Paper 141-2009

Avoiding Common Traps When Accessing RDBMS Data

Mike Rhoads, Westat, Rockville, MD

ABSTRACT

Accessing data from databases such as Oracle and SQL Server has become an increasingly common requirement for SAS® programmers. SAS offers powerful and flexible capabilities for accessing such data, and many aspects of this process are quite straightforward, especially when LIBNAME engines are used. However, there are a number of traps that lurk for the unwary that can potentially produce unexpected error messages, incorrect output, or frantic calls from your normally-friendly Database Administrator. This paper highlights some of the most common traps and explains how to avoid them.

INTRODUCTION

When I first started programming in SAS, we read our data in from "flat files" (after entering our programs onto punched cards, of course). We had to worry about what column each field was in, and we might wind up making up our own variable names, but life was reasonably good.

As time passed, it became increasingly common to store important data in relational database management systems, or RDBMS. Our friends at SAS soon realized the need to bring such data into SAS, so the SAS/ACCESS® family of products was born. While early versions of SAS/ACCESS did allow us to bring data from sources such as Oracle and DB2 into SAS, it did not exactly make this easy. For each table we had to create "access descriptors" and then "view descriptors": in fact, the whole process could best be described as a pain in the descriptor. Life was OK – we could at least get to the data – but it certainly could have been better.

But the SAS development team certainly did not sit on their hands (although perhaps on their descriptors). The maximum length of SAS variable names expanded from 8 characters to 32 characters, meaning that in most cases RDBMS column names could be used as SAS variable names without alteration. Another notable improvement was the introduction of LIBNAME engines for the RDBMS that were supported by SAS. This meant that data could be retrieved from any supported RDBMS through simple queries in PROC SQL, without having to know the exact SQL dialect used by the target RDBMS (although if you were familiar with the native SQL, you had your choice of using that in a pass-through query). Even better, you could reference RDBMS tables directly in other SAS procedures and DATA steps, which made it possible to analyze such data without knowing any SQL whatsoever:

```
proc univariate  data=myoralib.temperatures;
  by month;
  var MaxDailyTemperature MinDailyTemperature;
  run;
```

Now life was very good – maybe a little too good. For it was now so easy and convenient to bring in RDBMS data, that it was also easy to believe that there were no differences at all between SAS data and behavior, and that of an RDBMS. The similarities, in fact, were so great that we tended to completely forget about the differences – until one of them raised its ugly head to bite us in the descriptor.

The purpose of this paper is to make you aware of some of these key differences, so that you can avoid or work around them. We will focus on 3 major areas:

- Errors and other unexpected output,
- Plausible (but incorrect) output,
- Basic efficiency tips.

The examples in this paper were run using SAS 9.1.3 (under WindowsXP Professional) and SQL Server 2005. Unless otherwise noted, however, the issues discussed apply equally well to other RDBMS systems and SAS platforms.

BEFORE WE GET STARTED

This paper is not intended to be a comprehensive introduction to SAS/ACCESS software – for that, see Rhodes (2007) or the SAS OnlineDoc® documentation. However, there are a few basic points that may help you to understand how the software works and the traps that you may encounter.

In order to communicate with a relational database, you first need to establish a *connection* with it. Once that has been done, you can issue commands to the database to insert and retrieve data, or to perform more administrative tasks such as inserting database tables or modifying their structure. Relational databases use Structured Query Language (SQL) as the primary mechanism for issuing such commands to the database. The basics of SQL are standardized so that they work

regardless of the specific database involved, although nearly all relational databases also support specific extensions to the SQL standard, so that a command that works, say, in Oracle may not be understood by SQL Server.

Databases also provide *security* for the data that they contain. You will almost always need to provide a username and password, either directly or indirectly, in order to connect to a database. (Some databases may coordinate with the operating system on your workstation, so that if you have already authenticated yourself to log on to the workstation and corporate network, it takes advantage of those credentials rather than requiring a second logon.) Once you have connected to your database software, it controls whether or not you have access to individual databases that it contains, and it may also limit your access to specific database *objects*, such as tables and views. Even if you have access to a specific database object, the database may limit the types of actions you can perform on it. For instance, you may be able to read a table that has a history of customer transactions but may not be allowed to change or add to the data in the table. Or, you may be able to both read from and write to a specific table, but may not be allowed to change its structure or delete it. Typically only the database administrator can perform such structural changes.

There is really no “magic” about the way that SAS/ACCESS interfaces work. As described above, SAS/ACCESS establishes a connection to the database, issues SQL commands to retrieve data or perform other actions, and then ends the database connection when it is no longer needed. All facets of database security are maintained: it must pass the same sort of authentication credentials to the database as would be required if SAS were not involved in the process, and it cannot enable you to perform any data retrieval or modification activities for which you are not authorized.

SAS uses two primary mechanisms for implementing its access to RDBMS: the Pass-Through Facility and the LIBNAME statement. (Earlier versions of SAS implemented a separate method using the ACCESS and DBLOAD procedures. This access mechanism is still supported to provide compatibility with older programs, but its use is no longer recommended.)

