Paper 2004-2015

Your Database Can Do SAS® Too!

Harry Droogendyk, Stratia Consulting Inc.

ABSTRACT

How often have you pulled oodles of data out of the corporate data warehouse down into SAS® for additional processing? Additional processing, sometimes thought to be uniquely SAS, might include FIRST. logic, cumulative totals, lag functionality, specialized summarization or advanced date manipulation. Using the Analytical (or OLAP) and Windowing functionality available in many databases (for example, in Teradata and IBM Netezza), all of this processing can be performed directly in the database without moving and reprocessing detail data unnecessarily.

This presentation illustrates how to increase your coding and execution efficiency by using the database's power through your SAS environment.

INTRODUCTION

SAS is continuing to work closely with leading database vendors to broaden and strengthen the SAS functionality that processes “in-database”. For example, when running a PROC FREQ against a database table via the SAS/Access libname, rather than pulling the detail data into SAS, SAS issues summarizing SQL to the database, counting and grouping as necessary and a small, summary result set is returned to SAS.

However… this paper is *not* about in-database processing. Rather, we’ll identify ways to use native database functionality *instead* of the SAS procedures or programming methods that we are accustomed to. It’s easy to become locked into a way of doing things (after all, it works!) and miss the advantages of new, or different methods.

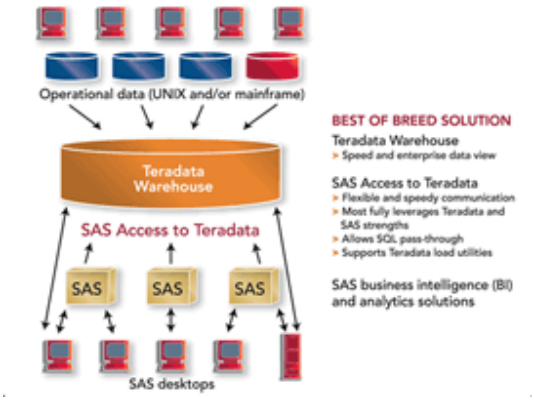
The first section of the paper will explore some reasons why SAS users often shy away from using native SQL database syntax that makes their programs more efficient. We’ll begin with a few simple examples to deal with collation differences, conditional logic and the use of temporary tables. The last portion of the paper will delve into the more complex world of Analytical / OLAP and Windowing functionality to accomplish things we might think are only possible in the SAS world.

While this paper will provide examples for only Teradata and Netezza, these same techniques can be used in many major databases.

THE PROBLEM

TYPICAL ARCHITECTURE

Most organizations have a data warehouse to house and make operational data available to reporting and analytics users. SAS Access software is installed on the desktop or server layer to provide seamless access to the data warehouse database from the SAS environment.



Database SQL CAN SUM DATA – WELL, YEAH, DUH.

Using SAS To Re-Merge Summary & Detail Data

SAS SQL is more forgiving than ANSI SQL. It's often helpful to merge summary data back into the original detail table using a query like below:

```
select name, sex, age, weight, height, sum(weight) as wgt_sum
  from sashelp.class
 group by sex;
```

Because we have more non-aggregate columns on the SELECT than are found in the GROUP BY, SAS will generate the NOTE below and very helpfully merge the summary field(s) back into the detail data.

```
NOTE: The query requires remerging summary statistics back with the original data.
```

	Name	Sex	Age	Weight	Height	wgt_sum
9	Janet	F	15	112.5	62.5	811
10	Philip	M	16	150	72	1089.5

Using the Database To Re-Merge Summary & Detail Data

However, the same syntax running against the database will generate an error because databases require that all non-aggregated columns must appear in the GROUP BY.

```
select * from connection to teradata (
       select name, sex, age, weight, height, sum(weight) as wgt_sum
         from sashelp_class
         group by sex
         );
```

```
ERROR: Selected non-aggregate values must be part of the associated group
```

How does one achieve the mixed SAS detail / summary result in a database query? One could code a sub-query, summing weight and grouping by sex and join the sub-query result set to the detail data by sex. Or, windowing functionality could be used:

```
select * from connection to teradata (
       select name, sex, age, weight, height
          ,   sum(weight)
                    over ( partition by sex )  as wgt_sum
         from sashelp_class
         order by 2,1
       );
```

In the example above:

- SUM() - aggregate function
- OVER - define windowing criteria
- PARTITION - defines grouping column(s)
- ROWS …. - specification is *not* included since we want to sum all rows in the partition
- GROUP BY ?? - not needed ☺

Cumulative Sums

When an insurance claim is opened, the adjuster establishes a reserve amount – setting aside a sum of money required to pay the company's estimated claim obligation. As time moves on the claim matures - additional details emerge, estimates are obtained, claim severity more accurately known and payments made. As each of the claim transactions occur, reserves are adjusted to reflect the anticipated outstanding payment amounts Because reserves are reported as a balance sheet liability, the bean counters are very interested in knowing the sum of outstanding reserves at any point in time. Hence, a requirement to report daily cumulative claims reserves by day.

