PAPER 242-31

SQL SET OPERATORS: SO HANDY VENN YOU NEED THEM

Howard Schreier, Howles Informatics

ABSTRACT

When it comes to combining data from multiple tables in the SQL Procedure, joins get most of the attention and subqueries are probably second. Often overlooked are the set operators (OUTER UNION, UNION, INTERSECT, and EXCEPT). This tutorial begins by relating OUTER UNION to similar functionality provided by the DATA step's SET statement, then explains and demonstrates the full repertoire of set operators.

INTRODUCTION

Set operators are so designated because they are conceptually derived from mathematical set theory. The three basic set operators are UNION, INTERSECT, and EXCEPT. All three, with variations (in particular, OUTER UNION) and options, are implemented in PROC SQL.

JOINS VS. SET OPERATORS

Before we delve into the details of PROC SQL's set operators, let's establish the fundamental distinctions between joins and set operators. This can be done with a simple example, starting with the creation of two tiny tables. Here's the code:

```
DATA first;
A = 1;
RUN;

DATA second;
B = 2;
RUN;
```

So each of these tables has one row and one column. We can use PROC SQL to combine the two via a simple cross join:

```
SELECT       *
FROM         first, second
;
```

The result is:

```
       A         B
------------------
       1         2
```

Now we'll look at the UNION, which is the simplest form of the most widely used of the set operators. The code to combine our two tables is:

```
SELECT      *
FROM        first
UNION
SELECT      *
FROM        second
;
```

Before we look at the effect of this statement, let's look at the syntax and compare it to that of the join. Notice that "UNION" is inserted between two SELECTs (each of which has, as it must, a subordinate FROM clause). A set operator works on the **results** of **two** SELECTs. This is unlike a join, which is implemented **within** the FROM clause of a **single** SELECT. Notice also that there is but one semicolon, terminating the entire composite statement.

Now it's time to look at the result generated by this code:

```
       A
--------
       1
       2
```

We see the two numeric values, this time arranged vertically rather than horizontally. This reflects the fundamental difference between joins and set operators. Joins align rows and accrete columns; set operators align columns and accrete rows. This is something of an oversimplification of course. SQL is not a matrix language and provides relatively little symmetry between rows and columns. So the contrast drawn here between joins and set operators is only a foundation for the details to follow.

OUTER UNION

Not all of the PROC SQL set operators have DATA step counterparts, and in some cases the DATA step counterparts are rather convoluted. Since the OUTER UNION operator, with the CORRESPONDING option in effect, does have a straightforward DATA step parallel, we'll start with it.

Our first chore is to create a pair of tables with which to demonstrate. This code:

```
CREATE TABLE one AS
SELECT      name, age, height
FROM        sashelp.class
WHERE       age<14 and LENGTH(name)<6
ORDER BY    age, RANUNI(1)
;

CREATE TABLE two AS
SELECT      name, weight, age
FROM        sashelp.class
WHERE       age<14 and LENGTH(name)>5
ORDER BY    age, RANUNI(1)
;
```

produces ONE:

Name	Age	Height
Joyce	11	51.3
John	12	59
Jane	12	59.8
James	12	57.3
Alice	13	56.5

and TWO:

Name	Weight	Age
Thomas	85	11
Robert	128	12
Louise	77	12
Jeffrey	84	13
Barbara	98	13

The reference to the pseudo-random number function RANUNI in the ORDER BY clauses has succeeded in shuffling the order of the rows within each AGE group.

Concatenation

The two data sets can be combined vertically, or concatenated, in a DATA step by naming them both in a single SET statement. Here is the code:

```
DATA concat;
SET one
    two
    ;
RUN;
```

The result looks like this:

Name	Age	Height	Weight
Joyce	11	51.3	.
John	12	59	.
Jane	12	59.8	.
James	12	57.3	.
Alice	13	56.5	.
Thomas	11	.	85
Robert	12	.	128
Louise	12	.	77
Jeffrey	13	.	84
Barbara	13	.	98

The equivalent SQL statement is:

```
CREATE TABLE concat AS
SELECT       *
FROM         one
OUTER UNION CORRESPONDING
SELECT       *
FROM         two
;
```

It produces the same result.

