SAS® GLOBAL FORUM 2020 #SASGF

Paper 4929-2020

Best Practices in SAS® Programming to Support Ever-Increasing Data Loads.

John Schmitz, Luminare Data LLC

ABSTRACT

In most large organizations, SAS® serves a pivotal role in data processing for warehousing, reporting, analytics and more. The SAS language provides multiple tools and options to streamline these data processing needs that may be unfamiliar to many developers. This paper will present some well-known and lesser-known SAS methods for efficient data handling. The focal point of the paper is more geared towards what is available and the use cases for each technique, rather than a detailed how-to on a specific solution.

INTRODUCTION

SAS® often serves a critical data processing role for many large businesses across industries. It offers a rich set of programming features and options combined with extensive data access features provided through an array of SAS/ACCESS® engines. These capabilities allow SAS to extract and process data from multiple sources and in some cases directly process data within the system where those data are stored.

Learning the capabilities of these various programming features, procedures, and SAS/ACCESS engine tools can be a challenge, especially when there are multiple methods to accomplish the same end. Knowing when to apply each method to efficiently accomplish the desired end is an art that can produce huge savings to the organization in terms of resource utilization and timely solutions.

The focus of this paper is to highlight some popular and powerful tools that the SAS developer can leverage for data extraction, transformation and load (ETL) processing. The discussion leans more towards the application of common tools that SAS developer should understand rather technical details regarding how a specific solution would be implemented.

The comments here apply primarily to SAS V9 programming. Many of the topics can be applied to SAS Viya® as well, but the CAS engine and memory-resident data design of SAS Viya create a unique dynamic that is not considered directly within this topic.

CORE BACKGROUND CONSIDERATIONS FOR ANY ETL PROCESS

Whenever ETL efforts are required for a SAS project, an initial assessment of key factors can direct the programmer to more efficient extraction methodologies. Choice of programming methods and join alternatives can have a dramatic impact on resource utilization and hence query runtime. Some key considerations for this initial assessment include:

- The data volume required for the extract.
- The number of distinct data source tables and source locations required for the effort.
- A general idea of the ratio of required data to the entire table size for larger tables.
- Presence of relevant sort order or indexing on larger source tables.

- Resource and performance capabilities of the core components that support various extraction methods.

Resource considerations focus primarily on:

- Drive input/output(I/O) or read and write performance.
- Available network bandwidth.
- System CPUs capabilities and load.
- Availability of system Random-Access Memory (RAM).

When data reside in SAS datasets, Excel spreadsheets or other various flat file formats, these considerations focus primarily on the SAS application server hardware. However, when data reside in a RDBMS or another advanced data appliance, the capabilities of these platforms come into consideration as well. SAS may leverage implicit and explicit pass-through logic to process data within the host system, leveraging its processing capabilities.

A DRIVE IS NOT A DRIVE

Many factors impact drive performance, including RAID settings, drive pools in SAN systems, cache settings, and resource competition from other jobs. The environmental factors are controlled by admins often well removed from SAS, but these settings can have significant impact on SAS job performance.

Some scenarios may benefit from moving data to alternative file locations for improved performance or to obtain adequate space. Usually, this would be accomplished by updating a LIBNAME statement. However, there are two options that can be useful for avoiding drive related issues. Developers can leverage the USER= option to redirect files an alternative library. For example, the statement:

```
option user=alt;
```

would redirect all single name datasets that are typically written to the WORK library to the ALT library. This would require a LIBNAME statement to define a libref ALT.

SAS also supports a MEMLIB option which moves the WORK folder to reside in-memory rather than on disc. MEMLIB may require SAS Admins configurations to be usable in your environment. MEMLIB is seldom used since WORK often contains multiple large datasets that can quickly consume all the available system memory. However, the combination of MEMLIB with USER= can create a workable environment where WORK files are redirected so only high-priority files are loaded to WORK in MEMLIB.

KNOW YOUR DATA PATH

When accessing datasets and flat files, FILENAME and LIBNAME statements define the location where the files are stored. Sometimes these file system files are directly attached to the SAS application server while in other cases, these can only be retrieved via a network shared file link. This distinction is important since the two types may use different channels to bring data to the application server for processing. Network share files will normally be retrieved via the ethernet channel while files arriving from attached file systems (local drives, attached SAS storage systems, etc.) can bypass ethernet and travel over much faster fiber or serial channels. These two channels may exhibit native speed differences of 50x or more, and users may experience 100x or more difference when network contention and load further degrade ethernet bandwidth. I have seen scenarios when pathnames to attached file storage were written using a shared file pathname structure and resulted in file i/o being needlessly routed through the slower ethernet path.

