Paper BB-03

Leveraging SQL Return Codes When Querying Relational Databases

John E. Bentley, Wells Fargo Bank, Charlotte, North Carolina

ABSTRACT

When querying a relational database, each PROC SQL query automatically produces macro variables that contain return codes and messages from the relational database, and the PROC SQL itself produces two more that can be of great help. This paper will explain how and when these macro variables are generated, what they represent, and how to use them to make your programs more dynamic, robust, and error-proof. For real-world examples we will look at using SQL return codes to (1) cleanly halt execution when a database query fails and send an email with the error message, and (2) get the number of records written to a table without running SELECT COUNT(*). The presentation will be useful to all levels of SAS® users not familiar with SQL return codes.

INTRODUCTION

Good news and bad news. The good news is that, generally speaking, the number of coding problems from logic and syntax goes down as a programmer becomes more experienced with the particular language they're working with. We control the level and magnitude of these problems and over time learns how to anticipate and avoid them.

The bad news is that problems beyond our control never go away. Data quality is usually out of a programmer's control, and we also have to deal with things like delayed production schedules, failed loads, missing files, and changes to ETL rules that we never hear about.

But there's more good news. We can take steps to anticipate and deal with some of the problems out of our control. We can't "fix" these problems, but we can write code that anticipates them and lessens their impact. When querying relational databases, for example, SQL return codes are a powerful tool to help us do that.

RETURN CODES AND ERROR MESSAGES

In SAS, a return code, also known as an 'rc', is a numeric code produced by a processing step (PROC or DATA step) that for our purposes provides a quick yes or no answer to the question "was the task successful". By itself it's not much help in figuring out what went wrong but we can use it to identify if there is a problem and as a pointer to the course of action we want to take. As a general rule, for SAS return codes a value of zero means that yes, the task successfully completed.

An error message is a text string that explains the return code. Some messages are more user friendly than others, but an error message almost always does a better job than the return code at pointing you in the direction of where the problem lies. "Bad command or file name" is not much help, but "Lp0 on fire" is unambiguous. This author would argue that compared to others, SAS error messages overall are actually pretty good.

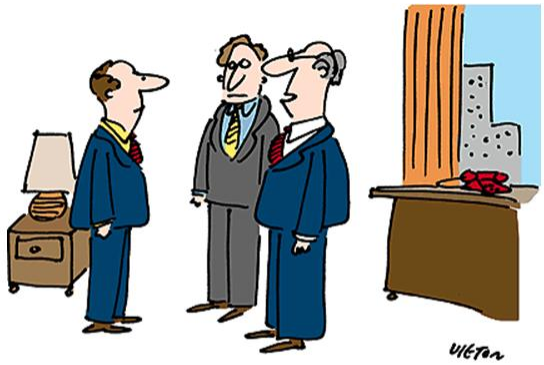

Figure 1. The root cause of many errors

SAS AUTOMATIC MACRO VARIABLES

As soon as a SAS starts, the macro processor creates a number of macro variables that capture information about the session and are available to the user. With the exception of one (SYSPBUFF), automatic macro variables are global in scope.

To use an automatic macro variable, reference it in the same way a user-defined macro variable is referenced—with an ampersand followed by the macro variable name. In an example from the SAS on-line documentation, this is FOOTNOTE statement contains references to the automatic macro variables SYSDAY and SYSDATE:

```
FOOTNOTE "Report for &sysday, &sysdate";
```

If the current SAS session was invoked on August 7th, 2012 the macro variable resolution produces this SAS statement:

```
FOOTNOTE "Report for Tuesday, 07AUG2012";
```

SAS procedures and the DATA step also return a couple automatic macro variables.

- SYSCC holds the overall session return code to that point.
- SYSERR stores the return code of a PROC or DATA step and are reset at each step boundary. Carpenter (2004) notes that the SYSERR values can differ between steps—the value assigned to SYSERR is dependent on both the type of step and the type of error.

