Exploit Jet SQL

Sure you can keep using DAO. But you're better off with Jet SQL—once you get the hang of it.

by Andrew J. Brust

Jet's version of SQL strays far afield from "standard" SQL in places, and I've found that few people take the time to master Jet SQL. Its annoying quirks still blindside VB programmers, while its useful extensions go unexplored. Most of us would rather write a query in a familiar SQL dialect, then tweak it until Jet stops tossing error messages at us. Eventually we'll get that inner join syntax right. Right?

Still, you ought to learn Jet SQL. That time you spend debugging queries, or designing them visually in Access and copying the generated SQL, could be better spent doing real programming. And what about upsizing? You may need to convert your Jet SQL queries to your server's SQL to implement passthrough queries. If you don't remember exactly what that strange Jet SQL meant, your translation task becomes even more arduous. Further, for certain data definition tasks, such as creating and maintaining tables and indexes, SQL can provide a more efficient and terse coding vehicle than DAO provides. Basically, whether you're using VB3 or VB4, 16- or 32-bit, getting a firm grip on Jet's SQL pays off.

If you now know standard SQL pretty well, adding Jet SQL's unique aspects to your repertoire doesn't represent an overwhelming learning curve. For example, the GROUP BY, HAVING, and ORDER BY clauses, and subqueries in general, work in the same manner either way—unlike the WITH OWNERACCESS OPTION, PARAMETERS, TRANSFORM, and PIVOT clauses, which bear discussion.

In addition, some of Jet's standard SQL features get underutilized because users haven't ventured beyond the procedural DAO code they already know. So we should also take a look at the SQL Data Definition Language (DDL) reserved words CREATE, DROP, and ALTER. These provide a great way to create and modify the structure of tables, indexes, and referential integrity relationships. They can often perform these tasks with much less code than you'd need to create and define TableDef, Field, and Index objects and modify their corresponding collections.

For the sake of clarity I'll present all SQL commands and keywords in CAPS, although Jet's SQL interpreter doesn't care about case. Also, I've terminated all SQL queries with a semicolon, in keeping with the formal, preferred syntax, although Jet SQL doesn't care about this either.

I recommend starting your study of Jet SQL's unique outlook on life with its most prominent syntax deviation, the JOIN clause. Most SQL dialects (including SQL Server's Transact SQL) make you express joins as logical expressions in a query's WHERE clause. But Jet allows for a JOIN subclause in the query's FROM list. A SQL Server inner join query:

```
SELECT tblCustomers.*, tblOrders.*
FROM tblCustomers, tblOrders
WHERE tblCustomers.iCustId =
   tblOrders.iCustId
```

becomes:

```
SELECT tblCustomers.*, tblOrders.*
FROM tblCustomers INNER JOIN tblOrders
ON tblCustomers.iCustId =
   tblOrders.iCustId;
```

in Jet SQL. You can use similar syntax for outer joins. Just substitute LEFT JOIN or RIGHT JOIN in place of INNER JOIN. Note that in certain cases you can nest JOIN clauses in Jet SQL (for example, table 1 inner-joins to table 2, which inner-joins to table 3). By the way, Jet does allow you to express inner joins in the WHERE clause as you would in standard SQL, but the INNER JOIN syntax is preferred. Outer joins must be expressed with the LEFT JOIN or RIGHT JOIN syntax. Interestingly, Jet SQL's JOIN syntax is close to that of ANSI-92 SQL, although it seems different from most widely used SQL dialects.

SELECTIVE EVOLUTION

Jet's SELECT command also deviates from garden variety SQLs. Look at the square bracketed components in the schematic syntax:

```
SELECT [predicate] { * | table.* |
   [table.]field1 [, [table.]field2.[,
   ...]]}
[AS alias1 [, alias2 [, ...]]]
FROM tableexpression [, ...] [IN
   externaldatabase]
[WHERE... ]
[GROUP BY... ]
[HAVING... ]
[ORDER BY... ]
[WITH OWNERACCESS OPTION]
```

Our first component, the predicate, defines the records in your query, if any, that will be filtered out of the result set that Jet gives back to you. Jet predicates consist of ALL, DISTINCT, DISTINCTROW, and TOP *n* [PERCENT]. ALL returns all records that meet the criteria outlined in the rest of your query. It is

Andrew J. Brust is president of Progressive Systems Consulting Inc., a New York City-based firm specializing in the development of, and developer training in, client/server and other custom business applications. Reach Andrew on the Internet at abrust@progsys.com or on CompuServe at 70274,1746.

implied—supplying the ALL predicate is like supplying no predicate at all.