MEMORY LIMITATIONS

System memory can be used to improve performance of many processes. Some procedures, such as PROC SORT, PROC SUMMARY, and PROC SQL cache information in memory to improve run performance. Other tools, such as HASH tables and FORMATS, load data into memory to quickly retrieve information when required by the process. However, when system memory requirements exceed available resources, data will be swapped to disc either directly by the procedure or via pagefile swapping. Options such as MEMSIZE, SORTSIZE, SUMSIZE can be used to allocate restrict memory to specific processes but may require settings within the job to change these limits to optimize memory allocation.

Another valuable tool available in the SORT procedure is TAGSORT. This option allows the procedure to drop fields from memory that are not necessary to the SORT process. This option reduces memory requirements for the sort but does require a second pass through data to remerge the dropped fields. This option can greatly improve dataset sort performance in applicable cases. All dropping and remerging is completed by the procedure whenever the TAGSORT option is requested so no additional programming is required.

SAS DATA STEP VS THE SQL PROCEDURE

The DATA step has been a mainstay of the SAS language from its beginning. It is designed to process a large volume of data with minimal resource overhead and performs efficient join logic on sequentially ordered data. The SQL procedure leverages structured query language (SQL) syntax, thus providing language syntax that may be familiar to a larger development audience. PROC SQL creates a flexible coding process that encapsulates much of the DATA step capabilities as well as the often-used SORT and SUMMARY procedures.

Although PROC SQL is frequently used in SAS coding due to its familiarity and flexibility, its inherent internal design can often result in increased resource requirements and diminished ETL performance. That said, PROC SQL certainly has a role in well-designed ETL processes.

The two alternatives differ at their lowest levels, provide different methodologies to reach similar ends. The DATA step uses file pointers to read files sequentially while PROC SQL leverages a cartesian join that is order independent. Properly understood, these competing methodologies can be leveraged to optimize ETL processing.

Consider two data tables to be joined, Table A which contains information about car makes and Table B containing information about specific models. Figure 1 shows the row matches that must be evaluated while processing the join with a DATA step while Figure 2 shows the matches that must be evaluated using PROC SQL. Even in this simple example, PROC SQL requires far more evaluations than DATA step to complete the join. Furthermore, the number of comparisons is increasing linearly for the DATA step and multiplicatively for PROC SQL as table sizes increase.

Syntax for these basic joins should be familiar to most SAS developers but are included here for reference and comparisons to other examples provided later in this paper. The Example 1 code joins the described tables using DATA step logic and a MERGE statement. Below it, Example 2 produces a similar result using a PROC SQL solution. The Example 1 solution appears as:

```
** EXAMPLE 1: SIMPLE MERGE USING DATA STEP **;
proc sort data=inputa;
   by key;
run;

proc sort data=inputb;
   by key model;
run;
```

```
data out1;
    merge inputa (in=a)
           inputb (in=b);
    by key;
    if a and b;
run;
```

Since this solution uses a BY statement, calls to the SORT procedure are included to ensure the data are properly ordered. These are followed by a simple DATA step join. The IN= option is used to identify which tables contribute to the record. Below the IF A AND B statement restricts the output to include records that appear in both tables, hence dropping the record for MAZDA that would otherwise be included in the output table.

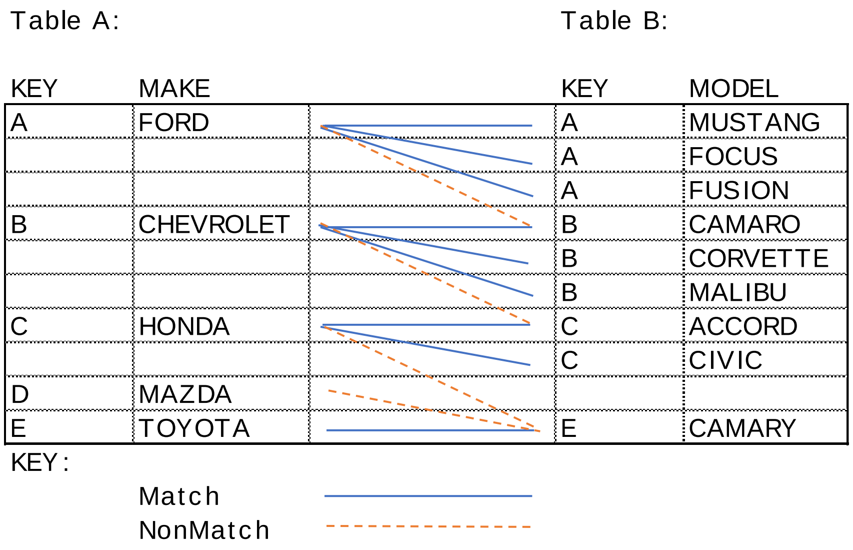

Figure 2. Row Evaluations Required for DATA Step Join.

