The Return of Working SQL: What's the Plan?

Peter Vogel

Peter adds another installment in the Working SQL series with a look at a fundamental topic in relational database theory. Just to demonstrate that it actually is useful information, Peter uses it to analyze a Northwind query.

WHEN I was trying to come up with a name for this series of articles about SQL, one of the titles I considered was "SQL for Mortals." I've rubbed shoulders with some real SQL wizards, and they've always seemed to operate at a level that had little to do with my reality of getting applications out the door. What I wanted to do in this series was talk about some aspects of SQL that would be useful to real developers.

One of the terms that the wizards bandied about often was "relational algebra." At the time, I ignored the subject because I figured it was too esoteric to make any difference to me. I was wrong. I discovered that by understanding the relational algebra and its relationship to SQL, I could get a real leg up on understanding why some of my queries really flew and others didn't.

Speeding up queries by making changes to the database (typically by adding indexes) is an art that few have mastered. The process that most developers follow is to apply an index and then run the query to see if it's any faster. If you're being paid by the hour, this is an excellent method, though frustrating when nothing seems to help.

If you're tired of that method, the relational algebra can provide you with a way to see what choices Access is making when it decides how to execute your query. With that in hand, you can make some intelligent decisions about how to speed your query up.

One of the things that I didn't realize was that SQL wasn't developed by the father of the relational database, Dr. E. F. Codd. Instead, a group of researchers in the same lab created a language they initially referred to as SEQUEL (for Structured English QUEry Language). Not surprisingly, SQL doesn't conform completely to relational theory. In this article, I'll also map the relational algebra back to the SQL commands you know and love. To demonstrate all this material, I'll use the records in **Table 1** and **Table 2** as examples.

The algebra

Remember, first, that an "algebra" is just a language. It consists of items to be manipulated (in relational theory, these are always tables), and operators that will manipulate the items. The *relational* algebra has seven operators with names just similar enough to SQL statements to be annoying. They are:

- Select
- Project
- Join
- Union
- Difference
- Intersect
- Divide

Table 1. The sales table.

Name	Country	Amount	SalesRep	Date
First Outfitters	England	$10,000	D. Hewitt	97/10
Master Crafters	Netherlands	$5,000	P. Wise	98/01
East-Side Mappers	England	$250,000	W. Pritchard	96/05
Robert-Jameson	England	$150,000	M. Rothko	97/05
Orlando's	France	$25,000	B. Garner	96/12
Shoolside Supplies	England	$15,000	J. Pollock	97/05

Table 2. The employee table.

Name
W. Pritchard
P. Wise
D. Hewitt
B. Garner

In this article, I'm going to restrict myself to talking about the three operators that are part of the SQL SELECT command: Select, Project, and Join.

Why do I say that these operators are similar enough to SQL to be annoying? The best example is the relational operator Select, which has very little to do with the SQL SELECT command. Some writers have recommended replacing Select with Restrict, which, as I'll show later on, is the term that Access uses internally.

The Select operator in relational algebra is used with criteria to limit the rows returned from a table. The syntax for using the Select operator would look something like this:

```
T1 = Select "Country = 'England'" (Sales)

D = PI * R^2)
```

In the previous example, T1 is the table that results from applying the Select operator, with its criteria, to the table Sales. In this case, T1 should have four rows in it. I've modified the syntax that's used in most discussions of the relational algebra to make it easier to read for non-wizards (like me).

You're probably used to using the SELECT command in SQL to specify which columns you want to see in your query. This is actually the province of the Project operator. The Project operator restricts the columns to be displayed in the table. The syntax for using it would look something like this:

```
T1 = Project Name (Sales)
```

In this example, T1 is the table that results from applying the Project operator, with its field list, to the table Sales. In this case, T1 should have six rows, but only one column—Name.

