Surviving Client/Server: Getting Started With SQL Part 1

by Steve Troxell

Delphi incorporates some very useful database components that make developing database applications a breeze. And because Delphi ships with a local version of Borland's InterBase database server, a great number of application developers now have an easy, inexpensive means of exploring the world of Client/Server technology right at their fingertips. For these reasons, many fledgling Delphi programmers may be introduced to Structured Query Language (SQL) for the first time. This article is intended to introduce you to the principal SQL data access statements: `SELECT`, `INSERT`, `UPDATE` and `DELETE`. These are the workhorses of SQL and by the end of this article you'll be able to effectively use SQL to accomplish a wide variety of tasks. We'll continue to expand our knowledge of SQL in the next issue.

All of the examples in this article use the sample InterBase database EMPLOYEE.GDB that ships with Delphi. You may find it helpful to connect to this database through the Windows Interactive SQL (ISQL) program that ships with Delphi and try the examples as you read about them. It's also handy to be able to experiment as ideas come to you while reading the text. This database is located in the \IBLOCAL\EXAMPLES directory if you installed Delphi with the default directories. We will be adding information in this database so you may want to make a copy of the EMPLOYEE.GDB file and work with the copy only.

Using ISQL

To use Windows ISQL, start the ISQL program from the Delphi program group. From the `File` menu, select `Connect to Database`. In the `Database Connect` dialog box, make sure you've selected `Local Server`, enter the path and filename for the EMPLOYEE.GDB database, enter `SYSDBA` for the user name, and enter `masterkey` for the password (make sure you enter the password in lower case). This is the default system administrator login for InterBase databases.

Using the ISQL program is simple: type in the SQL statement you want to execute in the `SQL Statement` window and click the `Run` button to execute the statement. The results of your statement will appear in the `ISQL Output` window. Once you run an SQL statement, it disappears from the `SQL Statement` window. If you want to run it again (and perhaps make small changes to it), you can retrieve any previous SQL statement by clicking the `Previous` button.

SQL Preliminaries

Before we get started, let's look at a few of the ground rules for working with SQL. First, three new terms: the familiar structures of file, record, and field are called table, row, and column in relational databases. A database is a collection of tables; in Paradox each .DB file represents a table, in InterBase each .GDB file represents a database and the tables are managed internally.

Second, let's take a look at the syntax of a SQL statement. A typical SQL statement might be:

```
SELECT name, address
  FROM customers;
```

SQL itself is case-insensitive, but in this article SQL keywords are shown in all uppercase and table names, column names, etc. are shown in all lower case. SQL generally requires a semi-colon at the end of each statement, but in certain tools (such as ISQL) it's optional.

Third, you can break an SQL statement across multiple lines by pressing `RETURN` anywhere you can legally place a space in the statement.

Finally, you can enclose literal strings with single quotes or double quotes. We'll use single quotes here.

Introducing The Column...

Delphi users seem to have settled down into several groups: firstly occasional programmers or first-time programmers (attracted by Delphi's ease of use), secondly professional full-time developers doing general Windows application building (enthralled by Delphi's amazing productivity) and thirdly those putting together Client/Server systems (impressed by Delphi's robustness and power). This column is aimed at the third group, but especially those who may be dipping a toe into the waters of Client/Server for the first time.

Steve is involved in developing a variety of Client/Server systems in his work at TurboPower Software, using different server databases, and we are looking forward to learning from his experience over the months. As well as SQL – an essential part of the Client/Server developer's skill set – Steve plans to cover a variety of other topics and also provide lots of hints and helps along the way. If there are things which you need some help with, why not drop us a line and let us know!

Reading Rows

SELECT is SQL's data retrieval statement and is probably the most frequently used SQL statement. The basic form of the SELECT statement is:

```
SELECT <column(s)>
  FROM <table(s)>;
```

For example, try the following SELECT statement in ISQL (the results are shown in Figure 1):

```
SELECT last_name, first_name
  FROM employee;
```

We selected the last_name and first_name columns from the employee table. By default, SELECT returns all the rows from the table, but only shows data for the columns we requested (called the select list). If you wanted to see all of the columns in a table, it would be cumbersome to enter the name of every column in the SELECT statement, so SQL provides a convenient shortcut: an asterisk can be used to indicate *all columns*. Try the following in ISQL:

```
SELECT * FROM employee;
```