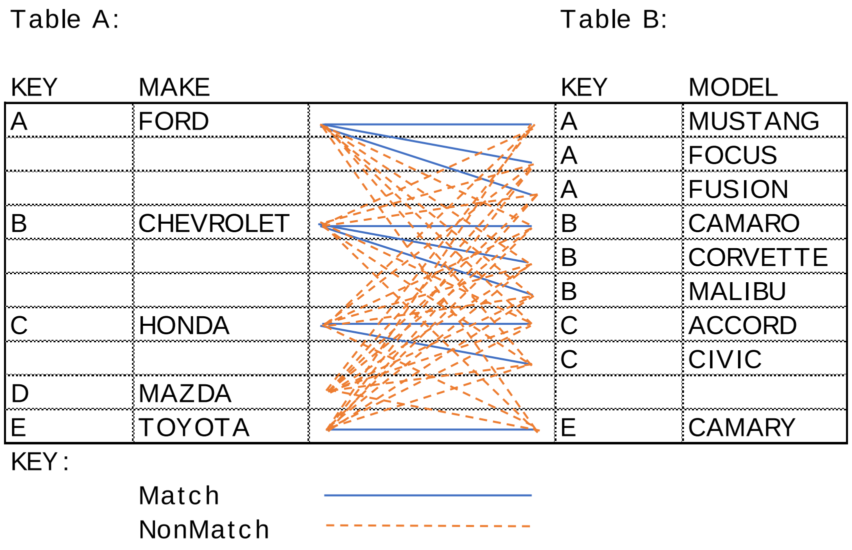

Figure 1. Row Evaluations Required for PROC SQL Joins.

Example 2 obtains similar results using SQL code:

```
** EXAMPLE 2: SIMPLE JOIN USING PROC SQL **;
proc sql;
   create table out2 as
   select
        a.key,
        a.make,
        b.model
   from
        inputa as a,
        inputb as b
   where a.key = b.key
   order by a.key, b.model;
quit;
```

In Example 2, an inner join of the two input tables is established. No sort of the input table is required since the step is order independent, but an ORDER BY is added so that the resulting data matches results from the DATA step.

When considering query performance, it is essential to consider the limitations imposed by the sequential order requirement of the DATA step. Sorting data within large table can also be very inefficient and sort time must be considered when evaluating these two options. When the sequential ordering required by the DATA step is consistent with a logical ordering of data for the broader efforts, sorting and DATA step often makes logical sense. However, if use of DATA step would require large table sorting before and perhaps after the merge, PROC SQL offers a compelling alternative. Later in the paper, other options will be discussed that may offer better options than either of these two more common approaches.

There are cases where PROC SQL will often provide a more efficient ETL process. One important case involves joins that leverage inequality constraints. These often include cases such as join on DATES between event records in another table and fuzzy-join logic. PROC SQL can also be very effective when data source is a RMDBS since data are seldom stored in sequential order on external databases. The DATA step works well for tasks that require sequence, for instance first and last record identification, incremental counters, processing lags, differences and other criteria that is based relative to a previous record.

IMPLICIT AND EXPLICIT PASS-THROUGH CONSIDERATIONS

Implicit and explicit pass-through logic becomes an essential consideration when using SAS/ACCESS engines to connect to remote data systems such as SQL Server, Oracle, Teradata, Netezza, or Hadoop. This is true whether development is done using PROC SQL or DATA step and its related procedures.

WHAT ARE IMPLICIT AND EXPLICIT PASS-THROUGHS

Pass-through are used by the SAS/ACCESS engines to execute query activity on a source database system rather than executing on the SAS application server. Pass-through logic can greatly reduce the resource load of an ETL process by executing core elements at the source. Suppose there is a large table on an external database system and the current ETL process requires one percent of the records from the table to complete its steps. Without pass-through, the entire table would need to be extracted to SAS for processing, usually over the slower ethernet channel so that the where clause could be evaluated that would dispose of 99% of the transferred data. With pass-through logic, the where clause can be passed to the source system and applied, greatly reducing the network load generated by the process.