The Pass-Through Facility

The Pass-Through Facility is the older of these two mechanisms. To begin using it, you issue a CONNECT statement within PROC SQL, which might look something like the following:

```
Connect to oledb as myTest (
  provider=sqlOleDb
  dataSource=YourServerName
  properties=(
    "Integrated Security"=SSPI
    "Initial Catalog"=YourDBName
  )
) ;
```

“Oledb” identifies the specific SAS/ACCESS interface involved, and “myTest” is a name that you assign so that you can refer to the connection later. The remaining information needed depends on the SAS/ACCESS product involved (here, SAS/ACCESS to OLE DB), your installation, and the specific database involved. In this example, “provider=sqlOleDb” indicates that we are using the OLE DB interface to Microsoft SQL Server, “dataSource” identifies the specific database server, “"Integrated Security"=SSPI” denotes that a separate login is not required, and “Initial Catalog” points to our specific database.

Once you have connected to the database, you use the CONNECTION TO construct in SELECT statements to issue SQL queries directly to the database. For instance:

```
select * from connection to myTest (
      select
        TrackingID,
        QueryID
      from
        dbo.MyDatabaseTableName
      order by
        TrackingID ASC,
        QueryID ASC
);
```

In this example, “connection to myTest” refers back to the database connection that you just established. It is important to note that the inner SELECT statement (all of the text within the parentheses) is sent directly to the underlying database as-is. Thus, this part of the statement can use SQL extensions that are specific to the underlying database, but it cannot use SQL features that are specific to SAS, such as data set options or SAS-specific functions.

You can also use the EXECUTE statement to send SQL statements other than SELECT to the database. Again, the parenthesized text goes directly to the database and must utilize its underlying syntax.

```
execute
  (insert into dbo.MyDatabaseTableName values (24582,12) )
    by myTest;
```

Finally, you end the connection by issuing a DISCONNECT statement. (If you omit DISCONNECT, the connection will be ended automatically when you exit from PROC SQL.)

The LIBNAME Statement

The Pass-Through Facility that we discussed above is the SAS/ACCESS equivalent of a manual transmission on a car. You need to explicitly specify each step of the process (connecting to the database, retrieving data, disconnecting), and you need to use SQL syntax that the underlying database will understand.

In contrast, using the LIBNAME statement to access your database is like an automatic transmission. Just issue a LIBNAME statement with the appropriate options (put the car in Drive), and you are off and running.

Using this method, you identify the database and specify all necessary connection information within the LIBNAME statement itself. For example:

```
libname  myTest  oledb
  provider=sqlOleDb
  dataSource=YourServerName
  properties=(
    "Integrated Security"=SSPI
    "Initial Catalog"=YourDBName
  )
  ;
```

Notice how similar this syntax is to the CONNECT statement we illustrated earlier. When you use the LIBNAME statement, however, everything else is taken care of for you. SAS automatically connects to the database and makes its objects available to you to the extent that your database permissions allow. Once you have done this, you can access the database's tables and views however you reference ordinary SAS data sets, using the familiar *libname.tablename* syntax. For example, to print two variables from the MyDatabaseTableName table in the previously-referenced SQL Server database:

```
proc print  data=myTest.MyDatabaseTableName;
var TrackingID QueryID;
run;
```

Comparing the two access methods

Neither the Pass-Through Facility nor the LIBNAME statement is inherently superior to the other for database access. There are reasons why you may prefer one to the other, however:

- All of the statements for the Pass-Through Facility are part of PROC SQL, which makes its use more suitable for those who are comfortable within the SQL environment. While it is possible to access database objects from the DATA step and procedures other than SQL with the Pass-Through Facility, you need to create a PROC SQL view in order to do so. In contrast, with the LIBNAME statement it is extremely easy to access as many database tables as you need from within any DATA step or procedure.
- As discussed above, the Pass-Through Facility requires you to know and use the SQL syntax of the underlying database. This can be an advantage if you are already familiar with the database. For instance, if you use a database client tool, such as SQL Server Management Studio, you can develop and test your SQL queries with that tool, and then later embed them within the CONNECTION TO clause of a PROC SQL SELECT statement. There may also be situations where you need to use a function or construct that is specific to the database. On the other hand, if you are not familiar with the database's idiosyncrasies, you are probably better off sticking with the LIBNAME statement.
- When SAS first implemented the LIBNAME method for accessing relational database, it was not uncommon for its performance to be inferior to that which you could obtain using the Pass-Through Facility. Over the years, however, SAS has significantly improved its SQL optimization, so this is no longer necessarily the case. If you are working with large database tables where performance may be a problem, it is a good idea to try both techniques to see whether there is a significant difference.

Debugging programs that access an RDBMS

One of your most valuable tools when trying to figure out what's really going on when you access RDBMS data is to be able to see exactly what instructions SAS is sending to the database. SAS won't tell you this by default, but you can get this information by including the OPTIONS statement below. Try it – it may become your best friend.

```
options sastrace=',,,d' sastraceloc=saslog nostsuffix;
```

EXPECT THE UNEXPECTED

Although SAS has come a long way in making SAS access to RDBMS data as transparent and pain-free as possible, SAS veterans who are newcomers to the SAS/ACCESS world may encounter some surprises along the way. This section discusses the most common such situations and provides tips on how to handle (or avoid) them.