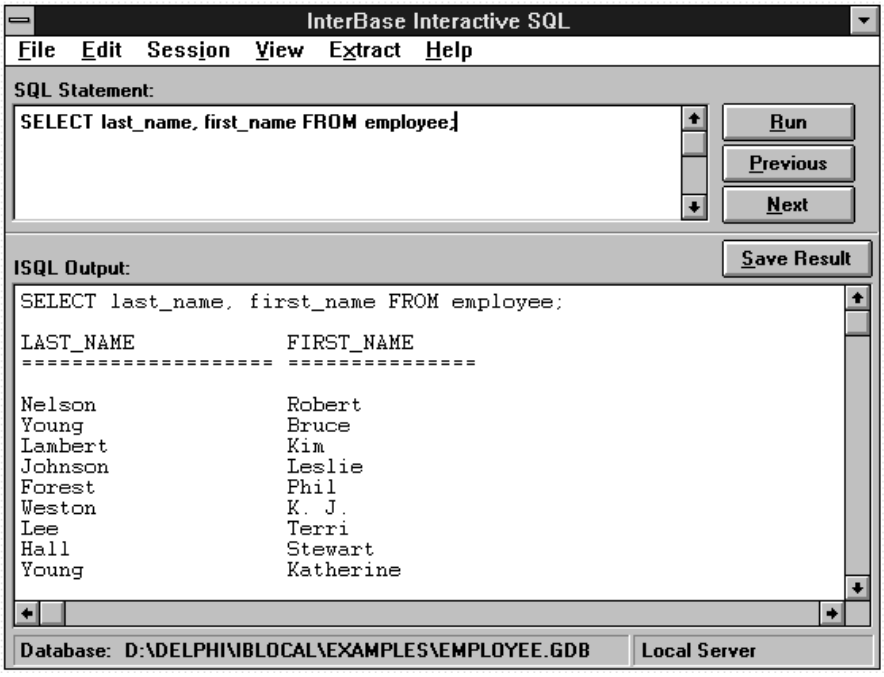
Computed Columns

You can also show calculated results with a SELECT statement. In this case, you simply provide the expression used to make the calculation in place of a column name in the select list. The example below estimates the monthly earnings of each employee:

```
SELECT last_name, salary / 12
  FROM employee;
```

Take a look at the results of this statement in Figure 2. Notice that there isn't a column name for the salary calculation. That's because the data shown doesn't exist as a named column in the table we selected from. A column without a name is just not a very useful thing to have, so we need to assign an alias to the column (see Figure 3):

```
SELECT last_name, salary / 12
  AS monthly_salary
  FROM employee;
```



➤ *ISQL in action*

```
SELECT last_name, first_name FROM employee;

LAST_NAME            FIRST_NAME
==================== ===============

Nelson               Robert
Young                Bruce
Lambert              Kim
Johnson              Leslie
Forest               Phil
Weston               K. J.
```

➤ *Figure 1 (partial listing of rows)*

```
SELECT last_name, salary / 12 FROM employee;

LAST_NAME
==================== ======================

Nelson                                 8825
Young                                  8125
Lambert                              8562.5
Johnson                            5386.25
Forest                                 6255
Weston                          7191.078125
```

➤ *Figure 2 (partial listing of rows)*

There's no reason why we couldn't do the same thing to a regular named column as well, if we wanted to reference the data by something other than its defined column name. You might need to do this if you were selecting data from two or more tables that use the same column name.

Selecting Subsets Of Rows

The selects we've looked at so far return all the rows in the table. Many times you're not interested in wading through all of the rows to get the information you want. Usually all you want is a specific row, or a set of rows with something in common. You use the WHERE clause to restrict the rows returned by SELECT.

For example, the following SQL statement selects all the employees in the Software Development department (see Figure 4):

```
SELECT * FROM employee
  WHERE dept_no = 621;
```

The criteria defined by a WHERE clause is usually referred to as the *search condition*. The SELECT statement returns only those rows that meet the search condition. WHERE supports a variety of conditional operators and allows multiple criteria to be logically combined using AND and OR. Some of the common operations that can be performed are listed in Figure 5. Many SQL databases offer additional operations.

The following examples illustrate some of the expressions you can use to formulate search conditions. They are all valid search conditions for the employee table, so feel free to try them out in ISQL before continuing (try SELECT * FROM employee using each of the following WHERE clauses).

```
WHERE job_grade IN (1,3,4) AND
  job_country = 'USA'

WHERE salary / 12 > 10000

WHERE phone_ext IS NULL

WHERE hire_date BETWEEN
  '1-1-90' AND '12-31-90'

WHERE UPPER(first_name)
  LIKE 'ROBERT%'
```

It should be noted that the columns used in the WHERE clause do not have to be part of the select list; they only have to be available in the table defined in the FROM clause of the SELECT statement.