Notice that the common (that is, like-named) columns (NAME and AGE) have been aligned. That is a consequence of the CORRESPONDING option. The mismatched columns (HEIGHT and WEIGHT) also appear, with missing values for the cells which have no “ancestry” in the source tables; that is characteristic of an **OUTER** UNION (as distinguished from a simple UNION). The fact that the DATA step and PROC SQL place the columns in the same order is a coincidence, attributable to the relative simplicity of this example (SQL places the common columns first, followed by the columns which appear only in the first SELECT, followed by the columns which appear only in the second SELECT; the DATA step places the variables from the first data set first, followed by the variables which are found only in the second data set).

The order of the rows is also the same. Since there is no ORDER BY clause in the SQL code, the SQL processor is not obligated to deliver its results in any particular order. The order we observe is basically a consequence of internal optimization; the processor is avoiding unnecessary work by simply preserving the order in which it encounters the rows.

Data Type Compatibility

The alignment of columns in these examples has worked smoothly because the aligned columns have matched with respect to data type (numeric or character). Since column alignment is an essential aspect of almost all of the set operators, it’s worth exploring this a bit more. We’ll need some test data sets with deliberate type mismatches:

```
DATA num;
id = 3;
value = 0;
RUN;

DATA char;
id = 4;
value = 'abc';
RUN;
```

Notice that VALUE is numeric in data set NUM but character in data set CHAR. So when we attempt a DATA step concatenation with

```
DATA both;
SET num char;
RUN;
```

we get a failure, with this log message:

```
ERROR: Variable value has been defined as both character and numeric.
```

The new data set (BOTH) is created, but contains no observations. If we run the parallel SQL code:

```
CREATE TABLE both AS
SELECT       *
FROM         num
OUTER UNION CORRESPONDING
SELECT       *
FROM         char
;
```

the log message is

```
ERROR: Column 2 from the first contributor of OUTER UNION is not the same
type as its counterpart from the second.
```

Unlike the DATA step, PROC SQL does not create even an empty table in this situation.

There is just one set operator which is immune to data type mismatches because it does no column alignment; that is the OUTER UNION operator, **without** the CORRESPONDING option. To illustrate, we can run a display query (that is, a freestanding SELECT statement rather than one within a CREATE statement):

```
SELECT       *
FROM         num
OUTER UNION
SELECT       *
FROM         char
;
```

The result is

```
      id     value        id  value
-----------------------------------
       3         0         .
       .         .         4  abc
```

Notice that the original ID columns from the two source tables are kept separate, even though they are compatible with regard to data type. The OUTER UNION operator attempts no column alignment whatsoever. Thus it is immune to error conditions due to type mismatch and will display all data from the source tables. However, this capability is rather limited in value because the results cannot always be loaded into a table. If we try, by running the same query within a CREATE TABLE statement, as

```
CREATE TABLE not_corr AS
SELECT       *
FROM         num
OUTER UNION
SELECT       *
FROM         char
;
```

we get

```
WARNING: Variable id already exists on file WORK.NOT_CORR.
WARNING: Variable value already exists on file WORK.NOT_CORR.
```

and the new table only contains two columns and looks like this:

id	value
3	0
.	.

So, when set operators are used, the burden of assuring that aligned columns are compatible with respect to type rests with the programmer.

OVERVIEW: UNION, INTERSECT, AND EXCEPT

The available set operators, and the variations introduced through the use of optional keywords, can be categorized in terms of four issues, which in turn can be presented as two pairs of two. The issues:

(1A) What is the rule for aligning columns?

(1B) What is done with columns which do not align?

(2A) What is the rule for accreting rows?

(2B) Are duplicate rows allowed to appear in the result?

This framework will reveal that the OUTER UNION set operator is rather distinctive and unlike the other three (UNION, INTERSECT, and EXCEPT), which differ from one another only in terms of their row-accretion rules (2A). That's why this overview makes its appearance midway through this paper.

Specifically, when we look at the row-accretion rules, we will see that UNION, like OUTER UNION, accepts those rows which appear in either operand (that is, in the results produced by either embedded SELECT clause). INTERSECT accepts those rows which appear in both operands. EXCEPT accepts rows which appear in the first operand but are absent in the second.

In other respects, UNION, INTERSECT, and EXCEPT are essentially alike in behavior, and stand in contrast to OUTER UNION.