So where are my tables?

You have successfully issued a LIBNAME statement to connect to your RDBMS and are all ready to start running the reports the boss asked for yesterday. Just to see what the database looks like from the SAS point of view, you decide to run PROC CONTENTS – and get the following message in the log:

```
WARNING: No matching members in directory.
```

Or perhaps your symptoms are slightly different – you do get PROC CONTENTS output, but only some of the tables that you know are there (and are permitted to access) show up.

Or, you may try to read a specific RDBMS table or view in a DATA step or procedure, and come up with the following:

```
ERROR: File MYDB.MyTable.DATA does not exist.
```

In all likelihood, the RDBMS "schema" is causing your difficulties. C. J. Date defines schema as "that piece of the database that is owned by some specific user." For the purposes of this paper, you can think of it as a logical subsection of the database, containing a set of database objects such as tables and views. This means that database objects are normally accessed within the RDBMS using a two-level reference consisting of the schema name and the object name (unless an appropriate default schema name is involved).

So, what happens when you use a LIBNAME to connect to an RDBMS? Basically, some default value for schema is used to determine what database objects that connection can "see". If you are lucky enough for that default to cover all of the tables and views you need, all will be well. If not, attempting to get an overview of the database using PROC CONTENTS, dictionary tables, or the SAS Explorer window will only show objects within the default to which you have access (which may be none). And trying to access a specific RDBMS table in a DATA step or procedure will work only if its schema corresponds to the default, even if you have rights to access the table.

Fortunately, this problem is easy to fix once you know how. Just find out the schema name of the objects that you need to access, and specify it in the SCHEMA= option of the LIBNAME statement:

```
libname  mydb  oledb
  provider=sqloledb
  datasource=myservername
  properties=( "initial catalog"=mydbname )
  prompt=yes
  schema=myschemaname
;
```

What if, in the same SAS program, you need to access database objects that live in more than one schema? There are two approaches you can take. One is to issue a separate LIBNAME statement for each schema, and then use the corresponding libname whenever you reference an RDBMS table or view, depending on which schema it belongs to. The second approach is to use the SCHEMA= data set option whenever you need to access an object that is not in the schema you specified on the LIBNAME statement. For instance, if you needed to access the "SpecialTable" table that is in the "schema2" schema of your database, you could do so using the LIBNAME statement above by specifying the following:

```
proc print  data=mydb.SpecialTable (SCHEMA=schema2);
```

There is actually another situation that might cause perfectly good DBMS tables to be "invisible" to SAS. The naming restrictions that SAS has for data sets are almost certainly different from the rules that your RDBMS software uses for table names. Your database probably allows longer names that the SAS maximum of 32 characters; it may also allow a wider variety of characters to appear in the name.

The character-set issue is not a big problem. Depending on the value of the VALIDVARNAME system option, SAS will either translate the characters it considers invalid to those that it accepts (VALIDVARNAME=V7), or will leave them alone (VALIDVARNAME=ANY). In the latter case, you can use a SAS "name literal" to specify the name:

```
data newfile;
set  dbmslib.'Table With Embedded Blanks'N;
```

There is no corresponding technique for dealing with table names that are longer than 32 characters: these will not show up in SAS Explorer or in PROC CONTENTS, and cannot be accessed directly in SAS procedures or DATA steps. The only way to access them is by passing native SQL through to your database through the PROC SQL pass-through facility. If you

frequently need to access one or two of these tables, you might want to consider setting up a view for each one, either in SAS or within your DBMS, and giving the view a SAS-compatible name.

Why won't it let me do that?

Some of your SAS programs may contain code like the following to reorder a permanent SAS data set:

```
proc sort  data=mydb2.class;
by height weight;
run;
```

Or, perhaps you are trying to add some new variables, rather than changing the order of the records:

```
data mydb2.class;
set  mydb2.class;
wh_ratio = weight / height;
run;
```

Either of these techniques works fine with SAS data sets. However, if the mydb2 libref is an RDBMS connection rather than a reference to a native SAS data set library, you can expect to get the following lengthy but somewhat cryptic message:

```
ERROR: The OLEDB table class has been opened for OUTPUT. This table already exists, or
there is a name conflict with an existing object. This table will not be replaced. This
engine does not support the REPLACE option.
```

Another situation where you may get this message is if you have a job that creates a table in your RDBMS, and you run the program a second time.

The last sentence of the error message is really the key. Experienced SAS users have come to take for granted the ability to replace SAS data sets "on the fly" – that is, without explicitly deleting them first. SAS manages this process behind the scenes, initially writing the records for the new incarnation of the data set to a separate file with a temporary name, and then at the end (if all goes well with the previous processing) deleting the old instance of the data set and renaming the new data set to have the same name as the previous version. So, you really wind up with a completely "new" data set that has the same name as the original one.