Wildcarding

You can use wildcards to select on a character column matching a given pattern (amazingly enough, this is sometimes referred to as *pattern matching*).

SQL supports two wildcard characters: % is used to match any number of characters (including zero), and _ is used to match exactly one character. You must use the LIKE operator to use wildcards in a character search, otherwise the wildcards are taken literally.

```
SELECT last_name, salary / 12 AS monthly_salary
  FROM employee;

LAST_NAME                MONTHLY_SALARY
================ =====================

Nelson                             8825
Young                              8125
Lambert                          8562.5
Johnson                         5386.25
Forest                             6255
Weston                     7191.078125
```

➤ *Figure 3 (partial listing of rows)*

```
SELECT * FROM employee WHERE dept_no = 621;

EMP_NO FIRST_NAME      LAST_NAME            PHONE_EXT   HIRE_DATE
====== ============== ==================== ========= ===========

     4 Bruce          Young                233       28-DEC-1988
    45 Ashok          Ramanathan           209        1-AUG-1991
    83 Dana           Bishop               290        1-JUN-1992
   138 T.J.           Green                218        1-NOV-1993
```

➤ *Figure 4 (partial listing of columns)*

Operator	Meaning
=	Equal to
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
BETWEEN x AND y	Range: greater than or equal to <x> and less than or equal to <y>
IS NULL	Contains null value
IS NOT NULL	Contains non-null value
IN (x,y,...)	Value found in a list
NOT IN (x,y,...)	Value not found in a list
LIKE	Matches a wildcard pattern

➤ *Figure 5 Common SQL operators*

Suppose you needed to look up a customer and all you could remember was that they were located on Newbury Street or Avenue or Newbury something. You could use the following SQL statement:

```
SELECT customer, address_line1
  FROM customer
  WHERE address_line1
  LIKE '%Newbury%';
```

As Figure 6 shows, this select finds all rows containing the word Newbury anywhere in the column address_line1, regardless of what characters (if any) preceded or followed the word.

Sorting Rows

You can sort the rows returned by a SELECT statement by including an ORDER BY clause. ORDER BY simply names the columns you want to sort on. Suppose you wanted a list of sales orders sorted by the amount of the order (see Figure 7 for the output):

```
SELECT cust_no, order_status,
  total_value
  FROM sales
  ORDER BY total_value;
```

Like WHERE, the columns defined in ORDER BY do not have to appear in the select list. This is true for InterBase but may not apply to other SQL databases.

You can define the sort sequence for each sort column by adding the ASC (ascending) or DESC (descending) keyword after the column name in the ORDER BY clause. Here's how you would make the query above return a list of sales orders in order of decreasing amount:

```
SELECT cust_no, order_status,
  total_value
  FROM sales
  ORDER BY total_value DESC;
```

As an alternative to specifying the name of the column to sort on, you can specify the number of the column from the select list. This is useful if you've included an unaliased computed column in the select list. For example, suppose you wanted a list of employees in order by monthly salary (see Figure 8):

```
SELECT full_name, salary / 12
  FROM employee
  ORDER BY 2;
```

Here the 2 in the ORDER BY means "order by the second column in the select list"; that is, by the computed column salary / 12.

If you use both a WHERE clause and an ORDER BY clause in the same select statement, the WHERE clause must appear before the ORDER BY clause, as in the following example:

```
SELECT * FROM employee
  WHERE dept_no = 621
  ORDER BY phone_ext;
```

Selecting From Multiple Tables (Joins)

One of the most powerful features of the SELECT statement (and of SQL itself) is the ease with which you can combine data from multiple tables into one informative view.

For example, suppose you want a roster of all employees by department. Employee names are stored in the employee table and department names are stored in the department table, so you'll have to combine this information somehow into a single report. This is where the concept of a "join" comes into the picture.

A join can occur between two or more tables where each pair of tables can be linked by a common field.

For example, departments are identified by the dept_no column in both the employee table and department table, so this column can be used to link the two tables in a join. In SQL you can use the WHERE clause to specify the link field for a join between two tables. The select statement below produces the employee roster we want:

```
SELECT full_name, department
  FROM employee, department
  WHERE employee.dept_no =
    department.dept_no
  ORDER BY department;
```

This means "show the selected columns from the given tables and combine the rows such that dept_no in the employee table matches dept_no in the department table" (in this case the department table also happens to contain a column called department). Take a look at the results shown in Figure 9. The result of a join select is indistinguishable from a single table select.