How does this apply to SQL? Take the following SQL statement:

```
Select Name
From Sales
Where Country = 'England'
```

In the relational algebra, it would break down into two operations:

```
T1 = Select "Country = 'England'" (Sales)
T2 = Project Name (T1)
```

The first operation produces table T1, which has only the four rows for England in it. In the second operation, the table from the first operation is restricted to just one column.

The Project could have been done first, at the cost of an additional operation:

```
T1 = Project Name, Country (Sales)
T2 = Select "Country = 'England'" (T1)
T3 = Project  Name (T2)
```

Why would you do this? If the Sales table had many columns, it might be faster to reduce it to two columns before doing the Select. Presumably, this would give you less data to work with and therefore be able to deliver the results faster. When you run your query, the decision about which method to follow is one of the responsibilities of Access's query optimizer.

Joining up

The decision on which operations to perform first becomes more important when you add the Join operator. The SQL JOIN keyword acts like a combination of the relational Product and Select operators. The Product operator, given two tables, produces a Cartesian product of them. That is to say, Product produces a table containing every combination of every record in the two tables (see the Working SQL column in the May 1997 issue, "SQL Without Joins," for more on Cartesian products). Since it's unlikely that this is what you want, you'll almost always follow a Product with a Select to limit the rows.

With the Product operator tamed, you can tackle more complex SQL statements. For instance, the following SQL command:

```
Select *
From Sales Join Employees
On SalesRep = Name
```

would be expressed in the relational algebra like this:

```
T1 = Product (Sales, Employee)
T2 = Select "SalesRep=Name" (T1)
```

In this case, T1 is a table of 24 rows consisting of all six rows in the Sales table joined in every combination with the four rows in the Employee table. Table T2 is the result of restricting the table to those rows where the SalesRep field matches the Name field.

Since a Product almost always involves a Select, the Join operator is included in the relational algebra as well. Join combines the Product and Select operators into one operation:

```
T1 = Join "SalesRep=Name"  (Sales, Employee)
```

But what if the query was slightly more complicated? This query not only has a restriction in the Join, it also has a Where clause:

```
Select Name, Salary
From Sales Join Employees
On SalesRep = Name
Where Country = 'England'
```

Should the data be retrieved by doing the Join first and the Selects afterward? I'll call this Plan 1:

```
T1 = Join "Salesrep=Name"  (Sales, Employee)
T2 = Select "Country='England'" (T1)
T3 = Project Name, Salary (T2)
```

Or, would it be it faster to narrow down the Sales table before doing the Join? I'll call this Plan 2:

```
T1 = Select "Country='England'" (Sales)
T2 = Join "Salesrep=Name" (T1, Employee)
T3 = Project Name, Salary (T2)
```

The answer is that it depends. The table that results from joining the Sales and Employee table is very small—only 24 records. As a result, creating it and then selecting the two matching records out of it is pretty trivial. If the Sales table was very large, though, and the number of sales in England small, it might be smarter to follow Plan 2 and create the table of sales to England first.

Conventional wisdom suggests that the appropriate thing to do, for Plan 1, is to create indexes on SalesRep and Name to speed the Join of the two tables in the first step of the plan. For Plan 2, an index on the Country column should speed its first step. On the other hand, there's nothing to be gained in Plan 2 by creating an index on SalesRep, since that field isn't accessed until the temporary (unindexed) table T2 is created.

However, before creating any indexes, it's important to recognize how inexpensive sequential reads of a table are.

Doing it sequentially

Reading an index to find a single record is cheaper than reading the whole table, if the table is large. However, reading an index isn't free. The index records are stored on disk like any other table and must be read into memory. Several reads might be required to find the entry in the index for the record you want. And then, when all that's done, another read must be done to get the actual record that was requested. Obviously, a computer with lots of memory is a big help here as it can cache the indexes in memory and so avoid having to read them from disk.

Straight sequential reads from disk aren't all that expensive. Since records are typically blocked together, one read of the disk can load many records into memory. If the table isn't fragmented, movement of the disk drive's read/write head is minimized as blocks are read one right after another. Shifting from index to table and back again, on the other hand, can mean lots of disk head movement. Typically, if you'll be handling more than a small number of the records in the table, it's faster to read a file sequentially than to access records through the index. On some systems, if you're accessing more than 20 percent of the records, you're better off to read the table sequentially.