Relational databases don't work this way: creating a table is normally a long-term commitment, with changes made within the table, rather than by creating a new instance of the table. It's much more of a structured environment, which certainly isn't a bad thing. Keep in mind the most RDBMS operate in a multiuser environment, and other users may not appreciate some of its tables disappearing (even if briefly), or undergoing major structural changes.

As SAS programmers, we need to adjust to this reality. "Thinking like the database" is normally a better approach than trying to work at cross-purposes with it. The following tips should prove to be helpful.

Consider using an "in-place" strategy to work with records within an existing database table. Probably the easiest approach is to use the DELETE, INSERT, and UPDATE statements within PROC SQL to make the desired changes to the table rows (records). However, you can now also do similar processing within a DATA step, using the MODIFY, REMOVE, REPLACE, and OUTPUT statements.

One characteristic that relational databases share with SAS data sets is that neither allows changes to the structure of a table (adding, dropping, or changing the data type of a column) on the fly. As noted above, SAS pretends to do this, but it actually is creating a new instance of the table, which necessitates rereading and writing all of the rows of the table. RDBMS are actually somewhat more flexible in this regard, in that you can make some changes (such as adding new columns) without having to reprocess the existing records. The ALTER TABLE statement in PROC SQL is the way to make such structural changes (assuming that you have sufficient privileges within the RDBMS to do this, of course). Once you have created a new variable (say, wh_ratio), you can then populate it for existing records using either PROC SQL or a DATA step with the MODIFY statement. Just don't expect to be able to both create and populate the new column at the same time.

If you have programs that write new DBMS tables, bear in mind that the table will have to be dropped (using DROP TABLE in PROC SQL) before the step writing out the table is executed, if the job has been run previously.

If necessary and appropriate, you can manually implement the traditional behind-the-scenes "replace on the fly" SAS logic in an RDBMS environment. Bear in mind the possible effects on other database users, and that you will need database rights to drop tables and create new ones. Begin by creating the new instance of the table with a temporary name. If there are no errors in this step, you would then get rid of the old instance of the table (using the DROP TABLE statement in PROC SQL), and finish by renaming the new table instance to the permanent name. This final step would need to be done with the PROC SQL pass-through facility or with the CHANGE statement in PROC DATASETS, since it is not supported by native SAS PROC SQL statements.

One final thing to keep in mind: your RDBMS adminstrator may have given you rights to read, but not write to, your corporate database. In such cases, expect to get a message similar to one of the following if one of your jobs tries to write something into the database:

```
ERROR: Error attempting to CREATE a DBMS table. ERROR: Execute error: ICommand::Execute
failed. : CREATE TABLE permission denied in database 'MyCorporateDB'.

ERROR: Execute error: IRowsetChange::DeleteRow failed. : Multiple-step OLE DB operation
generated errors. Check each OLE DB status value, if available. No work was done.: DELETE
permission denied on object 'MyTable', database 'MyCorporateDB', owner 'corp_db_admin'.

ERROR: File MyCorporateDB.MyTable.DATA is sequential.  This task requires reading
observations in a random order, but the engine allows only sequential access.
```

But I just sorted that blasted file!

On first glance, this is one of the most perplexing anomalies you will encounter when working with RDBMS data (or any other data, for that matter) in SAS. You begin by creating a sorted data set from one of your DBMS tables:

```
proc sort  data=dbmslib.CollationTest  out=SortedFile;
by CharVar;
run;
```

Perhaps you even run a PROC CONTENTS to verify that the sort (and everything else) worked correctly. Then, you innocently run some code that relies on the sort you just performed:

```
data _null_;
set  SortedFile  end=eof;
by   CharVar;
if eof then put 'Everything worked';
run;
```

What could possibly go wrong with that? You probably wouldn't expect the following message (even if you hadn't called it "SortedFile"):

```
ERROR: BY variables are not properly sorted on data set WORK.SORTEDFILE.
```

There is a small clue in one of the previous messages produced by the PROC SORT:

```
NOTE: Sorting was performed by the data source.
```

We old-timers tend to think of sorting as a simple, unambiguous task: just use the ASCII (or EBCDIC for IBM mainframes) collating sequence for character variables, and off you go. These days, however, sorting has become a much more complex subject – "linguistic collation" is the new catch phrase. These sophisticated algorithms, which reflect the increasingly multinational (and thus multilingual) world in which many enterprises operate, produce more natural character string ordering by keeping occurrences of the same letter together regardless of case and accents. It's even possible to sort embedded numbers within a character field by their numeric rather than character values.

Since over half of SAS Institute's revenues come from outside the Americas, their senior executives and product managers are well aware of these developments. SAS 9.2 marks a major leap forward in supporting advanced lingustic collation within SAS. This gives SAS programmers many more options when it comes to ordering data within SAS.