Given the claims reserves transactions below, what are the total outstanding reserves on April 11, 2015 ?

```
data claims_reserves;
       claim_no=1; trans_dt='04apr2015'd; reserve_amt=  500; note='Open '; output;
       claim_no=1; trans_dt='12apr2015'd; reserve_amt= -300; note='Pymt '; output;
```

```
        claim_no=1; trans_dt='13apr2015'd; reserve_amt=  600; note='Add  '; output;

        claim_no=2; trans_dt='09apr2015'd; reserve_amt= 1200; note='Open '; output;
        claim_no=2; trans_dt='12apr2015'd; reserve_amt= -800; note='Pymt '; output;
        claim_no=2; trans_dt='13apr2015'd; reserve_amt= -400; note='Close'; output;

        format trans_dt date9.;
run;
```

Normally this exercise would require SAS datastep code to:

- create the cumulative outstanding reserve totals by transaction date
- use of LAG function to detect gaps in transaction dates
- DO loop to generate rows for missing transaction dates

The SAS data is loaded into Teradata using the libname / proc append method used in the previous examples. The steps below illustrate how daily outstanding reserves can be calculated exclusively in **Teradata** SQL. The solution below is presented as a series of separate steps for clarity, but they could easily be combined for efficiency. Note that the PERIOD data type and EXPAND ON are features found only in Teradata v13+.

In the first query, cumulative reserve sums are created by transaction date.

```
        execute(
                create volatile table claims_reserves_cumulative as (

                     select claim_no, trans_dt
                        ,   sum(reserve_amt)
                              over ( partition by claim_no
                                        order by trans_dt
                                   rows unbounded preceding ) as os_reserve_amt
                              from claims_reserves
                                    )
                    with data primary index ( claim_no )
                       on commit preserve rows
                       ) by teradata;
```

In the example above:

- SUM() - aggregate function
- OVER - define windowing criteria
- PARTITION - defines grouping column as claim_no, i.e. cumulative sum is within claim_no
- ORDER BY - order rows by transaction date within the partition
- ROWS …. - all rows up to and including the current row are to be included in SUM()

The second query creates the **Teradata** PERIOD datatype field which is used to generate the daily outstanding reserve amounts. The PERIOD datatype contains a range of dates in a single DB column (Note: completely unusable if brought into SAS). The period_dt value will be the starting & ending date for the outstanding reserve amount.

```
        execute(
         create volatile table claims_reserves_period as (

              select claim_no, os_reserve_amt
                 ,   period(trans_dt,
                       coalesce( min(trans_dt)
                                  over ( partition by claim_no
                                            order by trans_dt
                                       rows between 1 following and 1 following
                                    ), current_date )) as period_dt
                from claims_reserves_cumulative
                             )
              with data primary index ( claim_no )
                 on commit preserve rows
```

```
                ) by teradata;
```

In the example above:

- MIN() - aggregate function, looking for the next transaction date for this claim_no
- COALESCE - returns the next transaction date, or the current_date if no more transactions are found
- OVER - define windowing criteria
- PARTITION - defines grouping column as claim_no, i.e. next transaction date is within claim_no
- ORDER BY - order rows by transaction date within the partition
- ROWS …. - only consider the row following the current row

```
execute(
        create volatile table claims_reserves_daily as (
             select claim_no
                 ,  begin(trans_dt2)    as trans_dt
                 ,  os_reserve_amt
               from claims_reserves_period

             expand on period_dt as trans_dt2
                    by interval '1' day
                   for period ( cast ( '2015-04-01' as date ),
                                cast ( '2015-04-14' as date ) )
              )
             with data primary index ( claim_no )
                on commit preserve rows
                ) by teradata;
```

In the example above:

- BEGIN() - Teradata function to return the start period value of a PERIOD data type column
- EXPAND ON - creates a time series based on the period value in the input row
- BY INTERVAL - time series interval, begins with start value of PERIOD value, terminates with end value
- FOR PERIOD - limits the number of rows to be expanded