Implicit pass-throughs are automatically generated by SAS on the user's behalf without any specific programming required. Often query requirements such as field selection, where clauses, table joins, basic transformations and table sorts are passed automatically to the host where memory, storage, table indexes, and processing are optimized to support these queries. Rather than transferring all data to the SAS application server, the query can the passed to the source and only the result set from the query is transferred to SAS.

Implicit pass-through are not limited to PROC SQL. DATA Steps, PROC SORT, and other procedures that involve a dataset that is being accessed via a SAS/ACCESS engine may potentially leverage pass through operations while executing.

Explicit pass-through involves specific programming efforts where the developer creates a connection to the source data system and transfer explicit programming instruction written in that system's native syntax. This process requires specific knowledge of the programming language used on the host but can be utilized to leverage host-specific query capabilities that cannot be assessed by the SAS language or implicit pass-through logic.

REQUIRMENTS FOR A SUCCESSFUL IMPLICIT PASS-THROUGH QUERY

Implicit pass-through logic does not apply in all cases since SAS cannot always pass the query requirements to the source. Three common issues that negate implicit pass-through logic include:

- Mixed libref within the query.
- Where clauses that require data not available on the remote system.
- Where clauses that include SAS-specific functions that do not readily map to SQL.

One of the more common scenarios for a query to execute local to SAS involves the mixing of librefs within the query statement. When the query involves joins across two or more tables, SAS can only pass the query if all tables exist on the same source system. Example 3 provides a case where the join involves data from WORK as well as a REMOTE system:

```
** EXAMPLE 3: JOIN THAT FAILS IMPLICIT PASS-THROUGH **;
proc sql;
    create table out3 as
    select
          x.key,
          x.field1,
          x.field2,
          y.fielda,
          y.fieldb
    from
          work.inputx as x,
          remote.inputy as y
    where
          x.key = y.key
          and y.date GE '01JAN2020'd;
quit;
```

In this query, the RDBMS does not have the table WORK.INPUTX, hence the table REMOTE.INPUTY must be copied to SAS to execute the query. The first line of the WHERE clause requires information from X so it cannot be transferred to the remote system either. The second item of the where clause with the DATE restriction can be transferred so the extract of INPUTY can apply this filter to reduce the transfer file size from REMOTE. The same query using the SAS INTNX function to compute the desired date would likely NOT be passed as part of the INPUTY query resulting in every row of that table being transferred to SAS for join processing.

WHEN TO AVOID IMPLICIT PASS-THROUGH OPERATIONS

Some specific scenarios may perform better without pass-through logic. It is common for warehouse schema to leverage many small DIMENSION or LOOKUP tables that support a large FACT or TRANSACTION table. Many keys inside the FACT table reference to the DIMESION tables that often contain larger STRING field descriptions and other details. Resolving all these joins on the source system can result in a much large result set to return to the SAS application server. In these cases, it may be quicker to transfer data to SAS, then resolve the various joins to DIMENSION tables. Within PROC SQL, this can be done quickly by adding the NOIPASSTHRU command to the PROC SQL statement. When used, this is option is then applied universally within the PROC SQL step.

ALTERNATIVE JOIN CONSTRUCTS AND TRANSFORMATIONS

Most ETL processes require data from multiple sources, necessitating join constructs to be an essential component of the task. In many ETL process, table joins represent a significant portion of the overall process. Proper structuring of these joins is critical to an efficient ETL processing.

The basic SQL and DATA step joins were discussed and compared above. This section highlights other options that can supercharge ETL processes when properly implemented. Each method has specific strengths and limitations that must be considered.

USER-DEFINED CUSTOM FORMATS

Many database schemas leverage small lookup tables that may contain a single name or description field, identified through a coded key. User-defined formats provide an efficient method to populate such fields. A simple DATA step and call to the FORMAT procedure using the CNTLIN= option can create a format from the lookup table. The resulting format can then be leveraged using a PUT or INPUT statement to quickly incorporate the data in a subsequent step. The format can be applied in either the DATA step or in a PROC SQL step.

In application, the format is loaded to memory when used by the DATA step or PROC SQL. This can limit the maximum size of table that can be joined by this method, but that is usually quite large. The more restrictive elements are that the USER-DEFINED format can only leverage a single field key and only return a single value per format.