Unfortunately, even in SAS 9.2 the SAS/ACCESS products haven't yet caught up with the more sophisticated sorting capabilities that are built into Base SAS software. In particular, as our example above demonstrates, SAS/ACCESS does not recognize that the underlying RDBMS may be using a collation sequence that is different from the binary default assumed by SAS. The SQL Server database used in the database above uses the collation sequence "SQL_Latin1_General_CP1_CI_AS". The CI_AS suffix indicates that the sort order is case-insensitive but accent-sensitive. The case-insensitivity is an obvious difference from the default SAS sort processing. Nevertheless, SAS will hand the sort processing over to the RDBMS (which is generally more efficient), oblivious to the fact that the RDBMS may be using a different collating sequence.

Another wrinkle is how databases treat null values when sorting. In SAS, missing values (the closest native equivalent to database null values) sort low, before any non-missing values. Some RDBMS systems, such as Oracle and DB2, return records with null values last, rather than first, when sorting in ascending order (and the converse for a descending sort).

There are several ways to deal with these collation order conflicts. The first point to recognize is that it may not be critical that the data set is not sorted as you (or SAS) expect it to be. If all you are trying to do is produce report output in a logical order, the "incorrect" results produced by the RDBMS sort may be fine, and in fact may put records in a more logical order than the default SAS sort. For example, it might be useful to have the jam band "moe." come up between "Moby" and "Mountain" in a listing of musical artists, despite its insistence on using all lowercase characters in its name.

If you need to use BY-group processing to group your records for output or processing, you can use the NOTSORTED option. This will cause SAS to check whether the values of the sort key(s) have changed from one record to the next, thus ending

one group of records and starting another, but not to care whether the change is in the expected direction (ascending or descending).

If the precise sort order is essential – for instance, if you have to merge the data set with another one – you have two main options.

The first option is to force SAS to perform the sort. The most common way to do this is by specifying the system option SORTPGM=SAS. It may take SAS longer to do the sort, as opposed to letting the DBMS handle it, but at least the data set will come out in the order that SAS expects.

Note that in some instances, you may not even have an explicit PROC SORT or ORDER BY in your program: you may simply be using the database table directly in your SAS data step procedure with the assumption that it is "correctly" sorted. For instance, the variable in question may be a primary key for its database table. In such cases, you will need to make a separate copy of the data in SAS to get the rows in the order that SAS wants.

The second option is to induce the database to order the data the way that SAS expects it. Occasionally there may be an obscure SAS approach that will expedite this. For instance, if you are using SAS 9.1.3 and Oracle and your problem is with null values sorting high rather than low, specifying the statement below will cause Oracle to put nulls at the beginning, where SAS expects them. (In SAS 9.2, SAS/ACCESS for Oracle takes care of this automatically.)

```
options debug=ora_sortnulls_like_sas;
```

In the absence of such a trick that handles your specific situation, you may be able to use SQL pass-through and write DBMS-specific SQL that causes the DBMS to perform the sort in the way that SAS expects. In SQL Server, for instance, you can use the COLLATE clause with the Latin1_General_BIN collation to have the database order the data in a SAS-compatible manner:

```
create table CompatibleCollation as
  select * from connection to dbmsconn (
      select * from dbo.CollationTest
      order by CharVar COLLATE Latin1_General_BIN
  );
```

SAS Problem Note 16160 illustrates a technique to work around the null-ordering problem for DB2.

LISTING limitations

When you bring in character columns from an RDBMS, SAS associates a format and informat for each of them that is equal to the width of the column. For instance, if you have a variable LongField with a length of 2000, it will be assigned a format and informat of $2000.

For the most part, this doesn't matter. However, if you are still using the LISTING destination, be aware that the maximum number of characters that PROC PRINT can output for any column is slightly less than the value of the LINESIZE system option. Therefore, if you try to print such a wide column, you will get a message similar to the following:

```
WARNING: Data too long for column "LongField"; truncated to 116 characters to fit.
```

The presence of the format means that you will get the message even if none of the data values you are printing exceed this limit. In that case, you can eliminate the warning message by removing the format from the variable, either permanently or within the PROC PRINT step.

If some of your values do exceed this limit, assign a format in the PROC PRINT step that has a shorter value, e.g.:

```
FORMAT LongField $100.;
```

Your data will still be truncated, but you will at least avoid the message.

Better yet – use a more modern output destination such as HTML, RTF, or PDF, and avoid both the truncation and the warning message!

HOW COULD SOMETHING SO (APPARENTLY) RIGHT TURN OUT TO BE SO WRONG?

While the line above may sound like the title of the latest smash hit on the country music charts, it's actually a lament that is heard on occasion among SAS programmers. Unlike the previous section, where SAS error messages make it quite clear that something is not quite copacetic, here we talk about situations where there is no obvious warning of rough waters ahead. The first sign of trouble is when your boss, your client, or some other Very Important Person discovers that the data or reports that you provided were not completely accurate.

Clearly this is Not A Good Thing. This section will identify a few situations peculiar to RDBMS access where you may wind up with plausible but incorrect output.

Nothing Compares 2 U

You may remember that the pesky NULL (the absence of a value) can cause problems when ordering RDBMS data. Not surprisingly, this SQL concept can also cause problems in other ways which may not be immediately obvious.