The WHERE clause defines the association between the two tables. The linking columns are not required to have the same name, but they must be compatible data types. As an alternative to the WHERE clause, you can also define a join in the FROM clause as follows:

```
SELECT full_name, department
  FROM employee JOIN department
    ON employee.dept_no =
      department.dept_no
  ORDER BY department;
```

You can join more than two tables by simply ANDing the join

```
SELECT customer, address_line1 FROM customer
  WHERE address_line1 LIKE '%Newbury%';

CUSTOMER                  ADDRESS_LINE1
========================= ============================

Buttle, Griffith and Co.  2300 Newbury Street
```

➤ *Figure 6*

```
SELECT cust_no, order_status, total_value
  FROM sales
  ORDER BY total_value;

    CUST_NO ORDER_STATUS TOTAL_VALUE
========== ============ ===========

       1003 waiting             0.00
       1001 shipped             0.00
       1006 shipped            47.50
       1014 shipped           100.02
       1010 shipped           210.00
```

➤ *Figure 7 (partial listing of rows)*

```
SELECT full_name, salary / 12 FROM employee ORDER BY 2;

FULL_NAME
================================ ====================

Bennet, Ann                                   1911.25
Brown, Kelly                                     2250
O'Brien, Sue Anne                             2606.25
Guckenheimer, Mark                 2666.666666666667
Reeves, Roger                              2801.71875
```

➤ *Figure 8 (partial listing of rows)*

expressions together in the WHERE clause (or concatenating JOINs in the FROM clause). The linking columns do not have to be the same for all tables in the join. For example, to get a list of all employees assigned to a project and the name of the projects they are assigned to, you must join the employee, employee_project, and project tables:

```
SELECT full_name, proj_id,
  proj_name
  FROM employee,
    employee_project, project
  WHERE employee.emp_no =
    employee_project.emp_no
  AND
   employee_project.proj_id =
   project.proj_id
  ORDER BY full_name;
```

You can see the results in Figure 10.

One improvement we could make is to reduce the bulk of this statement a little by assigning aliases to the tables just as we assigned aliases to columns previously. We do this in the same fashion by following the actual table name with its alias, however we do not use the AS keyword in between. We're going to redefine the employee, employee_project and project tables to have the aliases a, b and c respectively. So our final select statement now looks like:

```
SELECT full_name, proj_id,
  proj_name
  FROM employee a,
   employee_project b,
   project c
  WHERE a.emp_no = b.emp_no
  AND b.proj_id = c.proj_id
  ORDER BY full_name;
```

Summing Up SELECT

SELECT is where most of the power of SQL lies. There are even more clauses and functionality to SELECT than were covered here, so check your manual. This was meant just to show you the basic nuts-and-bolts needed to do anything really useful with SELECT.

In the next issue we'll cover a lot more on SELECT, but now that we've got a handle on looking at the data, we'll turn to altering the data.

```
SELECT full_name, department FROM employee, department
  WHERE employee.dept_no = department.dept_no
  ORDER BY department;

FULL_NAME                         DEPARTMENT
================================= ======================

O'Brien, Sue Anne                 Consumer Electronics Div.
Cook, Kevin                       Consumer Electronics Div.
Lee, Terri                        Corporate Headquarters
Bender, Oliver H.                 Corporate Headquarters
Williams, Randy                   Customer Services
Montgomery, John                  Customer Services
```

➤ *Figure 9 (partial listing of rows)*

```
SELECT full_name, proj_id, proj_name
  FROM employee, employee_project, project
  WHERE employee.emp_no = employee_project.emp_no AND
    employee_project.proj_id = project.proj_id
  ORDER BY full_name;

FULL_NAME                         PROJ_ID PROJ_NAME
================================= ======= ====================

Baldwin, Janet                    MKTPR   Marketing project 3
Bender, Oliver H.                 MKTPR   Marketing project 3
Bishop, Dana                      VBASE   Video Database
Burbank, Jennifer M.              VBASE   Video Database
Burbank, Jennifer M.              MAPDB   MapBrowser port
Fisher, Pete                      GUIDE   AutoMap
Fisher, Pete                      DGPII   DigiPizza
```

➤ *Figure 10 (partial listing of rows)*

Committing Your Work

One of the key characteristics of SQL is that when you add or modify data in the SQL table, the changes are not permanently recorded in the table and other users of the database will not see them, until you commit them. In this way you can work with the data as much as you like until you get it just the way you want it and then commit it permanently to the database.

In ISQL you commit your changes by selecting Commit Work from the File menu. If you want to undo your modifications, you can select Rollback Work from the File menu. This will rollback all the changes you've made since the last time you committed your work. Think of this as a refresh of the data you're working with.