To see the return code for a step, use `%PUT`:

```
<PROC or a DATA step code>
%put SYSERR= &SYSERR;
```

The documentation for every version of SAS has a table listing the automatic macro variables and detailed descriptions of what they represent, the data they contain, and whether or not the user can change the value. To learn more, definitely read the documentation. Just search for 'automatic macro variables'.

You can easily see the current values of your session's automatic macro variables. `%PUT _AUTOMATIC_;` dumps a list of all the automatic macro variables and their values to the log. You can also see the system and user-defined macro variables by using `%PUT _SYS_;` and `%PUT _USER_;`. The command `%PUT _ALL_;` writes all macro variable values to the log.

PROC SQL AUTOMATIC MACRO VARIABLES

Every query in PROC SQL generates four automatic macro variables: SQLRC, SQLOBS, SQLOOPS, and SQLEXITCODE. For a PROC SQL that runs multiple queries the value of the first three changes with each query. At the end of the PROC, the final values will be those for the last query. The value of SQLEXITCODE is set only at the end of the PROC.

SQLRC contains a numeric value that provides the completion status of the query. These are straight-forward values which show the success or failure of a query, but the UNDO_POLICY= option and the SQLUNDOPOLICY system option can affect the value assigned. If you use either of these two options, read the documentation for the implications on SQLRC.

RC	Definition
0	PROC SQL statement completed successfully with no errors.
4	PROC SQL statement encountered a situation for which it issued a warning. The statement continued to execute.
8	PROC SQL statement encountered an error. The statement stopped execution at this point.
12	PROC SQL statement encountered an internal error, indicating a bug in PROC SQL that should be reported to SAS Technical Support. These errors can occur only during compile time.
16	PROC SQL statement encountered a user error. For example, this error code is used, when a subquery (that can return only a single value) evaluates to more than one row. These errors can be detected only during run time.

RC	Definition
24	PROC SQL statement encountered a system error. For example, this error is used, if the system cannot write to a PROC SQL table because the disk is full. These errors can occur only during run time.
28	PROC SQL statement encountered an internal error, indicating a bug in PROC SQL that should be reported to SAS Technical Support. These errors can occur only during run time.

Table 1. PROC SQL Return Codes—SQLRC

SQLOBS is the number of rows that were output by the query. The documentation says it "contains the number of rows that were processed by a SQL procedure statement" but this is a bit misleading. It is not always the number of rows 'processed' to get the results set. This query illustrates the issue.

```
libname prod oracle path='STAGE' user="&_hemidbuser" pass="&_hemidbpwd"
                    schema='HM_LOAD' readbuff=10000 multi_datasrc_opt=in_clause;

proc sql threads;
   select count(*)
   from prod.phone_data;
quit;


%put sqlobs= &sqlobs;
```

The SQLOBS value is assigned after the SELECT statement executes. In this case the number of records read to get the count total is over 9 million but the value of SQLOBS is 1 because only 1 record was output. More on SQLOBS later.

The documentation provides some important considerations about how SQLOBS is affected by the NOPRINT option. With NOPRINT the value of the SQLOBS macro variable depends on whether an output table, single macro variable, macro variable list, or macro variable range is created:

- If an output table is created, then SQLOBS contains the number of rows in the output table.
- If no data set or table, macro variable list, or macro variable range is created, then SQLOBS contains the value 1.
- If a single macro variable is created, then SQLOBS contains the value 1.
- If a macro variable list or macro variable range is created, then SQLOBS contains the number of rows that are processed to create the macro variable list or range.
- If an SQL view is created, then SQLOBS contains the value 0.

SQLOOPS contains the number of iterations that the inner loop of PROC SQL processes. The number of iterations increases proportionally with the complexity of the query. This value can be used to somewhat control inefficient processing when it is used to set a value for the PROC SQL LOOPS= option. Details though are beyond the scope of this paper.