The biggest issue is the treatment of NULL values in SQL comparisons. Any "standard" comparison (equality, inequality, less-than, greater-than) will not be satisfied when either of the operands has a null value. Let's assume you have a column called Profit in your DBMS table. If Profit has a null value and the RDBMS is evaluating comparisons, "Profit equals 0" will not be satisfied, which is not a big surprise. However, "Profit not equals 0" will also not be satisfied – neither will "Profit > 0" or "Profit < 0". In SAS, however, the "not equals" and "less than" conditions would be satisfied when a missing value is involved. The difference is that the SQL standard mandates ternary (three-valued) logic, where a comparison may have 3 possible results: True, False, and Unknown. Any comparisons involving null values produce an Unknown result, which will never satisfy a WHERE clause or other comparison that is evaluated by the database.

To avoid this problem, take advantage of the IS NULL and IS NOT NULL operators that are offered by all relational databases and supported by SAS in relevant contexts (PROC SQL, and WHERE clauses). Whenever you have a comparison involving an RDBMS column that may contain null values, specify either IS NULL or IS NOT NULL whenever necessary to avoid ambiguity. For example, to exclude null/missing values regardless of whether SAS or the RDBMS processes the comparison, specify:

```
where Profit < 0 and Profit is not null
```

On the other hand, if you do want to include null/missing values, specify:

```
where Profit < 0 or Profit is null
```

A related anomaly is when both values in an equality comparison are null/missing. When processed by the RDBMS, such comparisons will not be satisfied, since any comparison involving null values produces a result of Unknown (rather than True). In SAS, however, such comparisons would be satisfied in situations where each column holds a "standard" missing value.

Fred Levine's white paper (see References) provides an excellent discussion of these situations and others, including an outer join example where the problem is with generated null values rather than null values in the underlying tables.

All characters are not created equal

Ji (2003) discusses another possible pitfall when making comparisons involving RDBMS data, at least when Oracle is involved. (SQL Server 2005 did not exhibit this behavior.) Unlike SAS and some other database systems, Oracle does not pad the shorter operand with blanks when making comparisons involving varying-length character fields (VARCHAR2). This can result in failed comparisons when the text in such an Oracle field has trailing blanks at the end. The paper recommends using the SAS TRIMN function in situations where this might be a problem.

Not all there ...

Under some circumstances, extremely long text fields may be truncated when transferred from an RDBMS into SAS. (In SQL Server, long VARCHAR columns seem to come over correctly, but TEXT fields may be truncated.) In such cases, use the DBMAX_TEXT option on either your data set reference or on the associated LIBNAME statement. The default value is 1024, so keep an eye on your data when you have columns where the number of characters may exceed this limit.

Watch that date!

As you have probably discovered by now, SAS has only two fundamental data types: numeric and character. While SAS certainly provides wonderful functionality when dealing with date, time, and date-time data, it accomplishes this by storing these values in numeric variables, rather than by providing dedicated data types. The informat and/or format associated with the variable normally indicates when a variable is being used to hold such values.

RDBMS, on the other hand, typically have at least one dedicated data type for such values. SAS does a good job of accurately bringing these data over and assigning them appropriate informats and formats. However, depending on the RDBMS, the result may not be what you expect. Let's say your project has a variable called DateOfFirstVisit. Based on the variable name and your knowledge that it is used to track dates, you take it for granted that it holds a SAS date value (number of days since January 1, 1960) when it is brought over from SQL Server. However, while SQL Server has a data type (two, actually) to hold date-time values, it does not have a data type specifically for *dates*. Therefore, SAS naturally brings the data values over as date-time values. You will likely realize this if you take a look at the associated informat and format, but if you don't you will get surprising results when making comparisons, using date-specific formats for reporting, etc.

You can, if you like, convert the data to a more convenient representation as you bring it in from the RDBMS. For example, since you know that DateOfFirstVisit only holds dates, you may decide that you would like SAS to store it as a date value rather than a date-time value, particularly given the wider variety of built-in formats that SAS provides for dates. To do this, specify the DBSASTYPE data set option for the RDBMS table:

```
select ... from mydblib.MyTable (DBSASTYPE=(DateOfFirstVisit='DATE')) ... ;
```

QUICK TIPS ON EFFICIENCY

The relative ease and transparency with which SAS lets you access data from RDBMS tables makes it easy to overlook that there is a lot of work going on behind the scenes whenever this happens. As we have seen in the preceding sections, occasionally this can result in unexpected error messages and subtly inaccurate output if you don't know what to watch out for. It can also result in some types of requests taking a lot longer to run than they should. This not only affects your own productivity, but it also may have a negative impact on other database users. This section provides some tips on how to make your processing as efficient as possible when accessing RDBMS data.

