MWSUG 2016 - Paper BB16

Solving Common PROC SQL Performance Killers when using ODBC

John Schmitz, Luminare Data LLC, Omaha, NE

ABSTRACT

The PROC SQL routine is commonly used by many SAS® programmers. However, poorly formed queries can create needless performance killers. This paper reviews common SQL coding practices observed in code reviews and provides simple alternatives that can dramatically reduce query run time and required resources. This paper will focus on techniques that should be comfortable to SQL programmers and yet can shave 75 percent or more off of query run times.

INTRODUCTION

Much has been and could be written on the topic of PROC SQL optimization. Many of the worst impacts occur when developers write code without considering or understanding the basic elements of efficient PROC SQL coding. Sometimes, it may be the rush to complete the urgent request. Others may be copying and paste of code from another program that may not be optimal for this scenario. Regardless of the cause, the result can generate serious degradation of job performance and, left unchecked, lead to overall system performance issues.

This paper draws from more than 20 years' experience reviewing and optimizing SAS code. These simple changes can dramatically impact query run time and require minimal effort to implement. A few assumptions made within this document:

- Examples will be for PROC SQL. Some of these scenarios apply to DATA steps as well, but the DATA step equivalent will not be discussed here.
- The examples assume source tables are being retrieved from a remote database (RDBMS), typically using SAS ACCESS/ODBC® or related access products. The specific access product should matter little, but it is important to note that many of the scenarios do not apply if the core source tables are SAS data sets.
- The scenarios presented are working and generally acceptable SQL code. The idea is to highlight elements specific to the SAS/SQL query connection running through PROC SQL.
- The system architecture has adequate resources to support properly configured processes. We will briefly talk about architecture, but it is not the focal point of this discussion.

Before diving too far into specifics of code, it seems appropriate to offer some high level system perspectives in which to frame the conversation. Clearly, one paper cannot cover every scenario. Rather, the idea is to provide a framework through examples so that developers themselves can identify what may be impacting their specific query and how they may best address the performance concerns.

KNOW YOUR STUFF (ARCHITECTURE AND SQL PASS-THROUGH OVERVIEW)

There are many variations in system architecture and each can impact query performance, especially when specific resources are strained. This section provides a very generalized overview of system architecture to establish a framework to discuss how query style impacts the overall process. Use these as rules of thumb; actual results may vary by system.

Figure 1 provides a simplistic view of a typical system architectural environment for SAS client/server that works in conjunction with remote database and remote file server components. This diagram depicts 4 primary hardware components:

- A SAS application server (where SAS code executes).
- RDBMS servers where SAS would access data via SAS/ACCESS.
- A file server where SAS may access network shared drives.

- An external drive array that provides the bulk of disc storage for all servers.

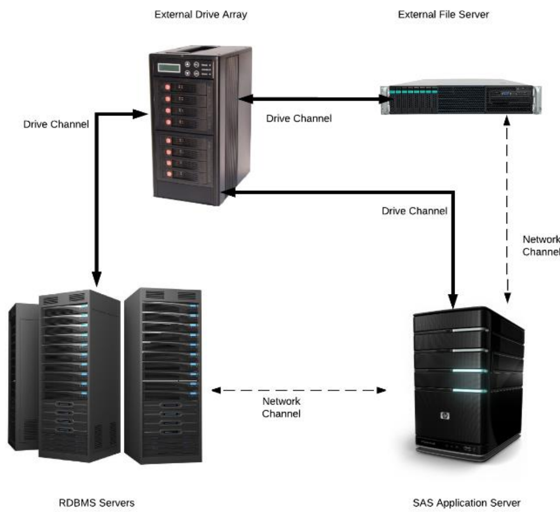

Figure 1 Typical SAS Architecture Environment

In this environment, servers communicate with their attached drives via drive channel connections. However, servers communicate with each other via network channel connections.

In general, the primary resources to consider include:

- Available physical memory (RAM)
- Virtual memory utilization and the associated page swap activity.
- CPU load.
- Drive I/O performance.
- Network (Ethernet) performance.

It is important to distinguish between physical and virtual memory. Virtual memory utilizes drive space to buffer memory when physical memory is inadequate for system needs. Excessive page swaps on virtual memory can be an indicator that physical memory limits are having a notable impact on performance.