SQLEXITCODE contains the highest return code that occurred from SQL insert operations. This return code is written to the SYSERR macro variable when PROC SQL terminates. The default value is 0 when there are no insert operations.

AUTOMATIC MACRO VARIABLES FOR RELATIONAL DATABASES

PROC SQL produces two more automatic macro variables when SAS queries a relational database such as Oracle or Teradata—SQLXRC, and SQLXMSG. It is important to remember that both are available only with the Pass-Through Facility—they are not produced when the DMBS is queried via a LIBNAME engine. If you're not using a CONNECT TO statement, they're not available.

SQLXRC contains a database-specific return code for the query. An Oracle code of 43568 is probably not the same as the Teradata 43568.

SQLXMSG provides descriptive information for the DBMS-specific return code. Again, each DBMS has its own idiosyncrasies and some messages are better than others. But because each DBMS has a 'format' for their messages you can scan or parse them to find specific information, such as the phrase 'invalid identifier'. This lets your error trapping routines be a bit more robust and database agnostic.

The value of the SQLXMSG macro variable can contain special characters such as &, %, /, *, and ; so SAS

recommends using the %SUPERQ macro function when working with SQLXMSG content.

```
%let dbErrorMsg=%superq(sqlxmsg);
```

PUTTING SQL RETURN CODES TO WORK

CAPTURING RETURN CODES

Return codes and error messages are reset after each query so it's important to capture them with `%LET` to create a user-defined macro variable each time you might want to use the value. In some cases you might want to recycle the user-defined macro variable name to simplify error handling routines or you can create a set of uniquely named variables with each query.

In the example below, we're working with a LIBREF that uses the Oracle engine and we have a PROC SQL with two queries and the second query fails. For each query we're capturing the SQLRC value it returns and we're writing the values to the log so we see the query failed with rc=8, a non-specific error.

```
115  libname userTabs oracle path='STAGE' schema='JBENT' user="&_hemiDbUser"
pass="&_hemiDbPwd" readbuff=5000;
117  proc sql noprint threads;
118     select count(*) as numRecs
119     from userTabs.dqf_results;
120
121     %let _sql_rc_rec_count=&sqlrc;
122
123     select avg(col_recrd_ct) as avg
124     from userTabs.dqf_results;
ERROR: The AVG summary function requires a numeric argument.
ERROR: The following columns were not found in the contributing tables:
col_recrd_ct.
NOTE: PROC SQL set option NOEXEC and will continue to check the syntax of
statements.
125
126     %let _sql_rc_avg_cols=&sqlrc;
127  quit;
NOTE: The SAS System stopped processing this step because of errors. <snip>
128  %put _sql_rc_rec_count= &_sql_rc_rec_count;
_sql_rc_rec_count= 0
131  %put _sql_rc_avg_cols= &_sql_rc_avg_cols;
_sql_rc_avg_cols= 8
```

This next example is the same query as above but we're using SQL Pass-Through to query an Oracle table. Notice that even though the log shows a message from Oracle the SQLRC remains the same. Why? Because it's the SAS return code. We don't yet see the Oracle return code, but notice how nice the Oracle message is? So how do we capture the Oracle return code and error message so we can put them to work?

```
272  proc sql threads;
273  connect to oracle as orc (path='STAGE' user="&_hemiDbUser" pass="&_hemiDbPwd"
readbuff=5000);
275     select * from connection to orc
276     (select count(*) as numRecs from jbent.dqf_results);
277     %let _sqlrc_rec_count=&sqlrc;
278
279     select * from connection to orc
280     (select avg(col_recrd_ct) as avg from jbent.dqf_results);
ERROR: ORACLE prepare error: ORA-00904: "COL_RECRD_CT": invalid identifier. SQL
statement: select avg(col_recrd_ct) as avg from
       jbent.dqf_results.
NOTE: PROC SQL set option NOEXEC and will continue to check the syntax of
statements.
281     %let _sqlrc_avg_cols=&sqlrc;
282     disconnect from orc;
283  quit;
NOTE: The SAS System stopped processing this step because of errors. <snip>
284
285  %put _sql_rc_rec_count= &_sqlrc_rec_count;
_sql_rc_rec_count= 0
286  %put _sql_rc_avg_cols= &_sqlrc_avg_cols;
_sql_rc_avg_cols= 8
```