The final result is shown below. The total outstanding reserve on April 11, 2015 is $1,700.

```
            os_reserve_
 trans_dt            amt
----------------------
<snip>
10APR2015          1700
11APR2015          1700
12APR2015           600
13APR2015           800
```

typical SAS / DB architecture

When confronted with multiple processing options that an environment like the one pictured affords, it's important to remember some useful principles:

- use the correct tool for the job, each has strengths and weaknesses
- do the processing where it makes sense
- do not move data unnecessarily
- do the work once

SAS DATA STEP VS SQL

While from the SAS users' perspective, it's becoming more difficult to draw the lines as in-database functionality continues to advance, it's important that we think through the principles above when developing reporting and analytical solutions. The tendency of many of us (certainly the author falls into this!!) is to stay with the tried and true and use familiar methodologies that have served us well for years. But, technology moves on, vendors grow the

usefulness and utility of their products and we should venture outside our comfort zone and embrace the efficiencies these advancements often provide.

Learning to use the latest database / SQL functionality is especially difficult for the SAS programmer. We love the data step and the ultimate control it provides. We are very comfortable with the step-oriented structure that allows large tasks to be broken into manageable chunks. The implied data step loop is very helpful, but if we want complete control, we can program around that as well and deal with our data, row by row. (as an aside, see Ian Whitlock's very helpful treatment of how the SAS data step thinks at http://www2.sas.com/proceedings/sugi31/246-31.pdf) We can visualize rows of data and two-dimension table structures very easily. SAS reads tables in order, it know the first and last rows of a table or group, it allows us to create arrays across our columns and deal with the bits and bytes of our data very easily. SAS data step allows us to define both *what we want to do* and also *how we want to do it*.

Databases and SQL work very differently – remember Relational Algebra and all the talk of "sets"? Rows are concrete, "sets" are somewhat abstract. And, since databases are generally at least partially normalized, tables typically need to be joined using their common relational keys to create the "result set", e.g. the Employee set is joined to the Department set via department_id.

When performing a table join, the programmer writes SQL to specify the input tables, define the join criteria, any data filters required to limit the result set and the selection criteria. Our SQL defines *what we want to do*, but the database decides *how it wants to do it* based on the mysterious wisdom of its query algorithms.

Massively Parallel Processing (MPP) architectures, used by some databases, adds to the complexity. Multiple processes are running concurrently, each churning away on a slice of the input sets, finally aggregated to produce the result set. *How does THAT work ?!?*

Sets are abstract, SQL seems uncontrollable, Cartesian products can be the unfortunate result of missing a single join criteria, arrgghh… SAS and data step are so much friendlier…. ☺

Figure 2. SQL ?!?! Ba boom

Often it proves just too easy to pull data from the DB into SAS rather than fight with the vagaries of the database and SQL, even if the data volumes are large, the network slow, and storage space sparse…

But, your database can do SAS too! And it's not as difficult as you might think. The remainder of the paper will deal with a number of techniques to allow the database to do the heavy lifting, reduce the movement of data and use database functionality to deal with processes we might think are exclusively within the SAS domain. Some of these areas are brief and quite simple and one might question why they're included - others more involved and complex. In either case, the examples and database techniques presented are drawn from relatively common, recent experience.

STARTING SIMPLE – KEEP IT IN THE DATABASE

WHY CAN'T THE DATABASE SORT CORRECTLY ?

An installation uses Teradata databases to regulate user access to specific views within the corporate data warehouse. Many different databases exist, each containing a number of views. To accommodate the various access levels required, many views are found in more than one database. While attempting to determine the degree of overlap between two databases, a list of views for each database was extracted from Teradata, sorted in a Teradata pass-thru query and SAS MERGE was used to compare the views in each database.

```
proc sql;
  connect to teradata (mode=teradata user=&td_u password="&td_p" );

  create table tables as
     select * from connection to teradata (
           select databasename, tablename
             from dbc.tables
            where databasename in ( 'DDWV01','DDWV04I' )
            order by tablename
       );
quit;
```

```
data idm;
    merge tables ( where = ( databasename = 'DDWV01' )  in = ddwv01  )
          tables ( where = ( databasename = 'DDWV04I' ) in = ddwv04i ) ;
      by tablename;

    flag = ddwv01 + ddwv04i * 2 ;
run;
```

Unexpectedly, the data step merge did not complete successfully, instead throwing an error because the BY variables were out of order:

```
ERROR: BY variables are not properly sorted on data set WORK.TABLES.
ddwv01=1 ddwv04i=0 DatabaseName=DDWV01 TableName=tln_loans FIRST.TableName=1
LAST.TableName=1 flag=. _ERROR_=1 _N_=2207
NOTE: The SAS System stopped processing this step because of errors.
```

Investigation showed that Teradata ignored the case of the TableName column and sorted the column as shown below.