The key element to recognize in this is the flow of data and information through the network. The fastest data activity occurs in physical memory. If the system hardware has adequate memory, there are simple steps that can be leveraged to utilize in memory processing. Drive channels are slower than memory but they are relied upon heavily within many environments. Network channels are typically much slower than drive channel. I have worked in environments where drive channels provided 50 to 100 times faster transfer rates than network connections. The goal in that environment clearly becomes one of reducing the amount of data moved over network connections. Remember, network connections include access to both the RDBMS systems and the remote file servers.

The system used for query processing is determined through SQL pass-through. In SQL pass-through, PROC SQL will send the query or subsets of it to the source RDBMS for execution. There are two types of pass-through to consider, EXPLICIT and IMPLICIT. Explicit pass-through occurs when the developer intentionally adds connection logic and includes native code for the source data system while implicit pass-through is done by SAS inside PROC SQL. In implicit pass-through, SAS generates code based on all or a portion of the SQL query and sends that code to the source system to process. In most cases, this results in faster query execution since it leverages the processing capacities of the source system

and normally reduces the data traffic over the network connection. For a more detailed discussion on pass-through logic, refer to the paper by Johnson in from last year's proceedings.

As a final note before wrapping up this section, most client/server systems resources are actively monitored and would rarely be the cause of consistent performance degradation. However, if performance issues arise sporadically, these resources may be reaching limits and restricting the query. Performance issues may arise from capacity limits on any resource on any hardware system involved. When investigating such potential issues, it is important to consider where the query processing occurs and the overall process load from all active SAS and non-SAS jobs. Developers facing resource-based performance issues should work closely with their SAS Platform Administrators and internal hardware support teams to identify and resolve the causes.

"KILLER" SQL QUERIES (AS IN THEY KILL PERFORMANCE)

With this framework in mind, it is time to turn attention to common query coding practices that can be performance killers in an RDBMS environment. These have been gleaned from reviewing thousands of process jobs and categorized into a few sample cases.

Before moving to examples, here are some coding conventions used below:

- A libref DBx refers to a library on a remote database using ACCESS/ODBC or similar
- A libref FSx refers to a library on a remote file server connection.
- A libref SDx refers to a library on a mounted drive to the SAS application server.
- Table names are generic LARGE_TABLEn, SMALL_TABLEn, etc.
- Table name TARGET is used for output and is assumed to be a SAS data set.
- Field names are generic, FIELDn for general fields, DATEn for a date field, DTn for a datetime field, KEYn for a table keys.

USING SELECT *

This comes up early on many people's list, but why, especially when you often need most of the fields anyway? Consider the simple example:

```
proc sql;
   create table sd1.target as
   select *
   from db1.large_table
   where date >= '01JAN2016'd;
quit;
```

The query selects all fields from the source table and all records beginning this year and writes them to a table named target on the SAS server. The concern over this approach primarily occurs when the fields you don't need are large character fields, perhaps names, addresses, descriptions, notes, or even unpopulated fields. These fields can often be lengthy strings. A 2000-byte string can generate the same connection traffic as 250 numeric fields. Once retrieved, the field is then included in output to TARGET and presumably included on the next read from TARGET in subsequent processing, generating further impact to subsequent processes.

Often this happens because developers balk at the need to type each field required to complete the query. Fortunately, SAS offers the FEEDBACK option in PROC SQL. This option resolves the set of fields retrieved and lists them to the log. That listing can be easily copied, pasted into code, and edited to remove unnecessary fields. Use of the DROP= option is discouraged here since that would not be applied until the data is transferred to the SAS environment. The resulting code including the feedback option, appears like:

```
proc sql feedback;
   create table sd1.target as
```

```
      select key, date, field1, field2, …
      from db1.large_table
      where date >= '01JAN2016'd;
   quit;
```

MIXING LIBREFS WITHIN A QUERY STATEMENT

When the PROC SQL query involves multiple librefs, SAS does not use implicit pass-through but rather brings all data to the SAS application server for processing. Consider the query:

```
proc sql;
   create table sd1.target as
   select a.key, a.key2, a.date, field1, field2, …
   from db1.large_table a,
      db1.small_table b,
      db2.small_table c
   where a.key = b.key
      and a.key2 = c.key2;
quit;
```

This creates two potential performance considerations. First, without the benefit of implicit pass-through, SAS will transfer all records from each table across the network connection. Second, the SAS server is now responsible for processing the join. In some environments, the SAS system may have more significant restrictions for memory and CPU which may impact join performance. It is also likely that data sets will not be indexed or ordered properly, resulting in more resource intensive joins.