Here we capture and display the Oracle return code and error message contained in SQLXRC and SQLXMSG. Notice that the Oracle error message starts with the return code—it always starts in column position 4 and if there are no errors the value is zero. That makes it easy to substring out the error message itself, and if you don't mind substringing you don't might not need the SQLXRC.

```
322  proc sql threads;
323     connect to oracle as orc (path='STAGE' user="&_hemiDbUser"
pass="&_hemiDbPwd" readbuff=5000);
324
325     select * from connection to orc
326     (select count(*) as numRecs from jbent.dqf_results);
327
328     select * from connection to orc
329     (select avg(col_recrd_ct) as avg from jbent.dqf_results);
ERROR: ORACLE prepare error: ORA-00904: "COL_RECRD_CT": invalid identifier. SQL
statement: select avg(col_recrd_ct) as avg from
       jbent.dqf_results.
NOTE: PROC SQL set option NOEXEC and will continue to check the syntax of
statements.
330
331     %let _sqlrc_avg_cols=&sqlrc;
332     %let _sql_xrc_avg_cols=&sqlxrc;
333     %let _sql_xmsg_avg_cols=%nrbquote(&sqlxmsg);
334  quit;
NOTE: The SAS System stopped processing this step because of errors. <snip>
335
336  %put _sql_rc_avg_cols= &_sqlrc_avg_cols;
_sql_rc_avg_cols= 8
337  %put _sql_xrc_avg_cols=&_sql_xrc_avg_cols;
_sql_xrc_avg_cols=-904
338  %put _sql_xmsg_avg_cols=&_sql_xmsg_avg_cols;
_sql_xmsg_avg_cols=ORA-00904: "COL_RECRD_CT": invalid identifier
```

LEVERAGING RETURN CODES AND ERROR MESSAGES

Now that we know how to capture the SQL return code and error message from the database, using them is simply a matter of deciding what to do when an a query fails and then coding it up. It's easy to create a dummy table that contains a few good records and a bad record for each condition you want to test.

In production jobs, it's we often want to stop processing when an error or exception is encountered so we don't generate a log full of cascading errors, create a bunch of bad data sets or tables, or just waste computer resources. The macro `%abort_if_error` uses the ABORT statement to check and maybe kill a SAS job running on a UNIX server. If it kills the job it also sends an email alert that something bad happened.

```
%macro abort_if_error();
   %if &_sql_xrc ne 0 %then %do;
      %if &_testRun=Y %then %do;
         X mailx -r DONOTREPLY@ECOMM.COM -s "3 AM Load Processing Error, TEST Run,
&_tabSchema..&_tabName, cycle &_cycle" &_emailDeveloper <
%lowcase(&_reportDir/dqf_processing_error_&_tabSchema._&_tabName._&_cycle..txt);
      %end;
      %else %do;
         X mailx -r DONOTREPLY@ECOMM.COM -s "3 AM Load Processing Error,
&_tabSchema..&_tabName, cycle &_cycle" &_emailRecipientList <
%lowcase(&_reportDir/dqf_processing_error_&_tabSchema._&_tabName._&_cycle..txt);


         %** Dump all the macro variables to the log, if needs restart.;
         data dataDir.dump_macro_vars_&_timeStamp; set sashelp.vmacro;
         run;


         %** Now kill the job. ;
         data _null_;
            ABORT RETURN;
         run;
      %end;
   %end;
%mend;
```

At the start of processing, macro variables are assigned that contain the directory holding a log produced by Oracle and a list of people to be emailed. &CYCLE is a macro variable derived from the calendar date. During processing two more macro variables are assigned that hold the names of the schema and table being processed.