VIEWTABLE: Rmtwork.Tables

	DatabaseName	TableName
2190	DDWV01	TEL_BKG_PYMNT
2191	DDWV01	TEL_BKG_TXN
2192	DDWV01	TFC_MSG_DTL
2193	DDWV01	tln_loans
2194	DDWV01	TLN_LOANS_HIST
2195	DDWV01	TM_ZNE

Figure 3. Teradata case insensitive order

The SAS data step merge does not ignore case and expects incoming BY variables to be in the proper collating sequence. Because of the different sort difference, many users do not trust the Teradata ORDER BY and always sort data extracted from the warehouse in SAS, losing the advantage of Teradata's MPP architecture. Changing the connection string to use mode=ANSI didn't fix the issue. Using the SAS Access Teradata libname had the same problem because SAS passed the sort to Teradata to perform:

```
libname dbc teradata user=&td_u password="&td_p" database=dbc;
proc sort data = dbc.tables ( keep = databasename tablename )
          out = tables2;
         where databasename in ( 'DDWV01','DDWV04I' );
           by tablename ;
run;
```

```
NOTE: Sorting was performed by the data source.
NOTE: There were 2258 observations read from the data set DBC.tables.
      WHERE databasename in ('DDWV01', 'DDWV04I');
```

The best solution to this problem takes advantage of the Teradata option **casespecific** to ensure the ORDER BY clause generates the correct sort order.

```
proc sql;
  connect to teradata (mode=teradata user=&td_u password="&td_p" );

  create table tables as
     select * from connection to teradata (
           select databasename, tablename
             from dbc.tables
            where databasename in ( 'DDWV01','DDWV04I' )
            order by tablename ( casespecific )
      );
quit;
```

The example cited is trivial, but the impact of not using the power of the database to order huge tables is not.

CONDITIONALLY TRANSPOSE A RESULT SET

Reporting activities often require columns to be created from data that ordinarily appear in the table row. Given a call center table like the one on the left below, create a report by date, operator and call center queue, reporting the inbound and outbound calls for each call center queue separately as the report layout on the right depicts.

Table Column	Column Values	Report Layout								
operator_id	integer	Operator		Date	HA		Creditor		General	
call_dt	date	ID	Name		In	Out	In	Out	In	Out
queue_id	'HA','H' – Home & Auto 'CR','C' – Creditor 'GN','G' – General Line									
ib_cnt	integer									
ob_cnt	integer									

Table 1. CALL_DATA columns and desired report layout

Rather than pulling all the **detail** call data from the database into SAS to perform the transpose and summarization, take advantage of the database's power using SQL's CASE / WHEN statements to generate the multiple, summarized columns required for the report, and bring the much smaller **summarized** result set into SAS.

```
create table sum_call_data as
  select * from connection to netezza (
   select c.call_dt, c.operator_id, o.operator_nm
        , sum(case when queue_cd in ( 'HA','H' ) then ib_cnt else 0 end) as ha_ib_cnt
        , sum(case when queue_cd in ( 'CR','C' ) then ib_cnt else 0 end) as cr_ib_cnt
        , sum(case when queue_cd in ( 'GN','G' ) then ib_cnt else 0 end) as gen_ib_cnt
        , sum(case when queue_cd in ( 'HA','H' ) then ob_cnt else 0 end) as ha_ob_cnt
        , sum(case when queue_cd in ( 'CR','C' ) then ob_cnt else 0 end) as cr_ob_cnt
        , sum(case when queue_cd in ( 'GN','G' ) then ob_cnt else 0 end) as gen_ob_cnt
   from  db..call_data           c,
         db..operator            o
  where  c.operator_id = o.operator_id
  group  by 1,2,3
            );
```

The CASE / WHEN statements can be used almost anywhere in SQL, e.g. SELECT, within functions like SUM(), WHERE, HAVING and ORDER, GROUP BY clauses providing very granular control, much like the IF / SELECT statements do in SAS data step. The execution and storage efficiency advantages realized by *not* moving detail data through the pipe from the database into SAS are enormous. Do the work where it makes sense.