Two potential improvements could be considered. If the librefs are all from remote databases and those databases are connected, explicit pass-through would allow the query to execute on one machine and return the result set as desired. An example may look like:

```
proc sql;
   connect to odbc as db1 (…);
   create table sd1.target as
   select *
   from connection to db1 (
      select a.key, a.key2, a.date, field1, field2, …
      from large_table a,
         small_table2 b,
         <db2>.small_table c
      where a.key = b.key
          and a.key2 = c.key2) ;
   disconnect from db1;
quit;
```

In this example, the (…) in the connect string would be replaced by connection credentials to db1 data source while the <db2> within the query would be replaced with appropriate values to connect to the second database table from db1 and may include server name, database name, and schema name. Note that everything included in the parenthesize that follow 'connection to db1' is passed as code to db1 to execute there and MUST conform to the syntax on that database. In this case, select * can be used since the field list is explicitly defined in the RDBMS system query.

I often see the scenario when one of the tables is a SAS work table. In that case, the explicit pass-through is not an appropriate solution since the work table is not accessible to the remote database. An alternative is to construct a query to pull the desired data from DB1 that would include any appropriate where clause and filtering we can provide. It is worth considering an ORDER BY statement or an INDEX to aid the subsequent join. A number of factors will influence if either of these options improve performance so it becomes a matter for performance testing by the developer.

In such cases, the interim data tables are seldom desired following the join. Performance can often be improved by creating data views from the initial queries to replace the i/o processing with in memory transfers between steps. An improved solution to the original code could be obtained using the logic:

```
proc sql;
   create view t1_vw as
   select a.key, a.key2, a.date, field1, field2, …
   from db1.large_table a,
      db1.small_table2 b
   where a.key = b.key
   order by a.key2;

   create view t2_vw as
   select c.key2, …
   from db2.small_table2 c
   order by c.key2;

   create table sd1.target as
   select a.key, a.key2, a.date, field1, field2, …
   from t1_vw a,
      t2_vw c
   where a.key2 = c.key2;
quit;
```

This query will allow implicit pass-through on the first query and will sort the resulting data set on the remote system to aid in the subsequent join.

QUERYING WITHOUT A PASSABLE WHERE CLAUSE

Most developers understand the importance of a WHERE clause and would incorporate one as part of the query. However, not all WHERE clauses are passed to the remote database as part of an implicit pass-through. The two most common scenarios that result non passable WHERE clauses are:

- WHERE clauses that are dependent on data not included in the query (see the mixed libref case above).
- WHERE clauses that include sas-specific functions.

When SAS encounters these circumstances, my experience is it will subset the WHERE clause and pass the portion that it can send. Consider the query:

```
proc sql;
   create table sd1.target as
   select a.key, a.key2, a.date, field1, field2, …
   from db1.large_table a,
      sd1.small_table b
   where a.key = b.key
      and a.date >= b.date;
quit;
```

In this case, SAS cannot pass the first part of the where clause (db1 does not have access to b.key) nor the second part (db1 does not have access to b.date) therefore it must retrieve the entire table to SAS. However, if the developer knows that table b only includes dates since July 2015, a simple addition can improve the query performance. Consider this alternative query:

```
proc sql;
   create table sd1.target as
   select a.key, a.key2, a.date, field1, field2, …
   from db1.large_table a,
```

```
        sd1.small_table b
    where a.key = b.key
       and a.date > b.date
       and a.date >= '01jul2015'd;
quit;
```

SAS can now send the third portion since it does not depend on any data from table b. It is not a perfect solution but can provide notable performance gains if the table has substantial history.