In the first two code examples presented, a one-to-many join was executed using DATA step and SQL logic. This simple example can be recreated with a user-defined format. The format is built to represent the unique values from the join from INPUTA. We can then apply that format to the many side (INPUTB) to assign the associated unique value. Sample code to create the user-defined format is shown in Example 4a:

```
** EXAMPLE 4a: BUILDING A USER_DEFINED FORMAT FROM AN INPUT TABLE **;
data Makefmt_vw /view=Makefmt_vw;
    format
          start $10.
          label $30.
          fmtname     $20.
          hlo         $3. ;
    keep start -- hlo;
    retain FmtName '$Makefmt'  HLO ' ';
    set inputa end=last;
    start=key;
    label=make;
    output;
    if last then do;
          start=.;
          label = 'ERROR';
```

```
            hlo='o';
            output;
      end;
   run;

   proc format cntlin=Makefmt_vw;
   run;
```

Here, the DATA step creates a view MAKEFMT_VW with 4 fields:

- START: the lookup key value.
- LABEL: the description field from the lookup table.
- FMTNAME: the name PROC FORMAT will assign to the user-defined format.
- HLO (high-low-other): Define the OTHER category, in this case setting to 'ERROR'.

This code also includes one additional row to capture errors. With the ERROR line, any keys not included in the FORMAT will receive the value 'ERROR' in the resulting field. Once the dataset is created, the FORMAT procedure reads and generates the user-defined format based on the dataset information.

The combinations of INFORMATS and FORMATS, CHARACTER and NUMERIC can create confusion. As a general rule:

- Use PUT and FORMAT when the resulting field is a CHARACTER.
- Use INPUT and INFORMAT when the resulting field is NUMERIC.
- Use a CHARACTER based FORMAT or INFORMAT if the input field is CHARACTER.
- Use a NUMERIC based FORMAT or INFORMAT if the input field is NUMERIC.

Example 4a uses a character-based lookup key, but the conversion to a numeric-based key or creation of an INFORMAT is very similar. When building a user-defined format definition using a dataset and the CNTLIN option, adding a @ as the first character in FMTNAME results in the generation of a INFORMAT while adding a $ as the first, or second in case of an INFORMAT, results in a CHARACTER-based FORMAT or INFORMAT.

Once the format or informat is generated, PUT or INPUT statements can be used in either a DATA step or PROC SQL. Example 4B shows both cases for the format defined above:

```
   ** EXAMPLE 4b: USING YTHE USER_DEFINED FORMAT IN CODE **;
   data out4a;
      set inputb (in=a);
      length make $15.;
      make = put (key,$Makefmt.);
      if model NE 'ERROR';
   run;

   proc sql;
      create table out4b as
      select
            b.key,
            put(b.key,$Makefmt.) length 15 as make,
            b.model
      from
            inputb b;
   quit;
```

In both cases, INPUTB is processed by the code and the newly created FORMAT is used to ASSIGN a value for MAKE.

When using this construct with PROC SQL, note that this format is available to SAS-side only so using this on a step that reads from a RDBMS system could adversely impact any implicit pass-through.

JOINING TABLES USING HASH OBJECTS

Hash objects give the DATA step a powerful alternative to combine data from multiple tables but require enough available memory to hold at least one of those tables in memory. Because the memory resident has table can be quickly searched, hash objects allow the DATA step to process without a BY statement, hence becoming order independent. The hash object can support multiple keys, multiple data fields, and non-unique lookup records, giving the hash object considerable deployment flexibility.

Example 5 reproduces the join of INPUTA and INPUTB:

```
** EXAMPLE 5: DATA STEP WITH HASH OBJECT **;
data out5;
    ** DEFINE AND POPULATE HASH DATA **;
    if _N_ = 1 then do;
          format key $1. make $15.;
          declare hash a (dataset:'inputa' ,ordered: 'N');
          rc = a.definekey('key');
          rc = a.definedata('key', 'make');
          rc = a.definedone();
          call missing (key, make);
    end;
    ** READ TABLE B **;
    set inputb ;
    ** SEARCH HASH FOR MATCH AND OUTPUT RECORD IF FOUND **;
    rc = a.find();
    if rc = 0 then output;
    drop rc;
run;
```

In this scenario, the hash object replaces the user-defined format of Example 4. The first section of code defines a hash object named A, derived from the input table INPUTA. The next steps define the structure of that table for keys and data contained within it. Note that the hash is not defined until the first record of the execution step is processed. Also, a format statement is needed to define the hash object fields since they are otherwise not available at compile time.

