the DBMS is really the crown jewel of the vendor's product. Coupled with the way that the data is actually stored on the disk, the optimizer controls the SQL performance and features offered by the DBMS vendor. As a result, how the query optimizer works is among a DBMS vendor's most closely guarded secrets.

Still, you can make some educated guesses about how the optimizer works. To begin with, there are a limited number of ways that data can be accessed. In addition, the DBMS vendor may provide a series of hints or suggestions on how to best take advantage of the optimizer. Finally, many DBMS products will, on request, produce a plan of the steps that the optimizer will follow to execute when the query is processed. For this article, I'll just assume the procedural code I described earlier.

Indexes

Looking at the process that I outlined, you probably identified reading the customer table and the order table sequentially as inefficient processing. If your DBMS produces a plan for the query that confirms that the tables are being read sequentially, you might decide that applying an index will speed up your processing. After all, with an index on the Country field in the Customer table, you won't have to read every record in the Customer table—using the index only, the records for the Canadian customers will be retrieved. It's not that simple, unfortunately.

Let me begin by taking a simple case: There are no Canadian customers. The code generated by the query optimizer will work its way through the index to find the first Canadian entry. This will go relatively quickly (assuming that the index isn't badly fragmented) since most indexes are stored as B-Trees. B-Trees keep the number of accesses to the square root of the number of entries in the index. The number of records in the Customer table is almost irrelevant. For instance, if there are 3,000 Customer records but they're split between two countries, then there will be only two entries in the Country index. A single access to the index will retrieve either country record. On the other hand, if I assume that there are 40 countries in the index, a maximum of six accesses will determine that there are no Canadian customers.

Since most indexes are short and contain little data (in this case, possibly just the two-letter abbreviation for the country), indexes are often kept in memory by the DBMS. This makes accesses very fast. With no Canadian customers, an index on the Country field will give you excellent performance.

Now look at the other case: Every customer is in Canada. If the optimizer uses the index to retrieve the Customer records, the procedural code looks like this:

1. Access the index to get the next Country index record.
2. Get the address for the related Customer record.
3. Retrieve the Customer record.
4. Repeat.

It's likely that each record in the Country index points to a Customer record that's in a different block on the hard disk than the previous Customer record in the index. These uncoordinated accesses to the hard disk will have the disk drive heads leaping all over the platter.

What if all the customers are from Canada? In this case, sequential access becomes very attractive. Ideally, the disk drive head is dropped at the start of the Customer table and, as the platter revolves underneath, the Customer records are read in one smooth pass. Even if there's an odd customer that isn't from Canada, it's just a matter of discarding those records.

The typical situation is probably between the two extremes of no Canadian (very fast with an index) and no non-Canadian customers (very slow with an index). Most DBMSs have a cut-off point where, even if there's an index, it won't be used. The cut off point is frequently surprisingly low—often around 15-20 percent. If the query optimizer believes that it will retrieve more than 20 percent of the records in a table, an index won't be used even if it's present. Adding an index in these situations won't speed up your query but will slow down updates to the indexed fields.

As an example, putting an index on a column that contains only M for male and F for female is a waste of

time since any value will result in accessing 50 percent of the records. An exception would be if there were four times as many males as females (or vice versa), forcing the females to less than 20 percent of the records. The index would then be useful but only when selecting on females.

Combining data in an index can help matters, though. If the gender index also includes the Country field, the number of records for any particular combination (that is, females in Canada) might well drop below 20 percent. If gender is frequently searched with country, this index could be very useful. It does, however, raise the question of what order to put the data in: Country-Gender or Gender-Country?

Again, thinking about how processing must take place can be helpful. A telephone book is a good example of a multiple-key index: Last Name-First Name. Finding someone by first name in a phone book is a nightmare, while searching by last name is much easier. Creating an index on Gender-Country will divide the index into two equal halves—M on one side, F on the other. On the other hand, putting the Country before the Gender will divide the index into 40 groups (I'm still assuming 40 countries in my database). The smaller groups can, potentially, be searched faster than the two larger groups.

Subqueries

A frequently used technique in SQL is a subquery. The same kind of analysis can help you understand how best to use them.