Don’t extract unnecessary columns

This, of course, is a good rule for all situations, not just when accessing DBMS data. It is certainly worth keeping in mind in this arena, however. Probably the most-frequently-encountered culprit here is the use of SELECT * in PROC SQL, when in fact only a small subset of the variables from the table are of interest. A good trick to minimize typing while not bringing in unnecessary variables is to use the DROP= or KEEP= SAS data set options on the database table: SAS will use these to generate a SELECT statement containing only the desired columns. You can even use SAS variable lists with DROP and KEEP:

```
create table SubsetVars2 as
select * from dbmslib.class (drop=age--weight);
```

Avoid repeated large extracts

When large database tables are involved, it is tempting to access the data directly from SAS, rather than extracting the data into a native SAS data set. If you only need the data once in your program, this is a good approach. However, if you will be reading the same large DBMS table repeatedly in your program, you will generally do better to first create an extract file and then read that SAS data set in your subsequent steps (assuming you have sufficient disk storage on the SAS side, of course). Getting data from the RDBMS into SAS can be a relatively slow operation; you don't want to do it more often than necessary.

Consistency is also an issue: if your database table changes frequently, creating an extract means that all of the sections of your program that use that data will return consistent results. Depending on the relative importance of having consistent vs. up-to-the-minute data, you may even wind up extending this strategy by creating more permanent extract files on a regular basis (say daily or weekly), and having your other SAS programs read the extracts rather than the original RDBMS tables.

Make the database do the summarization

For many tasks, all you may need from a database table is summary counts, rather than all the individual records. The easist approach is to use one of the standard SAS reporting procedures, such as SUMMARY/MEANS, REPORT, TABULATE, or FREQ. However, when a large table is involved and you only need to summarize a few variables, it may be much quicker to use PROC SQL and have the database do the summarization for you. For instances, compare the two sets of code below, where Mytable has about 1.7 million rows:

```
/* Approach 1 */
proc summary  data=mydb.Mytable  nway  missing;
class sex;
output out=summout (drop=_TYPE_);
run;

/* Approach 2 */
proc sql;
create table sqlout as
select sex, count(*) as _FREQ_
  from mydb.Mytable
  group by sex
  order by sex;
quit;
```

PROC SUMMARY consistently took 10-11 seconds to run on my PC, while the PROC SQL approach took only 1 second.

Be careful of SAS-specific functions

One of the strengths of SAS is the richness of its language, such as its extensive variety of functions (well over 400 in SAS 9). You need to keep in mind, however, that the vast majority of functions in this repertoire mean nothing to most RDBMS software: therefore, SAS has to carry them out itself. This is not a big problem in many situations, but it can degrade performance significantly if, for instance, you are working with a huge table and are using a native SAS function in your WHERE clause as a filter. Although only a handful of records might satisfy the test in the filter, the RDBMS will have to return all of the rows to SAS so that SAS can process the function and evaluate the filter. As we know already, sending millions of records from the DBMS to SAS isn’t the best way to win friends and influence people.

As with summarization, your best strategy is to have the database do as much of the filtering as possible. In some cases, you may be able to use explicit pass-through within PROC SQL to use a DBMS function that will do most or all of the filtering you need. Consider the following example:

```
/* Approach 1 */
select
  height, weight, name
from dbmslib.class
  where INDEX(name,'-') > 0;

/* Approach 2 */
select * from connection to dbmsconn (
    select
      height, weight, name
    from dbo.class

      where CHARINDEX('-',name) > 0
);
```

The first approach retrieves all rows from the database and then has SAS apply the INDEX function, while the second uses the SQL Server CHARINDEX function to do the filtering on the database side. This would run much faster if the underlying table is large.

The heterogeneous join conundrum

Another very common task is to access a small subset of records from an extremely large database table, based on the value of an ID variable. If the table of desired ID values also exists within the RDBMS, the join will be done within the database, with only the matching rows returned to SAS, and all will be well. But what if the list of desired ID values exists within SAS, rather than somewhere in the database? Obviously the last thing we want, if we can possibly avoid it, is for the database to return millions of records to SAS, just so that SAS can do a join to identify the few dozen or few hundred records that it is interested in.

If you can send the desired IDs to the RDBMS somehow so that it can do the filtering, your performance is likely to be much faster. One possible approach is to create a temporary table in the database, load the desired ID values into it, and then use that table in your query. While this approach is supported by SAS and discussed in the SAS/ACCESS documentation, it will not work for you unless your user privileges in the database allow you to create temporary tables.

The other approach is to have SAS include the ID values directly in the WHERE clause of the query it generates. You can, of course, do this by hand by hard-coding the values into the query (or writing a macro to generate the desired code), but SAS also provides options to do this work for you. While a number of papers discuss the DBKEY and DBINDEX data set options, the SAS documentation suggests that the newer LIBNAME option, MULTI_DATASRC_OPT=IN_CLAUSE, may provide better performance in many cases.

```
libname  mydb  oledb
  provider=sqloledb
  datasource=myservername
  properties=( "initial catalog"=mydbname )
  prompt=yes
  schema=myschemaname
  multi_datasrc_opt = IN_CLAUSE
;
```

As its syntax suggests, this option causes SAS to place the ID values into an IN-clause, sending code to the RDBMS similar to that illustrated below. You do need to be aware that your DBMS may impose limits on the number of IN-clause values that it supports. The generated SQL code that is passed to the database when you use MULTI_DATASRC_OPT = IN_CLAUSE should look something like the following:

```
SELECT  "Name", "Sex", "Age", "Height", "Weight"  FROM "dbo"."class"  WHERE  ( ("Name" IN
( 'Alice' , 'Bob' , 'Carol' , 'Ted' ) ) )
```

The space-time continuum

If you are going to be extracting data from your RDBMS and storing the records in temporary or permanent SAS data sets, pay a little attention to the characteristics of your variables. It's not uncommon for databases to store significant amounts of text – names, addresses, comments, etc. Since they support varying-length as well as fixed-length character data types, they can do this efficiently, even for fields that often don't contain any data at all, or where there are significant length disparities between the shortest and longest values.