SAS WORK VS DATABASE TEMPORARY TABLES

The SAS WORK library is very, very handy. There's no need to define anything, no tables / columns to CREATE, no need to clean up. SAS WORK is there, it's functional and it requires no maintenance – no wonder we love it.

When dealing with complex programming challenges in the database, it's often helpful to create intermediate tables to reduce query complexity, especially when dealing with outer joins. Intermediate tables also allow partial results to be verified and built on before continuing. It's often a temptation to bring the data down into SAS simply because it's easier to create intermediate results in WORK datasets than it is in the database.

In another scenario, a user will sometimes supply a list of account numbers in Excel for which credit card transaction data must be extracted from the datamart. Imagine that the account numbers are 16 digits long and the spreadsheet contains 10,000 accounts. How can those account numbers be used to subset the massive transaction table and produce the report required for those 10K accounts? It's easy enough to import the Excel spreadsheet into SAS. But, 16 bytes x 10,000 is too big to fit into a macro variable, and besides, the IN ( ) condition doesn't perform that well

with large lists. Pull the zillion credit card transactions down into SAS and join to the WORK table created by the Excel import?!?

Database temporary tables are almost as easy to work with as SAS WORK datasets. Since each database vendor seems to have a slightly different implementation for defining and using temporary tables, see the SAS Access and the database vendor's documentation for specifics if the Teradata and Netezza syntax below does not work in your databases.

LIBNAME ACCESS TO TEMPORARY DATABASE TABLES

The easiest way to move small work files between SAS and the database is through the use of the SAS Access libname engine for the database. The libname can point at a production schema or database, but it can also connect to temporary or sandbox areas in the database. The examples below illustrate moving data from a SAS library into a database temporary table.

This is the approach to use to populate a temporary table with the 10K account numbers in the example given at the beginning of this section. Once the temporary table is created on the database and populated with the account numbers, it may be included in a database join just like any other table – keeping the processing on the database where it belongs.

Creating Teradata Temporary Tables using Teradata LIBNAME

Teradata has two different types of temporary tables, Global Temporary Tables (GTT) and Volatile Tables (VT). In my experience, VT are somewhat easier to use. GTT cannot be defined WITH DATA and require a subsequent INSERT step to populate. On the other hand, VT can be defined and loaded on the fly in one step. In addition, the GTT structure will persist after the connection is closed though any data in the table will be deleted. VT disappear entirely once the connection is closed. Regardless of which type of Teradata temporary table is employed, a PRIMARY INDEX should be defined and COLLECT STATS executed against them to assist the query optimizer.

Note that FastLoad and FastExport are not supported for temporary tables due to a Teradata limitation.

```
libname dbtemp teradata user=&td_u password="&td_p" database=user mode=teradata
              connection=global dbmstemp=yes ;

proc append   base = dbtemp.sashelp_class
              data = sashelp.class;
run;
```

From the example above:

- connection = global - temp tables
- dbmstemp = yes - create VT, if this option is omitted create GTT
- proc append is used just as it would be if a SAS libref / dataset were the destination table
  - the destination table will be created if it does not exist
- the VT sashelp_class will persist until the DBTEMP libref is CLEARed

Creating Netezza Temporary Tables Implicitly using Netezza LIBNAME

The Netezza implementation is only different in that the libname engine will be different and the DBMSTEMP parameter is not required.

```
libname dbtemp netezza &netezza_connect database=sandbox connection=global;

proc append   base = dbtemp.sashelp_class
              data = sashelp.class;
run;
```

CREATING TEMPORARY DATABASE TABLES VIA PASS-THRU

Temporary tables can also be created using pass-thru EXECUTE (CREATE ….) syntax. The SQL within the EXECUTE parenthesis must be native to the database.