Be careful if you decide to experiment with these data modification statements outside the examples given. The tutorial database provided with InterBase defines some data validation and referential integrity constraints that may give you errors if you don't modify the data just right (this is yet another boon of SQL by helping preserve data integrity through automatic means).

Adding New Rows

We use the INSERT statement to add a new row to a table. INSERT expects us to enumerate the values of each column in the table for the new row.

For example, the country table identifies the currency used in a particular country and contains two columns: country and currency. To add a new country row in this table we would use:

```
INSERT INTO country
  VALUES ('SteveLand',
    'Twinkies');
```

In this case Twinkies are the form of currency in SteveLand. Try entering this statement into ISQL and then select all the rows from country to see the result.

The data values must appear in the same order as the columns are defined; the first value given will be inserted into the first column, the second value into the second

column, and so on. Alternatively, you can enter the values in any order you like as long as you specify the column names after the table name. The data values must then be in the order of the columns as given. For example:

```
INSERT INTO country (currency,
  country)
  VALUES ('Clams',
   'Troxellvania');
```

Using this same syntax you can insert data into only certain columns instead of all of them. Any columns not specifically included in the INSERT receive a null value. Try this in ISQL and then select all rows to see what happens:

```
INSERT INTO customer (customer)
  VALUES
    ('Bigwig International');
```

The results of this statement are shown in Figure 11. Notice that the customer column was set as we specified, and all the other columns except cust_no were set to null. Cust_no was set to a new value because of two advanced concepts called *triggers* and *generators*; concepts that are outside the scope of this article, but we'll cover them in a future issue.

Copying Rows

It is possible to combine the INSERT and SELECT statements to allow you to copy specific columns from existing rows in one table to another table. In this case, you simply omit the VALUES clause containing the explicit values and replace it with any legal SELECT statement with the same number and type of columns that you are inserting. It's difficult to illustrate this concept with the sample employee database we've been using, so here is a contrived example:

```
INSERT INTO shipped_orders
  (order_num, amount)
  SELECT order_num,
    order_total FROM orders
    WHERE order_status =
      'shipped';
```

In this example, we are reading all the rows from the orders table that have a status of shipped. For each row we find, we are inserting a row into the shipped_orders table that consists of the order_num and order_total columns from orders (realistically, we would probably delete these rows from orders after we've copied them). If there are any additional columns in shipped_orders, they default to null since we did not provide a value for them. Notice that the column names in the read table do not have to match the column names in the write table. We only need to have the same number of columns and of the same data type.

Changing Rows

Now that we have a new customer, let's add some useful information. To change the value of a column within an existing row we use the UPDATE statement. Let's add Joe Johnson as the contact for our new customer. Try the following in ISQL and examine the results:

```
UPDATE customer
  SET contact_first = 'Joe',
    contact_last = 'Johnson'
  WHERE customer =
    'Bigwig International';
```

(Note: Bigwig International must be spelled and cased exactly as you originally entered it in the INSERT statement previously).

The SET clause specifies a list of columns to modify and their new values. The WHERE clause operates just like the WHERE clause in the SELECT statement and defines the rows to apply the change to. If you omit the WHERE clause, the change will be applied to all rows in the table.

UPDATE can be used to set columns with calculated values as well. If you were in a generous mood and wanted to double everybody's salary, you could try:

```
UPDATE employee
  SET salary = salary * 2;
```

Removing Rows

Sooner or later, you'll need to remove some of the data from a table. To delete rows we use the DELETE statement. DELETE simply uses a WHERE clause to identify the rows to delete from a given table.

Let's say our company is downsizing and we now need to remove the Software Development department from the department table. Try the following in ISQL and then select all rows from the table to see the results:

```
DELETE FROM department
  WHERE dept_no = 621
```

If the WHERE clause is omitted, then all of the rows in the table are deleted.

Conclusion

There you have it: the real meat of SQL. We've covered the principal statements of SQL and with these you can perform a great deal of the common tasks of relational databases. In the next issue we'll cover some more features of SELECT that simplify creating reports from SQL data.

Steve Troxell is a Software Engineer with TurboPower Software where he is developing Delphi Client/Server applications using InterBase and Microsoft SQL Server for parent company Casino Data Systems. Steve can be reached on CompuServe at 74071,2207

➤ *Figure 11 (partial listing)*

```
SELECT * from customer

CUST_NO CUSTOMER                  CONTACT_FIRST   CONTACT_LAST
======= ========================= =============== ==================

   1008 Anini Vacation Rentals    Leilani         Briggs
   1009 Max                       Max             <null>
   ...more rows...
   1015 GeoTech Inc.              K.M.            Neppelenbroek
   1016 Bigwig International      <null>          <null>
```