Later in the code, the hash find method is called by the statement:

```
rc = a.find();
```

At this point, the method searches the hash for a matching record to KEY from INPUTB. If a match is found, the value for MAKE is returned from the hash and updates the value of the field within the DATA step. This logic also restricts output to records found in the hash object, keeping with the inner join logic used throughout.

Although results and performance of these two approaches are similar in this simple case, hash objects offer far greater flexibility to efficiently handle cases that are beyond the scope of what a user-defined format can accomplish. If required and adequate memory is available, multiple tables with multiple key orders can be combined in a single step.

SET WITH KEY= OR POINT= OPTIONS

Within a SAS DATA step, SET statements can be used that leverage KEY= and POINT= options. The KEY= option retrieves a specific record from the table, based on the value of the field KEY already assigned within the step. Typically, KEY= and POINT= are used in

DATA steps in conjunction with another SET or MERGE statement. Example 6 uses a SET with KEY= to reproduce the same join process:

```
** EXAMPLE 6: DATA SET WITH SET= **;
proc datasets lib=work nolist;
    modify inputa;
    index create key / unique;
run;
quit;

data out6;
    ** READ TABLE B **;
    set inputb ;
    ** LOOKUP VALUE FROM TABLE A **;
    set inputa key=key/unique;
run;
```

This example is very similar to the previous two cases where data from INPUTA is joined to INPUTB. Here the SET KEY= performs a similar role as the HASH object above. The process is order independent but does require an index on the field or fields used in KEY=.

The primary difference between the HASH OBJECT solution and the KEY= solutions is memory versus drive requirements. The KEY= process generally provides efficient results that are less memory intensive but more i/o intensive than either the USER-DEFINED FORMAT or HASH object options. The logic leverages indexing to determine which record is required, then reads the record from disc rather than memory. However, performance issues can be encountered, especially when:

- The size of the dataset (INPUTA) is large,
- I/O performance is marginal, and
- The internal ordering of the two datasets are significantly different.

The combination of these three factors can lead to multiple reads of the same data as SAS would be required to read the same dataset page multiple times to complete the query.

The POINT=option is similar to KEY= but provides a numeric ROW number to return rather than an index KEY field. One of the unique applications of POINT= is the ability to return data from the NEXT observation that can be used in the correct logic. The following example uses STARTTIME from the next record to populate missing ENDTIME from the current record (see Example 7):

```
** EXAMPLE 7: DATA SET USING POINT TO READ NEXT RECORD **;
data out7;
    set input7;
    by key starttime;
    nextobs= _N_+1;
    if not last.key then
       set input7 (keep=starttime
         rename=(starttime=nextstarttime)) point=nextobs;
    endtime = coalesce(endtime, nextstarttime, datetime());
    NEXTSTARTTIME=.;
    drop nextstarttime;
run;
```

The input data table INPUT7 is sorted by a KEY and STARTTIME but has missing ENDTIME values for some records. For these missing entries, the STARTTIME of the next record is used as ENDTIME, or the current time stamp if no subsequent record is available. The BY statement is used since it is necessary that the value used comes from the same KEY value and is the next sequential value. Hence the DATA step in this example is ORDER DEPENDENT. The COALESCE function returns the first nonmissing value from a list. In Example 7, the COALESCE function will return ENDTIME if available, NEXTSTARTTIME as the next possible value, or DATEIME() if neither of the other values exist. The not LAST.KEY skips the NEXT record retrieval when the next record up in the set is for another KEY and importantly stops the systems from querying a record beyond the last data set record.

USING DATA STEP TO TRANSPOSE DATA

Many SAS developers are familiar with the TRANSPOSE procedure for data transpose requirements. The procedure works effectively in most applications but does require a separate and specific program step to complete. A properly constructed DATA step can also complete most TRANSPOSE requirements. Further, since the process is incorporated within a DATA step, it may eliminate one or more additional programming steps. Users have more control over variable naming and can conceivably transpose multiple variables within a single step. If needed, additional code can be added to operate on data before or after the transpose occurs potentially combining additional steps.