Recall that OUTER UNION aligns columns by name if the CORRESPONDING option is coded. The other three set operators share this feature. However, in the absence of the CORRESPONDING option, OUTER UNION does no alignment; in contrast, the default rule for UNION, INTERSECT, and EXCEPT is to align by position.

The OUTER UNION operator preserves columns which do not align, and generates nulls (missing values) to complete the table. The other set operators shed unaligned columns if CORRESPONDING is specified, but not if the default positional alignment is in effect.

The UNION, INTERSECT, and EXCEPT operators by default purge duplicate rows (although the optional ALL keyword can be used to preempt this behavior). Because OUTER UNION results typically include mismatched columns, filled in with missing values, the very concept of duplicate rows is elusive; so OUTER UNION results simply preserve all rows.

Taken together, the shared characteristics of the UNION, INTERSECT, and EXCEPT set operators limit the extent to which equivalent processes can be simply coded using the DATA step. This is another point of contrast with the OUTER UNION operator.

We've already seen the OUTER UNION operator in an example. In the sections which follow, the behavior of the UNION, INTERSECT, and EXCEPT set operators will be illustrated through examples. Because of the extensive similarities among the three and because UNION is probably the most widely used, it will be covered first, and most extensively. Then the distinctive characteristics of INTERSECT and EXCEPT will be presented.

UNION

We'll begin looking at the UNION operator by using both the ALL and CORRESPONDING options. This yields the form of UNION which most closely resembles the OUTER UNION CORRESPONDING which we examined earlier. To demonstrate using the same data, we run

```
CREATE TABLE unionallcorr AS
SELECT       *
FROM         one
UNION ALL CORRESPONDING
SELECT       *
FROM         two
;
```

which yields

Name	Age
Joyce	11
John	12
Jane	12
James	12
Alice	13
Thomas	11
Robert	12
Louise	12
Jeffrey	13
Barbara	13

Tables ONE and TWO have columns NAME and AGE in common, so those are the columns which emerge in this result. Note that the data from the two AGE columns are properly combined in a single column, even though AGE is the second column in ONE and the third column in TWO. Each source also had an additional column (HEIGHT in ONE and WEIGHT in TWO), but these are shed by the UNION operator because their names do not match.

The ALL keyword prevents the UNION operator from eliminating duplicate rows. It is not well illustrated here, because it happens that there are no duplicates. Later, when we turn from column alignment issues to the subject of row accretion, we will examine and illustrate the effect of "ALL"; for now, note that if we omitted it, we would get the same rows, though they would be ordered differently as a side effect of the process which detects duplicates.

When you know that there are no duplicate rows, coding ALL can speed up processing by avoiding the search for duplicates. This is especially true if you don't need an ORDER BY clause.

Next, let's eliminate "CORRESPONDING" and investigate the alternative column alignment rule. Here is the code:

```
CREATE TABLE unionall AS
SELECT       *
FROM         one
UNION ALL
SELECT       *
FROM         two
;
```

and the result:

Name	Age	Height
Joyce	11	51.3
John	12	59
Jane	12	59.8
James	12	57.3
Alice	13	56.5
Thomas	85	11
Robert	128	12
Louise	77	12
Jeffrey	84	13
Barbara	98	13

It appears that there are a number of implausibly short and elderly students. What has happened, of course, is that the columns were aligned by position rather than by name; recall that the second column in table TWO is WEIGHT and the third column is AGE.

Don't conclude that omitting the CORRESPONDING keyword always leads to trouble. That was the case here because the column naming was consistent whereas the column ordering was not. In other situations the opposite might be true. Whenever the asterisk (*) is used in either or both of the SELECT clauses, the column alignment is to some extent implicit, and the appropriateness of the result will depend on consistency of table organization. Remember that you can always use explicit SELECT lists to control more precisely the column alignment. For example, the last example could be fixed by changing the code to

```
CREATE TABLE unionall AS
SELECT       *
FROM         one
UNION ALL
SELECT       name, age
FROM         two
;
```

The columns of table ONE are still encountered in their stored order: NAME, AGE, HEIGHT. However, the second SELECT clause now explicitly calls for just two columns from table TWO, and in the appropriate order.

The result is:

Name	Age	Height
Joyce	11	51.3
John	12	59.0
Jane	12	59.8
James	12	57.3
Alice	13	56.5
Thomas	11	.
Robert	12	.
Louise	12	.
Jeffrey	13	.
Barbara	13	.