Another alternative is to use the key. When the common key in a and b is sequentially assigned, one can gain substantial benefit from querying the minimum key value from b and using it in the WHERE clause. PROC SQL can quickly capture the minimum value in a macro variable and pass it in the subsequent query:

```
proc sql noprint;
   select min(key)
      into :Minkey
   from sd1.small_table;

   create table sd1.target as
   select a.key, a.key2, a.date, field1, field2, …
   from db1.large_table a,
        sd1.small_table b
   where a.key = b.key
      and a.date > b.date
      and a.key >= &MinKey.;
quit;
```

In this step, the first query assigns the minimum value of key from table b to the macro variable MinKey. The second query passes that key as part of the WHERE clause when reading table a from the remote database. Note, this process is not applicable in all cases, but works well when only recent keys are included in table b. One a related note, the first query does not include a create table / view so the resulting query result would be included in the list file. This can be suppressed with the NOPRINT option shown in the code above.

PARSING LONG STRINGS IN SAS

Some character fields can be very long and needed within the project, such as note fields, web strings (URL, campaign tracking strings, machine descriptors), and even XML strings. However, these fields are often parsed for specific data elements which can result in many records and much of the string length being unnecessary. The SAS-based parsing functions will not be passed as part of the implicit pass-through, so the entire string is copied to SAS over network before parsing and filtering can occur. Consider this query:

```
proc sql;
   create table sd1.target as
   select a.key,
      a.date,
      substr(a.field1,24,6) as parse1,
      substr(a.field1,50,5) as parse2,
      substr(a.field1,500,4) as parse3
   from db1.large_table a
   where a.date >= '01jan2016'd
        and substr(a.field1,1,8) in ('REQUEST', 'REPLY');
quit;
```

The query will apply the date filter as part of implicit pass-through. It will require the lengthy field1 be returned to SAS and will apply the resulting where clause and substring logic in SAS. As written, field1 will not be included in the output table sd1.target.

Here is a great role for explicit pass-through. Users can leverage explicit pass-through logic to parse the strings as needed on the remote server, and filtering the records based on parsed values. The explicit pass-through version would look like:

```
proc sql;
   connect to odbc as db1 (…);
   create table sd1.target as
   select *
   from connection to db1 (
      select a.key,
         a.date,
         substring(a.field1,24,6) as parse1,
         substring(a.field1,50,5) as parse2,
         substring(a.field1,500,4) as parse3
         from large_table a
      where a.date > '01/01/2016'
          and (substring a.field1,1,8) = 'REQUEST'
             or substring a.field1,1,8) = 'REPLY'));
   disconnect from db1;
quit;
```

This query returns only the desired fields and records and does not require field1 to be passed via network to the SAS server. Only the fields on the select statement of the explicit pass-through are returned (KEY, DATE, PARSE1, PARSE2, and PARSE3).

WRITING TO NETWORK DRIVES

Query run time can suffer greatly when the target table is on a network drive. Seldom would this approach be optimal since:

- the data is likely not in final form when generated by the query,
- the data will likely be used by SAS further in the process requiring a network read,
- The slower query may extent table locks and resource usage on the remote database.

Actual performance can vary greatly across installations but I have observed query run time increase by more than 500% when the output table writes to a network drive, due to slower output speed.

The solution is simple, change the target to the work folder or another local drive library, then copy the data set to the required destination. Consider this basic query as an alternative:

```
proc sql;
   create table sd1.target as
   select key, date, field1, field2, …
   from db1.large_table
   where date >= '01JAN2016'd;
quit;

data fs1.target,
   set sd1.target;
run;
```

This may provide a better solution when the data is in final form and needs to be written to the file server location. If the table is referenced later in code as a data source, developers should retrieve the sd1 version rather than the fs1 version.

The code change may not reduce overall job run time, but it may release the remote resources required to perform the query more quickly and may reduce contention for those resources. Note, unlike similar examples where views are used in place of tables when passing to another step, a view is not used here since it will not release the resources on the RDBMS while writing results to the file server.