Creating Teradata Volatile Tables

```
proc sql;
```

```
  connect to teradata ( user=&td_u password="&td_p" mode=teradata connection=global );

  execute(
    create volatile table ls_referrals as (

       select app_key_id
       ,      app_cse_num
       ,      app_sts_cd
       ,      app_sts_dt

       ,      sum(case when note_type_cd = 'Refer'  then 1 else 0 end) as ref_cnt
       ,      sum(case when note_type_cd = 'Manual' then 1 else 0 end) as man_cnt
       ,      sum(case when note_type_cd = 'Other'  then 1 else 0 end) as oth_cnt

         from  ls_policy_inter
       where note_type_cd in ( 'Refer', 'Manual', 'Other')
       group by 1,2,3,4

             )
       with data primary index ( app_key_id )

         on commit preserve rows
       ) by teradata;
quit;
```

In the example above:

- CONNECT clause
  - connection = global - temp tables
  - mode = teradata - automatically issues COMMIT WORK
- create VOLATILE table
- with data - defines *and* populates table
- primary index (app_key_id) - align indexes to other tables to be joined to in downstream queries

Creating Netezza Temporary Tables

```
proc sql;
       connect to netezza ( &netezza_connect database = sandbox connection=global );

       execute ( create temporary table case_open_dt as (

              SELECT parent.casenumber          as claim_casenumber
                   , child.casenumber           as benefit_casenumber
                   , child.open_dt

               from  fineos..voccase            parent
                           inner join
                     fineos..voccase            child
                on   parent.i = child.i_occase_childcases  )

               distribute on ( claim_casenumber, benefit_casenumber )

              ) by netezza;
quit;
```

In the example above:

- CONNECTION = GLOBAL - required for temporary tables
- create TEMPORARY
- distribute on
  - where possible, use same distribution key as tables to be joined to

Retaining Temporary Tables Across Steps

One of the helpful features of temporary database tables is the automatic clean-up that occurs when the connection between SAS and database is broken. However, there are times when it's necessary to create a temporary table, do additional processing in a non-SQL step and then use the temporary table again in a subsequent step. To prevent the temporary table from disappearing when the SQL step disconnects from the database (either explicitly or via the quit;), *first* establish a libref to the database temporary table area *before* creating any temporary tables. Doing so will ensure all temporary tables are preserved until the libref is cleared.

```
libname dbtemp netezza &netezza_connect database=sandbox connection=global;

proc sql;
       connect to netezza ( &netezza_connect database=sandbox connection=global );
       execute ( create temporary table case_open_dt as (
              …
              ) by netezza;
quit;

data dbtemp.case_history;
       set jibber_jabber;
       *....;
run;

proc sql;
       connect to netezza ( &netezza_connect database=sandbox connection=global );

       create table final_result as
              select * from connection to netezza (
                     select a.*, h.jibber, h.jabber
                       from case_open_dt        a,
                            case_history        h
                      where a.case_no    = h.case_no
                     );
quit;

libname dbtemp clear;
```

Only once libref DBTEMP is cleared are all the temporary tables deleted, regardless if they were created via the database libref or pass-thru CREATE.

REALLY TEMPORARY "TABLES" – USING WITH CLAUSE

Most databases support the WITH syntax to create *really* temporary "tables" on the fly (Teradata supports this feature starting in v14). The WITH clause allows the programmer to materialize sub-queries and write more readable code that is easier to maintain. They are very helpful in breaking complex queries into manageable chunks that are simpler to develop. The result of the WITH clause is persistent only through the query.

"WITH" is only stated once before the first materialized sub-query. Materialized sub-queries must be contained within parenthesis, commas separating subsequent sub-queries.

```
proc sql noprint;
  connect to netezza (&netezza_connect database=sandbox connection=global );

  execute ( create temporary table polmst_t  as (

       with polmst_int as (
        select p.pol_no, p.pol_exp_dt, p.pol_tran_no, pe.blah

              from ha..policy                 p
                     left join
                   stg_ha..pending            pe
                on p.pol_no = pe.pol_no
              ),
```

```
        auto_and_prop_int as (

            select a.pol_no, a.pol_exp_dt
                 , min(case when m.prop_type_cd in ('01','09','10')     then '1HO'
                            when m.prop_type_cd in ('02')               then '2CO'
                            when m.prop_type_cd in ('03')               then '3TE'
                                                 else '4XX' end)     as ptype
                   from   <snip>
            )

       select po.*, substr(pr.ptype,2,2) as ptype
         from polmst_int                 po
                    left join
              auto_and_prop_int          pr
           on po.pol_no           = pr.pol_no
          and po.pol_exp_dt       = pr.pol_exp_dt
           )
              distribute on ( pol_no )
       ) by netezza;

quit;
```

In the example above:

- WITH is stated once
- first materialized sub-query is named POLMST_INT
- second is named AUTO_AND_PROP_INT, preceded by a comma
- the “real” query starts immediately after the closing parenthesis of the last materialized sub-query.