Because the alignment is, in the absence of the CORRESPONDING option, position-based, the leftover column (HEIGHT) from table ONE is not discarded. Rather, it is included in the result, with nulls (missing values) occupying the rows drawn from table TWO.

The alignment of columns by position has no counterpart in the DATA step. When a DATA step (specifically, the SET statement) handles variables originating in different data sets, they are aligned strictly by name. The DATA step also lacks a mechanism for automatically shedding variables which do not align. Instead, all variables survive, with missing values arising where source data sets do not supply values. All this is another way of saying, again, that the behavior of the DATA step parallels that of PROC SQL's OUTER UNION operator with the CORRESPONDING option, and not any flavor of the simple UNION operator.

One could probably use the DATA step and other non-SQL SAS® facilities to emulate alignment by position and shedding of non-aligned columns. However, it would be intricate, involving things like inspection of metadata and a lot of systematic renaming of variables, and is beyond the scope of this paper.

At this point we have pretty much covered the issue of column alignment. On the other hand, the example we have been using does not illustrate the issues and exercise the features pertaining to row accretion. So we will introduce a new example, one which makes column alignment a non-issue. It involves two tables; the first is named ABC and looks like this:

v
a
a
b
b
b
b
c
c

and the second is named AB and looks like this:

v
a
a
a
b
b

Because each has only a single column, and the columns in both tables have the same name and the same data type (character, in this case), there is only one possible column alignment, and it will occur whether or not the CORRESPONDING keyword is coded. Indeed, column alignment is a non-issue.

So the following examples will be about row accretion. The fact that these tables are in sorted order is quite incidental. The SQL processor does not even know they are sorted.

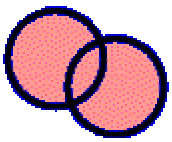
We'll start the exploration of row accretion by considering the ALL option. It's a negative option, in the sense that coding it causes PROC SQL to **not** do something (purge duplicates) which it would otherwise do by default. So the following query:

```
CREATE TABLE unionall AS
SELECT       *
FROM         abc
UNION ALL
SELECT       *
FROM         ab
;
```

has as its result

v
a
a
b
b
b
b
c
c
a
a
a
b
b

This is simply the concatenation of the two sources (that is, in this case, the tables ABC and AB). Of course, the ordering is incidental, since no ORDER BY clause was coded. The significant thing is the number of times each distinct row appears. The accretion rule for UNIONs is that a row appears in the result if it appears in **either** source. When the ALL option is used, the number of times it appears is the sum of its populations in the two sources. That is, if F represents the number of times a distinct row appears in the first source (the result of the first SELECT clause) and S represents the count from the second source, the row will appear F+S times in the result. So, because "b" appears four times in ABC and twice in AB, it appears six times in the simple UNION. Note that this is also the row accretion rule used by the OUTER UNION operator.



So let's see what happens when the ALL keyword is removed:

```
CREATE TABLE union AS
SELECT       *
FROM         abc
UNION
SELECT       *
FROM         ab
;
```

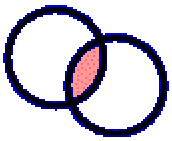
The result is

v
a
b
c

As stated earlier, in the absence of "ALL" the duplicate rows are purged.

The UNION operator is commutative, meaning that the results are not changed if the two operands are interchanged. However, column names and other attributes could be affected by such a switch.

INTERSECT



We turn now to the INTERSECT operator. With regard to column alignment, it behaves just as the UNION operator does, so we won't repeat those details here. However, whereas the UNION operator accepts rows which appear in either source, INTERSECT accepts only those rows which appear in both. We'll start by using it with the ALL keyword:

```
CREATE TABLE intersectall AS
SELECT       *
FROM         abc
INTERSECT ALL
SELECT       *
FROM         ab
;
```

which gives us

v
a
a
b
b

Here, if F represents the number of times a distinct row appears in the first source (the result of the first SELECT clause) and S represents the count from the second source, the row will appear min(F,S) times in the result.

If we remove the ALL option, leaving the query as

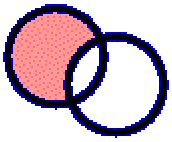
```
CREATE TABLE intersect AS
SELECT       *
FROM         abc
INTERSECT
SELECT       *
FROM         ab
;
```

the duplicates are removed and the result is

v
a
b

Like the UNION operator, INTERSECT is commutative. The positions of the operands can be switched without affecting the content of the result.