So, if you go with Plan 2 and over 20 percent of your sales are to England, it's not worth your while to create the index on country. But, you say, I won't always be asking about England. If this is true (and it might not be), then you need to look at the distribution of the data you'll be pulling from the table. If the majority of your data access will be for records that make up more than 20 percent of your table, you have to ask if the cost of the index justifies the benefits. Sure, once a month, you ask about sales to Mauritania—but do you want to optimize your database to be responsive to that single question?

To make it more interesting, the creation of indexes affects the optimizer's decision on which plan to use. Unless the database manager keeps statistics on the distribution of data in the column, if you create an index on a column, the optimizer will assume that the records you want are less than 20 percent of the total. As a result, the optimizer will create a query that uses the index. This is a situation where creating an index might actually slow down processing.

As another example, there's probably not much benefit in creating an index on the Gender column of a table, since selecting it will always give you about 50 percent of the records.

Showing the plan

Access 95 and Access 97 provide a way to find out what the optimizer is doing: Turn on ShowPlan (see the accompanying sidebar "Showing the Plan") and force a re-plan of your query by making some change to it. Access will write out the plan for your review.

Here's the plan for the "Employee Sales by Country" query from the Northwind database (see **Figure 1** for the query layout):

```
- Inputs to Query -
Table 'Employees'
   Using index 'PrimaryKey'
   Having Indexes:
   PrimaryKey 9 entries, 1 page, 9 values
     which has 1 column, fixed, unique, clustered and/or
```

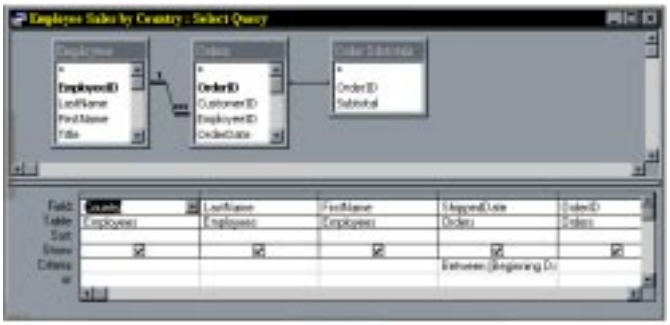

Figure 1. The Employee Sales by Country query from Northwind.mdb.

```
counter, primary-key, no-nulls
   PostalCode 9 entries, 1 page, 9 values
     which has 1 column, fixed
   LastName 9 entries, 1 page, 9 values
     which has 1 column, fixed
Table 'Orders'
Table 'Order Details'
- End inputs to Query -

01) Restrict rows of table Orders
     using rushmore
     for expression "Orders.ShippedDate Between
[Beginning Date] And [Ending Date]"
02) Group table 'Order Details'
03) Inner Join result of '01)' to result of '02)'
     using temporary index
     join expression "Orders.OrderID=[Order
Subtotals].OrderID"
04) Inner Join result of '03)' to table 'Employees'
     using index 'Employees!PrimaryKey'
     join expression
"Orders.EmployeeID=Employees.EmployeeID"
```

The first part of the output is the list of inputs to the query. This part of the ShowPlan report is only available in Access 97. Notice that Access does keep track of the number of different values in the index, presumably to decide if the index is worth using. Be aware that these statistics are generated when the database is compacted, so they can be out of date.

The second part of the output is the execution plan itself. You can see that the first thing Access does is a Restrict (or Select) on the Orders table, using the ShippedDate column. The notation "with rushmore" indicates (among other things) that the records will be selected based on the data in the index. As a result, Access won't have to read in the actual record to determine if it belongs in the query.