Distinguishing between the next two predicate elements, DISTINCT and DISTINCTROW, has confused me more than any other element of Jet SQL. For example, take a simple database consisting of tables tblCustomers and tblOrders, with this structure and (very carefully chosen) data:

tblCustomers:

iCustId	cFirstName	cLastName
1	Harry	Smith
2	Harry	Jones

tblOrders:

iCustId	iOrderId	iAmount	cQuarter
1	1	50	Q195
1	2	50	Q295
1	3	75	Q295
2	1	44	Q194
2	2	55	Q395

Now make three queries and consider their result sets:

```
SELECT tblCustomers.cFirstName
FROM tblCustomers INNER JOIN tblOrders
ON tblCustomers.iCustId =
   tblOrders.iCustId;
```

cFirstName
Harry
Harry
Harry
Harry
Harry

```
SELECT DISTINCTROW
   tblCustomers.cFirstName
FROM tblCustomers INNER JOIN tblOrders
ON tblCustomers.iCustId =
   tblOrders.iCustId;
```

cFirstName
Harry
Harry

```
SELECT DISTINCT tblCustomers.cFirstName
FROM tblCustomers INNER JOIN tblOrders
ON tblCustomers.iCustId =
   tblOrders.iCustId;
```

cFirstName
Harry

All three queries return rows containing "Harry" as the only result set field value, but the number of rows returned varies. The first query performs a join across tblOrders and returns five rows (one for each order). The second query returns two rows because with DISTINCTROW Jet returns *exactly* one result row for each row in the table where fields are actually being selected from (tblCustomers), and which the INNER JOIN clause does not filter out. In the third query, the DISTINCT predicate filters out all "visibly" duplicate rows, returning only one row in the result set.

Note that DISTINCTROW only works in queries that list more than one table in the FROM clause and where at least one of those tables is not represented in the SELECT list. In queries that don't meet these criteria, Jet SQL permits the DISTINCTROW predicate but ignores it. Given the obscure context in which DISTINCTROW has any effect, I wonder why Microsoft chose to place DISTINCTROW in Access queries by default. One more comment on these two predicates: in many cases DISTINCT and DISTINCTROW return the same result data, but DISTINCT always forces the return of a Jet snapshot, while DISTINCTROW avoids that limitation.

The last predicate, TOP, lets you select the records with the first n values of the ORDER BY expression. Alternately, you can select the records with ORDER BY expression values fitting in the first nth percentile. For example, you can select the 10 employees with the highest salaries and salaries composing the top 10 percent of all salaries, respectively. You do so by selecting the highest salaries first (using the DESC reserved word in the ORDER BY clause) so that the first 10 records will be the highest salaries:

```
SELECT TOP 10 [cLastName], [cFirstName],
   [lSalary]
FROM tblEmployees
ORDER BY lSalary DESC;

SELECT TOP 10 PERCENT [cLastName],
   [cFirstName], [lSalary]
FROM tblEmployees
ORDER BY lSalary DESC;
```

You're not yet done with SELECT's deviant tendencies. You need to consider the AS and IN keywords, which are not to be confused with the IN operator used in the WHERE clause. AS lets you define field names for both calculated and even uncalculated columns. Other SQL syntaxes usually do this by preceding the expression with an alias name and an "equals" symbol. The standard SQL query:

```
SELECT cFullName = cLastName + ', ' +
   cFirstName
FROM tblEmployees;
```

becomes:

```
SELECT cLastName & ', ' & cFirstName AS
   cFullName
FROM tblEmployees;
```

in Jet SQL. The IN Keyword lets us include tables in our SELECT list that exist in other databases, whether they are MDBs, ISAMs (xBASE, Paradox, Btrieve, and others), or ODBC databases. With one keyword Jet lets you seize the power of its heterogeneous joins features. Don't miss the subtle sleight of hand—you can write heterogeneous join queries dynami-

cally that don't require the external tables to be attached tables in the MDB. Also, you never have to do an OpenDatabase on the external data. Jet will do one implicitly during the CreateDynaset or CreateSnapshot method call. For example, you can do heterogeneous joins between tblEmployees and both a FoxPro table or an ODBC table:

```
SELECT [CustomerID]
FROM Customer IN "C:\FOXPRO\DATA\SALES"
   "FoxPro 2.0;"
WHERE CustomerID Like "A*";

SELECT [CustomerID]
FROM Customer IN ""
   "ODBC;DSN=SQLS;UID=sa;PWD="
WHERE CustomerID Like "A*";
```

Of course, Jet lets you SELECT more than just the fields in the tables: you can use its built-in aggregate mathematical functions or your own expressions in the SELECT list. Aggregate functions are the mathematical functions, defined by Jet SQL itself, that you can use in calculated fields.