Example 8 constructs a simple case where a dataset is transposed from a monthly level structure to a yearly structure with fields for each month. The DATA step ARRAY provides a convenient programing feature to readily support the imbedded loop logic required by the transpose logic. For completeness, a second process is shown that converts the dataset back to its original structure. The code shown does require the data to be in sorted order:

```
** EXAMPLE 8: USING DATA SET TO TRANSPOSE DATA **;
data out8a;
    do until (last.year);
          set input8 (keep=key year month value);
          by key year month notsorted;

          array v {12} value1-value12;
          v(month) = value;
    end;
    drop month value;
run;


data out8b;
    set out8a (keep=key year value1-value12);

    array v {12} value1-value12;
    do month = 1 to dim(v);
          value = v(month) ;
          if value NE . then output;
    end;
    drop value1-value12;
run;
```

In this logic, the DATA step for OUT8A places the SET statement inside a loop that processes all records for a given year in each pass through the DATA step. The implied output statements at the end of the step writes a record for each KEY, YEAR value. The ARRAY defines fields VALUE1-VALUE12 that capture VALUE for each month.

In the second step (OUT8B), a loop is created after the SET statement that loops over the dimension of the ARRAY. Here, MONTH is recreated by the loop and VALUE is captured from the 12 value fields created earlier. An OUTPUT statement is used inside the loop, to generate records for each month while an IF statement is used in this case so that missing values are not outputted to the final table.

For the first case, the process is not necessarily sequence dependent. The NOTSORTED option can be used here to avoid sorting by KEY YEAR, but the process will not produce the desired result if all the records for a given KEY YEAR combination are not grouped together. The second case shown would necessarily be grouped by KEY YEAR already, since that is the granularity level of the data. Hence the second case would not require any specify sequential ordering of the input data.

USING DATA STEP FOR A SORTED APPEND

SAS include the APPEND procedure and SQL offers UNION logic to append multiple data tables, adding rows from each table into the resulting table. Either of these processes are adequate for appending rows. However, these do not maintain sort ordering which may be established for the input datasets and may be desired for later use by the process. A subsequent PROC SORT could be executed, but this approach is very inefficient for larger tables if they were already in sorted order. Example 9 shows a DATA step alternative that maintains existing sort order by including multiple datasets on a single SET statement:

```
** EXAMPLE 9 USING DATA STEP TO PRODUCE SORTED APPEND TABLE **;
proc sort data=input9a;
   by key date;
run;
proc sort data=input9b;
   by key date;
run;

data out9;
   set input9a
        input9b;
   by key date;
run;
```

In this example, the two datasets in sorted order are combined into a new output table. Since a BY statement is used with the SET statement, records are read from each table and interweaved to produce a new table that remains in sort order, as specified by the BY statement.

LEVERAGING VIEWS AND INDEXES

A typical SAS process involves multiple programming steps, primarily composed of DATA steps and various procedure (PROC) statements. Most of these steps involve reading data from a permanent data store or interim results from a previous step, processing those data as required by that step, and writing results to a permanent or interim data store. When processing larger data structures, each of these steps can involve notable disk I/O load and can require substantial drive space. Many times, these outputs are interim tables stored within WORK or an alternative location used for interim tables.

USING VIEWS

One convenient method to reduce the storage and I/O load from such a process is to include data views. Views can be created in place of output tables by either PROC SQL or DATA step code. For PROC SQL, simply replace TABLE with VIEW in the CREATE statement:

```
CREATE VIEW myview_vw as
```

while in the DATA step a VIEW= option is added to the DATA statement:

```
DATA myview_vw / view=myview_vw;
```

DATA step and PROC SQL processing involves two distinct steps: COMPILE and EXECUTE. The COMPILE step validates most syntax elements and generated the specific machine operations required to complete the desired result but does not process the data. The EXECUTE step captures the required machine operations and applies those operations against the specified input data. In a view, the compile step is completed, and the required machine operations are stored, but the EXECUTE step is not triggered. The EXECUTE step can then be initiated by a subsequent program step when required, saving the OUTPUT and subsequent INPUT of the interim dataset and any storage requirement associated with the interim dataset. These I/O and storage savings can be significant on large processes and are ideally suited for cases where the resulting data is only required once. They can provide a hinderance in development if not well planned since opening a dataset for review will require the EXECUTE step to process.

One small note, dataset names should not be interchanged between data and view types. SAS will error when trying to write a view when a table by the same name exists and vice-versa. To avoid such confusion, a process to distinguish view names, such as adding '_vw' can be helpful, but it is not required by the SAS language. Also, it is important not to delete or overwrite data tables that views read since their successful execution requires these data to be available when executed.