EXCEPT



Finally, we consider the EXCEPT operator. With regard to column alignment, it behaves just as the UNION operator does, so we won't repeat those details here. EXCEPT's accretion rule is to preserve rows which appear in the first operand (SELECT clause), but not in the second. Another way of saying this is that rows are taken from the first operand unless they are cancelled by virtue of appearance in the second operand. We'll illustrate, first with the ALL option in effect.

```
CREATE TABLE exceptall AS
SELECT       *
FROM         abc
EXCEPT ALL
SELECT       *
FROM         ab
;
```

gives us

v
b
b
c
c

Here, if F represents the number of times a distinct row appears in the first source (the result of the first SELECT clause) and S represents the count from the second source, the row will appear max(0,F-S) times in the result.

If we remove the ALL option, leaving the query as

```
CREATE TABLE except AS
SELECT       *
FROM         abc
EXCEPT
SELECT       *
FROM         ab
;
```

the duplicates are removed and the result is

v
c

Unlike UNION and INTERSECT, EXCEPT is **not** commutative. Switching the operands changes the result. To illustrate,

```
CREATE TABLE switched AS
SELECT       *
FROM         ab
EXCEPT ALL
SELECT       *
FROM         abc
;
```

gives us

v
a

and without the ALL option returns no rows whatsoever.

EXAMPLES

Brevity

The scenario: You have data sets on sales for a number of years. Here's a DATA step to generate some test data:

```
DATA sales2004 sales2005 sales2006;
DO cust_id = 1001 TO 9999;
   DO year = 2004 TO 2006;
      date = MDY(1,1,year);
      IF ranuni(123)>0.5 THEN
       DO UNTIL (date > MDY(12,31,year) );
         date + ROUND(RANUNI(123) * 80);
         value = ROUND(250 * RANUNI(123),0.01);
         IF RANUNI(123)>0.6 THEN SELECT (year);
            WHEN (2004) OUTPUT sales2004;
            WHEN (2005) OUTPUT sales2005;
            WHEN (2006) OUTPUT sales2006;
            END;
         END;
      END;
   END;
RUN;
```

A few randomly chosen observations from 2005:

Obs	cust_id	year	date	value
2755	2417	2005	17JUN2005	60.14
4475	3265	2005	08JUL2005	87.39
4957	3508	2005	16JUN2005	33.96
6258	4126	2005	30DEC2005	91.90
11350	6650	2005	05NOV2005	150.27
13123	7519	2005	13MAR2005	30.44
14288	8144	2005	18JUN2005	142.06

You want to target some special promotions at people who were at one time good customers, but who then "disappeared" before more recently returning. The specific criteria: (1) at least $1,000 in orders during 2004, no orders in 2005, at least one order in 2006.This problem certainly can be solved with a DATA step. But first the data must be aggregated so that there is one observation per customer in each annual file. PROC SUMMARY is a good tool; here is the code for 2004:

proc summary data=sales2004 nway;
class cust_id;
output out=sum2004(where = (value2004>1000)) sum(value)=value2004;
run;The other years are a bit simpler, since the dollar values are not needed:

```
PROC SUMMARY DATA=sales2005 NWAY;
CLASS cust_id;
OUTPUT OUT=sum2005;
RUN;

PROC SUMMARY DATA=sales2006 NWAY;
CLASS cust_id;
OUTPUT OUT=sum2006;
RUN;
```

The solution can be derived by merging the three years' data:

```
data target;
merge sum2004(keep=cust_id in=in2004)
      sum2005(keep=cust_id in=in2005)
      sum2006(keep=cust_id in=in2006);
by cust_id;
if in2004 and (not in2005) and in2006;
run;;
```

Now here is a solution using SQL set operators:

```
PROC SQL;
CREATE TABLE target_sql AS
SELECT       cust_id
FROM         sales2004
GROUP BY     cust_id
HAVING       SUM(value)>1000
INTERSECT
SELECT       cust_id
FROM         sales2006
EXCEPT
SELECT       cust_id
FROM         sales2005
;
QUIT;
```

One somewhat long statement has replaced four separate steps. The SQL solution is more straightforward in the way it expresses and links the conditions. The results are identical.