YOU NEVER HAVE TO DO AN OPENDATABASE ON THE EXTERNAL DATA.

Jet doesn't have many aggregate functions, and fortunately their names pretty well describe them: SUM, AVG, COUNT, MIN, MAX, STDEV, and VAR. Two more, the population functions STDEVP (population standard deviation), and VARP (population variance), are nonstandard SQL aggregates provided by Jet.

In addition to Jet SQL's functions, you can use any VB3/Access Basic/VBA function in calculated fields. This gives Jet SQL lots of horsepower. Additionally, when running your query in Access, any functions you create and store in MDB module collections become valid functions in calculated fields. However, you can't run queries referencing these user-defined functions from VB.

And as soon as you use a nonstandard aggregate, a Basic function, user-defined function, or any other Jet-specific SQL feature in your query, you make that query unintelligible to an ODBC database engine. If you run Jet-specific queries against an ODBC database, Jet must send a portion of your query to the server, then evaluate the intermediate result set and calculate the results of your non-SQL calculated fields locally.

QUERYING MINDS WANT TO KNOW

Jet SQL's JOIN and SELECT commands contain the core of this SQL flavor's idiosyncrasies. But wait, there's more—especially regarding queries and Jet MDB database maintenance.

Queries can be challenging on secure databases. Perhaps you have a table that only certain users can read, but you'd like to create a query on that table and let all users access the query and see its result set. They can as long as the query contains Jet SQL's WITH OWNERACCESS OPTION keywords.

You've probably used Access to design parameter queries, but did you know you can create these objects from Visual Basic? Access lets you implicitly create parameters by using field names that don't exist in your database And you can specify these parameters' data types by choosing Query/Parameters... from the Access main menu. However, this dialog box is just a front end to Jet's PARAMETERS clause table. The clause simply needs to list each parameter and its data type, with multiple parameters separated by commas, and end with a semicolon.

To see how this works in practice, take the first query from the discussion of predicates:

```
SELECT tblCustomers.cFirstName
FROM tblCustomers INNER JOIN tblOrders
ON tblCustomers.iCustId =
   tblOrders.iCustId;
```

and tweak it to add a parameter text field—prmcOperator—to accompany the other fields in the result set:

```
PARAMETERS prmcOperator TEXT;
SELECT prmcOperator,
   tblCustomers.cFirstName,
   tblCustomers.cLastName
FROM tblCustomers INNER JOIN tblOrders
ON tblCustomers.iCustId =
   tblOrders.iCustId;
```

You can use parameters in a query's WHERE clause as well as its SELECT list (see Table 1).

You may see crosstab queries as just another Access "user" tool, but they are actually useful and easy to implement from VB. Crosstab queries result from creating regular GROUP BY queries and adding TRANSFORM and PIVOT clauses to them. Without crosstabbing, a conventional GROUP BY query produces rather vanilla-flavored results:

```
SELECT First(tblCustomers.cFirstName) AS
FirstOfcFirstName,
   First(tblCustomers.cLastName) AS FirstOfcLastName,
   tblOrders.cQuarter,
   Sum(tblOrders.iAmount) AS SumOfiAmount
FROM tblCustomers INNER JOIN tblOrders
ON tblCustomers.iCustId = tblOrders.iCustId
GROUP BY tblCustomers.iCustId, tblOrders.cQuarter;
```

FirstOfcFirstName	FirstOfcLastName	cQuarter	SumOfiAmount
Harry	Smith	Q195	50
Harry	Smith	Q295	125
Harry	Jones	Q194	44
Harry	Jones	Q394	55

You get a summary view of the total purchases made by each customer in each quarter stored in the database. That's fine, but you probably want to see the information presented this way instead:

FirstOfcFirstName	FirstOfcLastName	Q194	Q195	Q295	Q3
Harry	Smith		50	125	
Harry	Jones	44			55

A superior result set like this takes only a little more effort in your Jet SQL query:

```
TRANSFORM Sum(tblOrders.iAmount) AS
SumOfiAmount
SELECT First(tblCustomers.cFirstName)
AS FirstOfcFirstName,
First(tblCustomers.cLastName) AS
FirstOfcLastName
FROM tblCustomers INNER JOIN tblOrders
ON
   tblCustomers.iCustId =
tblOrders.iCustId
GROUP BY tblCustomers.iCustId
PIVOT tblOrders.cQuarter;
```

Notice that I removed the Sum(tblOrders.iAmount) AS SumOfiAmount expression from the SELECT list and placed it in the TRANSFORM clause to make it into the value data used in the crosstab. Then I pulled tblOrders.cQuarter from the SELECT list and the GROUP BY clause and placed it in the PIVOT clause to specify its results to be used as the column headers in the crosstab. In all other respects the two queries match. Comparing them should help you get the hang of crosstab queries.