SAS only supports fixed-length character fields. This limitation can result in bloated data sets when text data of this sort are involved, since it will use the same number of bytes to store the field on each record, even if many of them have values that are much shorter than the maximum length. Not only does this waste disk space, it is also likely to slow the performance of your jobs. The time required to read and write data is often the most important factor in determining how long your programs will take to run.

Fortunately, SAS does provide options for compressing its data sets, which goes a long way to overcoming the absence of direct support for varying-length character fields. The COMPRESS=CHAR data set option can be highly effective in minimizing the storage required for data sets with long character fields. And, because the programs using such compressed data sets will have to do much less reading and writing to disk, they will probably run significantly faster, even though they have to do some extra work in uncompressing or compressing the records. (Keep in mind that this increased CPU time may affect the cost of your job if you are working in a Unix or mainframe environment where CPU time is the predominant factor in the chargeback algorithm used.)

CONCLUSION

Clearly, there is a long-term relationship between SAS and relational database systems. Although the relationship may have been somewhat tentative and rocky when it began, it has improved greatly over the years, and seems likely to get even better in the future.

In any long-term relationship, a little work and knowledge can go a long way towards assuring its success. Even with the most compatible spouse or partner, learning a few things about what to do (and what not to do) will significantly enhance the likelihood that the relationship will continue to grow and thrive. With any luck, avoiding the traps described in this paper will enable your relationship with SAS and RDBMS to survive and prosper.

DISCLAIMER: The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

REFERENCES

Date, C. J. 1989. *A Guide to the SQL Standard, Second Edition.* Reading, MA: Addison-Wesley Publishing Company.

Ji, Xinyu. "The Dark Side of the Transparent Moon – Tips, Tricks and Traps of Handling ORACLE Data Using SAS." *Proceedings of the Sixteenth Annual Northeast SAS Users Group Conference.* September 2003. <http://www.nesug.org/proceedings/nesug03/at/at004.pdf> (June 15, 2008).

Levine, Fred. "Potential Result Set Differences between Relational DBMSs and the SAS System." <http://support.sas.com/resources/papers/resultsets.pdf> (April 30, 2008).

Plemmons, Howard. "What's New in SAS/ACCESS®." *Proceedings of the SAS® Global Forum 2008 Conference.* March 2008. <http://support.sas.com/rnd/papers/sgf2008/access92.pdf> (June 15, 2008).

Rausch, Nancy A. and Nancy J. Wills. "Super Size It!!! Maximize the Performance of Your ETL Processes." *Proceedings of the SAS® Global Forum 2007 Conference.* April 2007. <http://www2.sas.com/proceedings/forum2007/108-2007.pdf> (June 15, 2008).

Rhodes, Dianne Louse. "Talking to Your RDBMS Using SAS/ACCESS." *Proceedings of the SAS® Global Forum 2007 Conference.* April 2007. <http://www2.sas.com/proceedings/forum2007/239-2007.pdf> (June 15, 2008).

SAS Institute Inc. 2007. *SAS/ACCESS® 9.1.3 for Relational Databases: Reference, Fifth Edition.* Cary, NC: SAS Institute Inc.

SAS Institute Inc. 2008. SAS Technical Paper. "Linguistic Collation: Everyone Can Get What they Expect." <http://support.sas.com/resources/papers/linguistic_collation.pdf> (June 16, 2008).

SAS Institute Inc. SAS Problem Note. "Problem Note 11734: New option added to the SAS/ACCESS Interface to Oracle to change the sort order of data." <http://support.sas.com/kb/11/734.html> (June 16, 2008).

SAS Institute Inc. SAS Problem Note. "Problem Note 16160: DB2 data containing missing/nulls is not sorted the same way in SAS as it is in the database and can yield incorrect results or error messages." <http://support.sas.com/kb/16/160.html> (June 16, 2008).

Whitcher, Mike. "New SAS® Performance Optimizations to Enhance Your SAS® Client and Solution Access to the Database." *Proceedings of the SAS® Global Forum 2008 Conference.* March 2008. <http://support.sas.com/resources/papers/sgf2008/optimization.pdf> (June 15, 2008).

ACKNOWLEDGMENTS

Thanks to Michael Raithel for his review and suggestions, numerous other Westat and SAS-L colleagues for uncovering many of these traps and workarounds, and to the indefatigable developers at SAS Institute for their constant efforts to make the software better.

CONTACT INFORMATION

Please contact the author if you have any questions or comments:

Mike Rhoads
Westat
1650 Research Blvd.
Rockville, MD 20850
MikeRhoads@Westat.com