In this particular job, each query we want to monitor is a separate PROC SQL and we place the macro immediately after each query. In every query the names of the user-assigned macro variables are recycled.

```
proc SQL threads cpucount=4;
  connect to oracle …
   create table name as
   select …  from …  where… ;
  %let _sql_xrc= &sqlxrc;
  %let _sql_xmsg= %nrbquote(&sqlxmsg);
quit;

%abort_if_error;
```

Instead of simply killing the job, a more robust option might be to write a macro that parses SQLXMSG and then uses conditional execution based on the message content to implement an immediate work-around. If the work-around is implemented you'll still want to also send an email alerting people what happened.

- if SQLXMSG contains is 'table not available', use a different table.
- if SQLXMSG contains is 'table locked', use the SLEEP function to wait and try again later.

USING SQLOBS

As with most things SAS, there are multiple ways to accomplish a task. Everyone knows a couple different ways to find the number of records in a data set. Here's another way… use SQLOBS. But remember that there are a few considerations for using it.

Here we are using SQL Pass-Through to create a SAS dataset. At the same time we write the value of SQLOBS to a user-defined macro variable using the same technique as with capturing the rc and error message. Notice that because SQLXRC=0 (no error) the value of SQLXMSG is blank.

```
51   proc sql threads;
52      connect to oracle as orc (path='STAGE' user="&_hemiDbUser"
pass="&_hemiDbPwd" readbuff=5000);
54      create table current_cycle as
55         select * from connection to orc
                 (select * from jbent.dqf_results where cycle_cd='W201');
NOTE: Table WORK.W201 created, with 465 rows and 11 columns.
60         %let _nrecs_w201= &sqlobs;
61         %let _sql_xrc_w201=&sqlxrc;
62         %let _sql_xmsg_w201=%nrbquote(&sqlxmsg);
63      disconnect from orc;
64   quit;
NOTE: PROCEDURE SQL used (Total process time): <snip>
65
66   %put _sql_xrc_w201=&_sql_xrc_w201;
_sql_xrc_w201=0
67   %put _sql_xmsg_w201=&_sql_xmsg_w201;
_sql_xmsg_w201=
68 %put _nrecs_w201= &_nrecs_w201;
_nrecs_w201= 465
```

SQLOBS will be zero 0 whenever an empty data set is created, even when there's no error. But so will SQLXRC so that's an easy start for writing code to check for empty data sets if that's a concern during processing.

```
35   proc sql threads;
36      connect to oracle as orc (path='WFPROD' user="&_hemiDbUser"
pass="&_hemiDbPwd" readbuff=5000);
38      create table w201 as
39         select * from connection to orc
40         (select * from jbent.dqf_results where cycle_cd='W210');
NOTE: Table WORK.W201 created, with 0 rows and 11 columns.
44         %let _nrecs_w210= &sqlobs;
45         %let _sql_xrc_w210=&sqlxrc;
46         %let _sql_xmsg_w210=%nrbquote(&sqlxmsg);
47      disconnect from orc;
48   quit;
NOTE: PROCEDURE SQL used (Total process time): <snip>
49
50   %put _sql_xrc_w210=&_sql_xrc_w210;
_sql_xrc_w210=0
51   %put _sql_xmsg_w210=&_sql_xmsg_w210;
_sql_xmsg_w210=
52   %put _nrecs_w210= &_nrecs_w210;
_nrecs_w210= 0
```

SQLOBS is SAS-generated so it works great when creating a SAS dataset or printing output. But using SQL Pass-Through changes SQLOBS's behavior. When we use explicit pass-through to pass code directly to the database for execution, SQLOBS is always zero. SAS doesn't touch the results set so it doesn't have an opportunity to count the number of rows.