USING INDEXES

Properly indexed files can be critical to join performance when using PROC SQL. As shown above, indexing is also required when the KEY= option is added to a SET statement. Properly assigned indexes can also be used in place of sorted data when a BY statement is used in a DATA step or procedure.

Indexed datasets are common for tables stored within a RDBMS and are leveraged by implicated pass-through logic in PROC SQL, PROC SORT, DATA step and others. When SAS datasets and especially interim datasets within the job are used, files are seldom indexed. Users would need to create file indexes their interim files and recreate them whenever the dataset is replaced.

In many applications, the overhead of creating the index exceeds its benefit, especially for one-time use. Sorting the data can often be accomplished in similar run times and will typically execute more quickly. However, there are good use cases for indexed datasets. This is most notably true when PROC SQL joins are constructed that involve datasets with different inherit order requirement. An INDEX can also be used to avoid having to sort a dataset for a specific process, then resort the data after that process completes. Example 6 above includes a code example for indexing a dataset using the DATASTEP procedure, but they can able be created using DATA options and PROC SQL code.

USING TYPES AND ID WITH THE SUMMARY PROCEDURE

The SUMMARY[1] procedure is often used to aggregate data, similar to a GROUP BY in PROC SQL. A CLASS statement is used to assign aggregation levels for the output dataset. Unlike a SQL GROUP BY, the SUMMARY procedure returns aggregation results for all possible combinations of the CLASS variables unless the option NWAY is used.

With larger datasets and higher frequencies of CLASS statements, output from a SUMMARY procedure can become quite large. Yet in some scenarios, only a few of the possible

[1] The discuss references SUMMARY but comments apply equally to both SUMMARY and MEANS procedure.

combinations are required as output from the procedure. The TYPES statement provides a convenient method to stipulate the desired CLASS combine for the OUTPUT dataset. Example 10 demonstrates a PROC SUMMARY using the TYPES statement:

```
** EXAMPLE 10 USING TYPES WITH PROC SUMMARY **;
proc summary data=input10;
   class A B C D E;
   id K L;
   var val1 val2;
   TYPES ()
         A*(B C)
         A*B*C
         C*(D E);
   output OUT=OUT10 sum;
run;
```

The logic returns the SUM of VAL1 and VAL2 at each level of aggregation. In this example, a possible 32 (2^5) output combinations could be produced, but the TYPES statement reduces this to only 6 combinations for the final output. The first line () returns the total across all records. The second line returns A*B and A*C combinations while the third returns the combinations of A*B*C The final line of the TYPES statement returns C*D and C*E combinations only. Computationally, it is quicker to produce all the desired combinations in a single process than to run multiple SUMMARY procedures which compute each desired aggregation level.

By default, a PROC MEANS will generate results at the lowest level of granularity. However, TYPES can also be used with PROC MEANS to select additional levels to include in output.

Another option that can be helpful to simplify PROC SUMMARY code is the ID statement. Many times, the CLASS statement is hierarchical in nature. The input dataset may contain fields for the company region identifier as well as a field for the regional manager's name. Both fields may be desired in the final output, but we can use the ID statement to include the manager's name without incurring the computational expense of including it as a CLASS variable. In Example 10, the ID statement passes values for K and L to the final output table. Note, that only one value will be returned for each outputted level of aggregation so developers will normally only use ID if it returns a unique value at each level of aggregation.

CONCLUSION

The ideas and suggestions discussed above can be leveraged to improve performance of ETL processes in SAS. There is no best approach. Specific differences in system resources, load, data sources and internal data schemas create uniqueness in each environment. However, the developer who understands the strengths and weaknesses of their environment and the rich blend of tools at their disposal can transform an organization's data processes by writing fast efficient code that leverages each system's strengths while minimizing impact of slower or more strained resources.

Inefficient code is costly. Poor coding practices can result in increased hardware requirement to complete tasks, increased licensing fees for those added resources, delays in processing critical requirements, failed processes, time to determine root cause of such failures, and often decreased job satisfaction.

Certainly, there are more tools than we can discuss here. The DS2 (Datastep 2) procedure adds a rich set of multithreaded datastep options but are not leveraged in many environments. For those environments with SAS VIYA installations, the CAS engine can be leveraged for some processing on memory resident tables with little additional effort.

However, the author’s practical experience across many organizations have repeatedly shown that the tools discussed here can be leveraged to greatly improve ETL performance with minimal changes to the existing code and process flow.

CONTACT INFORMATION

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

John Schmitz
Luminare Data LLC
John.schmitz@luminaredata.com
Luminaredata.com