Access Combo Box Option	Jet SQL DDL data type reserved word
Text	TEXT
Memo	LONGTEXT
Number: Byte	BYTE
Number: Integer	SHORT
Number: Long Integer	LONG
Number: Double	DOUBLE
Number: Single	SINGLE
Date/Time	DATETIME
Currency	CURRENCY
Counter	COUNTER
Yes/No	BIT
OLE Object	LONGBINARY
Value (Variant data — for parameter queries only)	VALUE

TABLE 1 ***Choose Your Weapon.*** *If you're writing a SELECT query using the PARAMETERS clause or a DDL query using the CREATE or ALTER TABLE commands, you'll need to specify data types using Jet SQL reserved words. Here's a list of those reserved words and how they correspond to the more familiar combo box choices in the Access Table Design and Query Parameter dialogs.*

MAINTENANCE ISSUES

Now that you can create great Jet SQL queries, let's turn to the unglamorous but essential task of maintaining tables, indexes, and relationships. Here you can really leverage your productivity by using Jet SQL's DDL against Jet MDB databases. For example, you need only one SQL query to create a brand new table with indexes. To build the table tblCustomers, write:

```
CREATE TABLE tblCustomers
   ([cFirstName] TEXT,
   [cLastName] TEXT,
   iCustId INTEGER
CONSTRAINT ndxCustId PRIMARY KEY);
```

You must specify the data type for each field listed in the CREATE TABLE command. The data types used in the DDL queries have a close, but not exact, correspondence with the data type name listed when you create tables interactively in Access (see Table 1).

The CONSTRAINT clause in the iCustId specification in the above query

creates ndxCustId, a primary key index, on that field. For PRIMARY KEY you can swap in UNIQUE (or nothing at all, for a non-unique, nonprimary index) where it's appropriate.

For multiple-key indexes, you can use a standalone CONSTRAINT clause that names multiple fields (for details, see the help file or printed documentation in VB4, Access 2, or Access 7). Both single- and multiple-field CONSTRAINT clauses can also use the FOREIGN KEY subclause. This creates referential integrity relationships between tables (though you cannot use SQL to set the cascade update and delete attributes). You can add constraints after creating the table by using the ALTER TABLE command in place of CREATE TABLE.

JET ASSUMES THAT A FOREIGN KEY SHOULD REFERENCE THE PRIMARY KEY IN THE FOREIGN TABLE.

To demonstrate some of these concepts, consider a typical ALTER TABLE query in DDL. It uses ALTER TABLE with a FOREIGN KEY specification to assure referential integrity between tblOrders and tblCustomers:

```
ALTER TABLE tblOrders
ADD CONSTRAINT ndxCustId FOREIGN KEY
   (iCustId)
REFERENCES tblCustomers (iCustId);
```

I used an explicit specification of the tblCustomers.iCustId field at the end of the above query, though I could have skipped it. Jet assumes that a foreign key should reference the primary key in the foreign table. You can build nonprimary key references when the field is specifically listed.

You can also use ALTER TABLE with the ADD COLUMN, DROP COLUMN, and DROP CONSTRAINT reserved words to add and delete fields or indexes. You can also create indexes with CREATE INDEX and delete them with DROP INDEX. You can delete a whole table and all of its indexes with DROP TABLE.

Unlike the CREATE TABLE and ALTER TABLE commands, CREATE INDEX lets you create indexes with descending expressions as well as with IGNORE NULL and DISALLOW NULL validation rules (again, see the help file or printed documentation in VB4, Access 2, or Access 7 for details).

This concludes our crash course in Jet SQL. If you followed this discussion, you have almost completely mastered what you need to know about Jet's JOIN clause, predicates, aggregates, AS and IN clauses, the WITH OWNER ACCESSOPTION feature, parameter and crosstab queries, and DDL commands. Now you have at your disposal a powerful facility for making your applications adept at data manipulation and definition, and folks will be coming to you for help with their queries. ×