In this code sample we see that SQLOBS=0 even though SQLXRC=0. Is the dataset really empty? Maybe the `WHERE` criteria returned zero rows but we don't know that. How do we find out?

```
117  proc sql threads;
     connect to oracle as orc (path='STAGE' user="&_hemiDbUser" pass="&_hemiDbPwd"
readbuff=5000);
    execute (
121     create table current_cycle as
122        select * from jbent.dqf_results where cycle_cd='W210'
123   ) by orc;
125   %let _nrecs_w210= &sqlobs;
126   %let _sql_xrc_w210=&sqlxrc;
127   %let _sql_xmsg_w210=%nrbquote(&sqlxmsg);
128   disconnect from orc;
129   quit;
NOTE: PROCEDURE SQL used (Total process time):<snip>
130
131   %put _sql_xrc_w210=&_sql_xrc_w210;
_sql_xrc_w210=0
132   %put _sql_xmsg_w210=&_sql_xmsg_w210;
_sql_xmsg_w210=
133   %put _nrecs_w210= &_nrecs_w210;
_nrecs_w210= 0
```

As with all things SAS, there's more than one way to solve the problem. The most obvious work-around is to go back to the old faithful `select count(*)` in an implicit SQL Pass-Through construct. Then we write the number of records to a user-defined macro variable so we can do something with it.

```
117  proc sql threads;
     connect to oracle as orc (path='STAGE' user="&_hemiDbUser" pass="&_hemiDbPwd"
readbuff=5000);
    execute (
121     create table current_cycle as
122        select * from jbent.dqf_results where cycle_cd='W210'
123   ) by orc;
125
126   %let _sql_xrc_w210=&sqlxrc;
127   %let _sql_xmsg_w210=%nrbquote(&sqlxmsg);
128
129   Select numrecs
130   into : _nrecs_w210
131   from connection to oracle
132      (select count(*) as numRecs
133       from current_cycle);
134
135   disconnect from orc;
136   quit;
NOTE: PROCEDURE SQL used (Total process time):<snip>
137
138   %put _sql_xrc_w210=&_sql_xrc_w210;
_sql_xrc_w210=0
139   %put _sql_xmsg_w210=&_sql_xmsg_w210;
_sql_xmsg_w210=
140   %put _nrecs_w210= &_nrecs_w210;
_nrecs_w210= 465
```

CONCLUSION

The return codes and error messages produced when PROC SQL queries a relational database can be powerful tools. With just a little bit of work you can leverage them to make a program respond to errors by stopping a failed program immediately or implementing an automatic work-around when an error is thrown. They're not silver bullets, but with proper planning they can help you lessen the impact caused by problems whose solutions are out of your control.

REFERENCES

SAS 9.2 Macro Language Reference. Cary, NC: SAS Institute, Inc
http://support.sas.com/documentation/cdl/en/mcrolref/61885/PDF/default/mcrolref.pdf

SAS/ACCESS® 9.2 for Relational Databases: Reference, Fourth Edition. Cary, NC: SAS Institute, Inc.
http://support.sas.com/documentation/cdl/en/acreldb/63647/HTML/default/viewer.htm#a000386105.htm

Carpenter, Art. 2004. *Carpenter's Complete Guide to the SAS® Macro Language, 2nd Ed*. Cary, NC: SAS Institute, Inc.

Lafler, Kirk. 2004. *PROC SQL: Beyond the Basics Using SAS®*. Cary, NC: SAS Institute, Inc.

CONTACT INFORMATION

John E. Bentley
Wells Fargo Bank
John.bentley@wellsfargo.com

The opinions expressed here are the view of the author and do not necessarily reflect the views and opinions of Wells Fargo Bank. Wells Fargo Bank is not, by means of this article, providing technical, business, or other professional advice or services and is not endorsing any of the software, techniques, approaches, or solutions presented herein. This article is not a substitute for professional advice or services and should not be used as a basis for decisions that could impact your business.