RAMPING UP! OLAP, ANALYTICAL AND WINDOWING FUNCTIONALITY

One of the perceived advantages of a step-wise, row-oriented language like SAS data step is the ability to wrap one’s head around the data structures as they are processed row by row. SQL can deal with result sets in a row-wise fashion as well, but the programmer must progress into more complex functionality, often named OLAP / Analytical or Windowing capabilities. In this next section we’ll look at how databases can do a number of things that one might think are exclusively in the SAS domain:

- first. / last.
- lag()…, and lead(), which many wish SAS *did* have ☺
- cumulative sums, including those that appear on detail rows

In “normal” SQL, the familiar GROUP BY aggregations summarize multiple selected rows into a single output row. Window functions differ in that they allow an aggregate-like function to operate over some portion of rows selected by a query, but each of those rows remain separate in the query output and may be “attached” to the detail row output.

Though variations exist between databases, the syntax of window function calls generally have the following convention:

function_name (*expression*) OVER (*window_definition),* where:

function_name – e.g. MIN, MAX, AVG, COUNT etc… (DBs may have additional non-ANSI choices)

window_definition – optional clauses include:

PARTITION BY *expression* – similar to GROUP BY, define grouping column(s)

ORDER BY *expression*

frame_clause – defines the rows considered in the current partition, *after* ordering

RANGE | ROWS BETWEEN *frame_start* AND *frame_end*

where *frame_start* and *frame_end* can be:

UNBOUNDED PRECEDING | *value PRECEDING*

CURRENT ROW

value FOLLOWING | UNBOUNDED FOLLOWING

The *function_name* operates on the rows:

- in the partition created by the PARTITION BY clause
- ordered by the ORDER BY specification
- limited by the optional *frame_clause*.

The examples below will (hopefully !!) make this more clear.

DATABASES CAN DO FIRST.*BY_VAR*

Programmers coming from another language to SAS are sometimes a little overwhelmed at the number of tools, or coding choices, found in the SAS "toolbox" to accomplish a given task. One of the handier tools in the toolbox is the control break handling functionality that falls out of *first.by_var* and *last.by_var* syntax.

SQL has min() and max() functions, but if the same min/max value is found on multiple rows, how does one get that *one, unique* row required? A *PILE* of data is pulled out of datamarts and down into SAS simply so first. and last. syntax can be employed to weed out duplicate data.

The familiar SAS syntax below will find the oldest boy and the oldest girl in the SASHELP.CLASS table (names sorted alphabetically where age is the same). The OUTPUT window results are below the code.

```
proc sort data = sashelp.class
          out = class;
           by sex descending age name;
run;

data unique_class;
       set class;
        by sex;
       if first.sex;
run;

title 'first. - SAS';
proc print data = unique_class;
run;
```

```
first. - SAS

Obs     Name     Sex    Age    Height    Weight

 1     Janet      F      15     62.5      112.5
 2     Philip     M      16     72.0      150.0
```

To demonstrate the same ability within the database, a Teradata volatile table is created and populated with the contents the SASHELP.CLASS dataset.

```
libname dbtemp teradata user=&td_u password="&td_p" database=sandbox mode=teradata
                    connection=global dbmstemp=yes ;

proc append  base = dbtemp.sashelp_class
             data = sashelp.class;
run;

execute ( create volatile table unique_class as (
       select * from sashelp_class

         qualify row_number() over
              ( partition by sex order by age desc, name ) = 1

              ) with data  on commit preserve rows
```

```
) by teradata;
```

In the example above:

- QUALIFY - limit the result set, analogous to HAVING in a “normal” summary query
- ROW_NUMBER()- assign a unique number to each row to which it is applied
- OVER - define the windowing criteria
- PARTITION - similar to GROUP BY, defines grouping column(s)
- ORDER BY - sort result set *before* QUALIFY is applied
    - must explicitly specify ALL columns to be ordered
    - Teradata doesn’t have EQUALS default like SAS

Or, in other words, order the rows by sex, descending age and ascending name, partition or group by sex and return the first row within each partition. Your database ***can*** do SAS. ☺

Would `qualify rank() over ( partition by sex order by age desc ) = 1` have the same result? No, since the oldest female age is 15 and both Mary and Jane are 15 years old, *both would appear in the first rank*. Only row_number() = 1 returns the same result as first.sex in the SAS example.

LAG() AND LEAD() FUNCTIONALITY

It’s sometimes necessary to compare the values of a column in the current row or observation with values of the same (or different) column in the previous or next row. SAS users can employ the LAG function to return the column values of previous observations (though SAS users have long requested a LEAD function, it does not yet exist). However, LAG and LEAD functionality *is* available using database windowing techniques.