Speed

The scenario: You have a table with people's names, phone numbers, and e-mail addresses. There is some duplication, and also inconsistency in how the names are recorded (eg, nicknames vs. formal names). Phone number and e-mail addresses are easier to standardize, and that's already been done. The present task is to detect possible duplicates by finding pairs of observations where either phone numbers or e-mail addresses (or both) match, but where names do **not** match.

Here's a test data generator:

```
DATA roster;
DO i = 1 TO 30000; DROP i;
   name = i;
   phone = i + 0.1;
   email = i + 0.2;
   OUTPUT;
   IF RANUNI(111)>0.8  THEN name = name + 0.01;
   IF RANUNI(111)>0.8  THEN OUTPUT;
   END;
RUN;
```

The data are not realistic, but are suitable nevertheless for demonstration purposes. There are 36,052 observations generated.

A solution in SQL is rather straightforward:

```
CREATE TABLE slow AS
SELECT       DISTINCT roster.name,
                      copy.name AS diff_name
FROM         roster JOIN roster AS copy
ON           roster.phone=copy.phone OR
             roster.email=copy.email
WHERE        roster.name NELT copy.name;
```

The log shows:

```
NOTE: The execution of this query involves performing one or more Cartesian
product joins that can not be optimized.

NOTE: Table WORK.SLOW created, with 2432 rows and 2 columns.

NOTE: SQL Statement used (Total process time):
      real time           3:49.03
      cpu time            3:44.75
```

Because of the "OR" in the ON clause, the SQL processor could not optimize the evaluation. Instead it had to examine all of the potential name pairs, and there are more than a billion of those (36,052 squared). The code works, but the test took nearly four minutes.

We can separate the query into two parts, one for each of the join conditions, and combine the results with the UNION operator.

```
CREATE TABLE fast AS
SELECT       roster.name,
             copy.name AS diff_name
FROM         roster JOIN roster AS copy
ON           roster.phone=copy.phone
WHERE        roster.name NE copy.name
UNION
SELECT       roster.name,
             copy.name AS diff_name
FROM         roster JOIN roster AS copy
ON           roster.email=copy.email
WHERE        roster.name NE copy.name;
```

The DISTINCT specification can be omitted now because the UNION operator has the same effect. Logically, the two versions are equivalent, and they produce the same results. However, the log messages for the second form are:

```
NOTE: Table WORK.FAST created, with 2432 rows and 2 columns.

NOTE: SQL Statement used (Total process time):
      real time           0.45 seconds
      cpu time            0.39 seconds
```

The time required is now less than a second, a tiny fraction of what it was using the first form. Instead of examining a billion rows, the computer searched over merely tens of thousands of rows, twice, then combined those results.

SUMMARY

Set operators complement joins by providing alternative ways of combining data from multiple sources. Typically, set operators perform end-to-end combinations, in contrast to the side-by-side combinations which result from joins.

The OUTER UNION operator in a number of ways resembles the operation of a SET statement which processes two data sets in a DATA step. The other three set operators (UNION, INTERSECT, and EXCEPT) differ in nature from the OUTER UNION, differ from each other in terms of the set-theoretic rules they implement, but resemble one another in terms of their mechanics. UNION, INTERSECT,

and EXCEPT do not have simple DATA step counterparts, though some emulation can be programmed.

REFERENCES

SAS Institute Inc. (2004*), SAS 9.1 SQL Procedure User's Guide*
http://support.sas.com/onlinedoc/913/docMainpage.jsp or
http://support.sas.com/documentation/onlinedoc/91pdf/sasdoc_91/base_sqlproc_6992.pdf

SAS Institute Inc. (2004*), Base SAS 9.1.3 Procedures Guide* http://support.sas.com/onlinedoc/913/docMainpage.jsp or http://support.sas.com/documentation/onlinedoc/91pdf/sasdoc_913/base_proc_8417.pdf

REVISION HISTORY

Revised May 2007. Corrected misstatement regarding the treatment of not-aligned columns by UNION ALL. Refined PROC SUMMARY and DATA step code in the brevity example. Other, minor, editorial changes.
First presented at SUGI 31 (March 2006)

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Howard Schreier
Howles Informatics
Arlington VA

703-979-2720

hs AT howles DOT com
http://howles.com/saspapers/