A subquery combines two queries into one. Attempting to find all the customers with an order amount larger than the average number of orders would use a subquery like this:

```
Select Customer.LastName
From Customer Inner
Where Customer.OrderAmount >
  (Select Average(OrderAmount)
       From Customer)
```

Again, there's no magic here: The DBMS will have to run both queries. The subquery will be run first since its result is required to process the first query (at least, in this example). In addition, the result of running the subquery will have to be stored somewhere while the main query is processed. Since the result is a single value, this isn't hard to do and doesn't require a lot of resources. Comparing the OrderAmount to a single value stored in memory will also be quite fast. However, calculating the average OrderAmount requires processing all the records in the Customer table, which could be time-consuming depending on the number of records in the table.

A more complicated query finds all the customers with orders greater than any order made by a customer in Canada:

```
Select Customer.LastName
From Customer Inner
Where Customer.OrderAmount >
  (Select Customer.OrderAmount
       From Customer
       Where Country = 'Customer')
```

The subquery will probably still be run first, but, in this case, the result of the subquery is a list—potentially quite a long one. Not only is holding this list in memory potentially resource-intensive, but comparing each record in the main query to each entry in the subquery result will be slow. This query can be sped up my converting it to an equivalent of the previous query, which returned a single result:

```
Select Customer.LastName
From Customer Inner
Where Customer.OrderAmount >
  (Select Max(Customer.OrderAmount)
       From Customer
       Where Country = 'Customer')
```

In this case, an index on the OrderAmount might make the query run considerably faster—a single access to the index could retrieve the highest OrderAmount. However, you'd have to trade this off against the cost of maintaining an index on what is probably a highly volatile field.

Continues on page 23

Working SQL...

Continued from page 9

Another subquery operation is Exists. The Exists operator simply checks to see whether a corresponding record exists, using the subquery. This query finds all the customers with a last name that matches an employee in the shipping department:

```
Select Customer.LastName
From Customer Inner
Where Customer.LastName Exists
  (Select LastName
       From Employee
       Where Department = 'Shipping')
```

With Exists, the optimizers might run the main query first. As each record is retrieved, the subquery will be run to see whether a corresponding record exists. If the subquery returns nothing, the main query can drop the record retrieved by the main query. Alternatively, the optimizer might run the subquery first and build a list of LastNames to use with the main query.

Potentially, an index on Department and LastName could speed up this query. Given the existence of the index, the optimizer could determine whether the record exists just by checking the index without having to access the Employee table itself. Since the number of last names probably exceeds the number of departments, the order of the index should be LastName-Department. Notice that, in this case, the index is useful even if it violates the 20-percent rule since the optimizer is simply using it to determine whether a record exists.

As I've demonstrated in this article, working through the process that the optimizer must follow to retrieve your data will help you to understand how efficient your queries will be. Armed with this information, you can decide the best way to frame your own SQL commands. ▲

Peter Vogel (MBA, MCSD) is 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 *Smart Access* and *XML Developer* newsletters 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.

Do-it-yourself Queries...

Continued from page 15

Also, sort functionality was built in behind the labels of each column. To sort ascending, the user simply single-clicks on the appropriate heading label. To sort descending, the user should double-click on the appropriate heading label. To restore to the original view, the user should click on Restore Original. The code that will perform this action is shown here and is representative of both ascending and descending sorts on all the heading labels:

```
Private Sub lblName_Click()
On Error Resume Next

Me.txtName.SetFocus
DoCmd.RunCommand acCmdSortAscending
Me.cmdRemoveFilterSort.Visible = True

End Sub

Private Sub lblName_DblClick(Cancel As Integer)
On Error Resume Next

Me.txtName.SetFocus
DoCmd.RunCommand acCmdSortDescending
Me.cmdRemoveFilterSort.Visible = True

End Sub
```

The button cmdRemoveFilterSort contains the code that will remove the sort and any filter applied to the form. The code is shown in here:

```
Private Sub cmdRemoveFilterSort_Click()
On Error Resume Next

    DoCmd.RunCommand acCmdRemoveFilterSort
    Me.Filter = ""

End Sub
```

Wrapping it up...

There's other functionality built into the form behind the Export and Print buttons. The Print button contains the following code. If the filter is turned on, then only that data is sent out to the report. Otherwise, the entire view is sent to the report. Once again, the application uses one type of object for data presentation. The record source for the report turns out to be the same as the record source for the form at report generation.

```
Private Sub cmdReports_Click()
On Error Resume Next

If Me.Filter = "" Then
  DoCmd.OpenReport "rptViewSelectionList", acViewPreview
Else
  DoCmd.OpenReport "rptViewSelectionList", _
    acViewPreview, , Me.Filter
End If

End Sub
```

The Export button performs a similar function. The