Step 2, "Group table 'Order Details'," refers to the execution of the Order Details query that's used in this query. In step 3, we see a Join between the temporary table generated in step 1 and the temporary table in step 2. Since these are both temporary tables, the indexes on the Order table can't be used (which isn't to say that Access doesn't use the Order table indexes to help build the temporary indexes it does use). Finally, in step 4, a Join is done between Employees and the temporary table created in step 3. Since the Employees table is not a temporary table, the index on the Primary key can be used.

Given this information, if this was the only query in the database, you might consider dropping the key on OrderId because it should have no effect on the

Continues on page 24

Showing the Plan

THE ability to show the plan that a query will follow is an undocumented feature of the Jet engine. If you want to start gathering the plans for the queries you create, you need to add a string value called JETSHOWPLAN to the key HKEY_LOCAL_MACHINE\SOFTWARE\MICROSOFT\Jet\3.x\Engines\Debug key in your registry. JETSHOWPLAN should have the value "ON." The procedure to follow is:

1. Run the RegEdit program.
2. Click on the plus sign by HKEY_LOCAL_MACHINE.
3. Scroll down until you find "Microsoft" and click on the plus sign beside it.
4. Scroll down again until you find "Jet" and click on its plus sign.
5. Repeat for 3.0 (for Access 95) or 3.5 (for Access 97).
6. Click on the plus sign beside Engines.
7. From the Edit menu, pick New | Key.
8. Enter "Debug" (without the quotes) and click on the OK button.
9. From the Edit menu, pick New | String Value.
10. Enter "JETSHOWPLAN" (all in uppercase letters and without the quotes) and click on the OK button.
11. Now that you've added JETSHOWPLAN to the registry, double-click on it, enter "ON" (without the quotes), and click on the OK button.
12. Shut down RegEdit.

Once you've done that, every time you change a query, Access will append the plan for the query to a file called ShowPlan.Out. This file is put in the current default directory, so you might end up with several copies of it. Also, the name of the query isn't inserted into the file, so you'll need to keep track of which plan is for which query.

You probably won't want to keep this turned on all the time. To stop appending query plans, run RegEdit again and set JETSHOWPLAN to "OFF."

The Return of Working SQL . . .

Continued from page 15

performance of the query. To test this out, I added a new primary key to the Orders table and dropped the original primary key index. A few quick-and-dirty tests showed that adding or removing the OrderId index seemed to have no impact on the query's performance. Had I been trying to optimize the query without this information, creating the OrderId index would have been the first thing I would have tried. Armed with the algebra and ShowPlan, I now know better.

This has only been a brief introduction to relational algebra. However, with this information and the ShowPlan output, you have a powerful tool for improving your application's performance. And, the SQL wizards will stop kicking sand in your face. ▲

Peter Vogel is the editor of *Smart Access* and a principal in PH&V Information Services (a technology management company). Peter also teaches for Learning Tree International and is developing course material for them on developing Web-based applications. peter.vogel@odyssey.on.ca.

Downloads — December Subscriber Downloads

- **SPACE.EXE**—The sample database to go with David Saville's article on working with spatially oriented data. All of the routines from David's article are here with a slick front end for you to test out the routines. David's warning about using any routine with floating point numbers still applies: Make sure the level of accuracy used in these routines matches the level you require.

- **SORTFORM.EXE**—This database includes Chris Weber's code for providing a flexible sorting scheme for your forms. Chris makes use of Access 95's new form properties to let you provide your users with an Explorer-like interface for controlling the order of their displayed data. Also included is the bolCheckIndex routine from the article's sidebar, which checks to see whether the field passed to it is the first field in one of the table's indexes.

- **SYSCROSS.EXE**—Duane Hookom's database for displaying system information about your application. This routine makes it considerably easier to view fields used in multiple tables by converting the information about them into a crosstab.

Pinnacle Publishing, Inc.
1503 Johnson Ferry Road
Suite 100
Marietta, GA 30062
http://www.pinpub.com

PINNACLE PUBLISHING, INC.

BULK RATE
US POSTAGE PAID
ATLANTA, GA
PERMIT NO. 4453