QUERYING THE SAME DATA MULTIPLE TIMES

This is not so much a query efficiency item as general programming style. It is not uncommon to see the same basic query appear multiple times within code with minor variations. In most cases, the overall process will execute faster if we gather all the data from a particular table in one query. Consider the following two queries:

```
proc sql;
   create table sd1.target1 as
   select key, date, field1, field2, field3
   from db1.large_table a
   where a.date >= '01JAN2016'd
      and field1 in ('A','C');
quit;

proc sql;
   create table sd1.target2 as
   select key, date, field1, field3, field5
   from db1.large_table a
   where a.date >= '01JUL2016'd
      and field1 in ('B','C','D');
quit;
```

These two queries can easily be combined into a single query that includes the data requirements of both. A subsequent DATA step can be used to split the result into the two desired tables with the required records and fields assigned to each. When using this approach, it is typically more efficient to use a view for the initial SQL query since the desired result is the two separately generated data sets. The code below generates the same result with a single query and subsequent data step:

```
proc sql;
   create view target_vw as
   select key,date,field1,field2,field3,field5
   from db1.large_table a
   where a.date >= '01JAN2016'd
      and field1 in ('A','B','C','D');
quit;

data sd1.target1 (keep=key date field1 field2 field3)
     sd1.target2 (keep=key date field1 field3 field5);
   set target_vw;
   if date >= '01jan2016'd and field1 in ('A','C')
      then output sd1.target1;
   if date >= '01jul2016'd and field1 in ('B','C','D')
      then output sd1.target2;
run;
```

Even in cases where queries are substantially different, code may benefit from completing all data extracts from a given large table before proceeding to another. Most RMDBS as substantial memory and buffering space. When a table query completes, some or all of that table may be accessible from buffers. Executing a second query on that table immediately following the first may allow the RDBMS to utilize data in memory rather than forcing a second pull of data from spinning discs.

INCLUDING DATA STEP OPTIONS WITH SOURCE TABLES

Most SAS DATA step users are familiar with data step options such as DROP=, KEEP=, WHERE=, RENAME=, OBS= that can follow the dataset name enclosed in parenthesize. They are valid in most cases where a dataset is referenced, including in PROC SQL. However, when these are associated with SQL source tables that are being retrieved from a remote database, the implicit pass-through logic is bypassed. Consider the query:

```
proc sql;
   create table sd1.target as
   select *
   from db1.a_large_table (drop=notes) a
   where a.date >= '01JAN2016';
quit;
```

Here, the user is trying to drop the NOTES field rather than entering the desired fields in place of *. Without implicit pass-through, the entire table is transferred to SAS, before the NOTES field is dropped and before the where clause executes. This can in fact make the query slower rather than faster.

USING NON-INDEXED FIELDS FOR WHERE CLAUSE AND JOINS

SQL databases are optimized to leverage indexes within queries. However, due to maintenance requirements, it is not practical to index every field a user may wish to include in joins or where clauses. Therefore, the developer should consider the indexes that are available when writing the query. As an example, suppose a table has two date fields: DATE which records the day when the transaction occurred and DT which includes the DATE and TIME when the transaction occurred. Only the DATE field is indexed. The analyst is interested in transactions that occur between 8:00 AM and 11:00AM on August 1, 2015. The query could look something like:

```
proc sql;
   create table sd1.target as
   select b.dt, …
   from db1.LARGE_TABLE b
   where b.dt between '01AUG2016:08:00:00'dt and '01AUG2016:11:00:00'dt;
quit;
```

To execute this query, the database must read every transaction to check the where clause criteria since there is no index. A simple addition of the date field to the where clause allows the database to leverage the index on date:

```
proc sql;
   create table sd1.target as
   select b.dt, …
   from db1.LARGE_TABLE b
   where b.date = '01AUG2016'd
     and b.dt between '01AUG2016:08:00:00'dt and '01AUG2016:11:00:00'dt;
quit;
```

A similar scenario can often arise within data warehouse systems. The warehouse as business keys, derived from the original source data, and warehouse keys, generated and maintained during the warehouse construction. Analyst need to know whether business keys are indexed. In many cases, warehouse systems will be optimized to join on the warehouse keys and may or may not maintain indexing on business keys.

RESOLVING LOOKUP TABLES ON THE REMOTE DATABASE

Implicit pass-through is not always a good thing. Consider the case of a lookup table. A lookup table typically has few rows but may have one or more lengthy character fields. Perhaps the table contains a

50-byte character description associated with 10 distinct code values. One can download the lookup table and transfer the description a total of 10 times. If the lookup table is resolved on the remote side, the description is transferred one or more times for each row returned. This could result in downloading millions of cases. Consider the query:

```
proc sql;
   create table sd1.target as
   select a.key, a.key2, a.date, b.description
   from db1.large_table a,
      db1.lookup_table b
   where a.key2 = b.key2;
quit;
```

In this simple query, the easiest adjustment may be to turn off the SQL pass-through option and transfer processing to the SAS application server. That can be done by adding the option NOIPASSTHRU to the PROC SQL statement.

Although NOIPASSTHRU may be appropriate in the simple query above, we lose access to all the potential benefits of implicit pass-through. More advanced queries may benefit from an approach more in line with the approach used for mixed librefs. An improved solution to the original code could be obtained using the logic:

```
proc sql;
   create view t1_vw as
   select a.key, a.key2, a.date
   from db1.large_table a
   order by a.key;

   create view t2_vw as
   select b.key2, b.description
   from db1.lookup_table b
   order by b.key2;

   create table sd1.target as
   select a.key, a.key2, a.date, b.description
   from t1_vw a,
      t2_vw b
   where a.key2 = b.key2
   order by a.key;
quit;
```

This reduces the data transferred in description but has now placed sorting into the process. Left without sorting, the resulting table will typically be sorted in key2 order which will likely require some efforts to manage in subsequent processing. This sort can potentially override the value of the data reduction, generating no added benefit.

A potentially better solution is to leverage custom formats where the user converts the lookup table to one or more custom formats and applies the format to the table rather than imposing a join. This is very flexible and work well for simple lookup tables. Also, hash tables and set statements with key= can be used effectively within a DATA step and have the ability to handle larger lookup tables. The specific code examples to complete these alternatives are outside the scope of this document, but for those familiar with the process, they are fairly simple extensions.

KNOW WHEN TO SAY WHEN

These techniques can dramatically reduce job execution time. I have observed job execution times drop from 10-15 hours to under an hour. That offers clear value in most business scenarios. It not only runs quicker, but there is often a corresponding drop in use of limiting resources such as drive channel traffic,

network traffic, CPU and memory utilization on servers, table lock times on databases, and job scheduler slots. The benefits can be especially valuable when the resources released are reaching maximum capacity. For instance, if your SAS application server is running near max CPU consistently, code changes that reduces CPU load like implicit or explicit pass-through can be a huge benefit. If the network or drive channels are nearing max capacity, processes that reduce data transfers on these channels can be equally huge.

However, not all efforts translate to clear of benefit. While reducing a job from 5 hours to 1 hour may be a win, getting a job that runs in 5 seconds to run in 1 may not be greeted with the same enthusiasm. The value may be situational. While reducing run time from 5 minutes to 1 minute for a daily batch jobs may not offer much value, the same gain on a query executed multiple times per day on demand may be a major win, especially if managers trigger the query and are waiting for results to be returned.

For maximum benefit, focus effort on jobs that:

- Take longer to run,
- Run more frequently,
- Are more often used in interactive sessions,
- Tax resources that are heavily strained,
- Run during peak load periods.

And then, know when to say when. There is always more that can be done to improve code efficiency, but not all such efforts justify the effort to find, alter and test the recommended changes.

CONCLUSION

The coding examples discussed here highlight common constructs that can significantly hamper query and even system performance. The goal is to provide coders with specific examples to help them identify when their queries may be needlessly inefficient without getting into complex, system specific topics or the level of performance tuning that can be accomplished through more detailed platform administration. This should give users some tools to use, but these must be tested in your environment and with your data. Every configuration and every case is unique. Take these examples, try them in your environment, and implement what is applicable to your situation.

REFERENCES

Johnson, Misty. 2015. “Just passing through… Or are you? Determine when SQL Pass-Through occurs to optimize your queries.” *MWSUG 2015 Conference Proceedings*, Omaha NE:

Available at http://www.mwsug.org/proceedings/2015/BB/MWSUG-2015-BB-03.pdf

CONTACT INFORMATION

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

John Schmitz
Sr. Manager, Luminare Data LLC
john.schmitz@luminaredata.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.