Using the SAS LAG

SAS LAG example (note LAG *must not* be executed conditionally or results will be incorrect).

```
data lag_lead;
       item_id = 1;  captr_dt = '31mar2015'd;   sales=20000;  output;
       item_id = 1;  captr_dt = '30apr2015'd;   sales=22000;  output;
       item_id = 2;  captr_dt = '28feb2015'd;   sales=12345;  output;
       item_id = 2;  captr_dt = '31mar2015'd;   sales=14210;  output;
       item_id = 2;  captr_dt = '30apr2015'd;   sales=13299;  output;
       format captr_dt date9.;    * for NZ load;
run;

data lag_results;
       set lag_lead;
        by item_id;

       prev_month_sales = lag(sales);
       if first.item_id then
             prev_month_sales=.;
run;

proc print data = lag_results;
run;
```

```
                                 prev_
                                month_
item_id     captr_dt    sales     sales

   1       31MAR2015    20000         .
   1       30APR2015    22000     20000
   2       28FEB2015    12345         .
   2       31MAR2015    14210     12345
   2       30APR2015    13299     14210
```

LAG using Windowing Functionality

The SAS dataset lag_lead is created in Netezza using the same libname / append method employed in the *first.var* example.

```
proc sql;
       connect to netezza ( &netezza_connect connection=global database=sandbox );
       select * from connection to netezza (
              select item_id, captr_dt, sales
              ,      min(sales)
                            over ( partition by item_id
                                        order by captr_dt
                            rows between 1 preceding and 1 preceding
                     ) as prev_month_sales
                from lag_lead
               order by 1,2
               );
quit;
```

In the example above:

- MIN() - aggregate function
- OVER - define windowing criteria
- PARTITION - defines grouping column(s), item_id
- ORDER BY - sorts result set in captr_dt order *before* ROWS… is applied
- ROWS …. - defines the subset of rows to be passed to the aggregate function
  - between 1 preceding and 1 preceding = the previous row
  - if this is the first captr_dt in the item_id partition, null will be returned since there's no previous row

```
                                  PREV_MONTH_
 ITEM_ID   CAPTR_DT     SALES          SALES
-----------------------------------------
       1  31MAR2015     20000              .
       1  30APR2015     22000          20000
       2  28FEB2015     12345              .
       2  31MAR2015     14210          12345
       2  30APR2015     13299          14210
```

LEAD using Windowing Functionality

To effect LEAD functionality, the only coding **difference** is the "**following**" in the *frame_clause*.

```
       min(sales)
              over ( partition by item_id
                        order by captr_dt
                     rows between 1 following and 1 following
                     ) as next_month_sales
```

```
                                  NEXT_MONTH_
 ITEM_ID   CAPTR_DT     SALES          SALES
-----------------------------------------
       1  31MAR2015     20000          22000
       1  30APR2015     22000              .
       2  28FEB2015     12345          14210
       2  31MAR2015     14210          13299
       2  30APR2015     13299              .
```

Accessing the Nth LAG / LEAD using Windowing Functionality

If a row other than the immediately adjacent row is required, the **appropriate numeric value** is specified before "preceding" or "following".

```
min(sales)
      over ( partition by item_id
                order by captr_dt
              rows between 2 preceding and 2 preceding
              ) as prev_prev_month_sales
```

Rolling Average Over X Months

Rolling averages are often used to smooth out the volatility from period to period. In this case, the rolling 3 month average is calculated from the preceding, current and following row, within item_id, ordered by captr_dt.

```
avg(sales)
      over ( partition by item_id
                order by captr_dt
              rows between 1 preceding and 1 following
              ) as avg_sales_rolling_3_mths
```

ITEM_ID	CAPTR_DT	SALES	AVG_SALES_ROLLING_3_MTHS
1	2015-03-31	$20,000	$21,000
1	2015-04-30	$22,000	$21,000
2	2015-02-28	$12,345	$13,278
2	2015-03-31	$14,210	$13,285
2	2015-04-30	$13,299	$13,755

DATABASE SQL CAN SUM DATA – WELL, YEAH, DUH.

Using SAS To Re-Merge Summary & Detail Data

SAS SQL is more forgiving than ANSI SQL. It's often helpful to merge summary data back into the original detail table using a query like below:

```
select name, sex, age, weight, height, sum(weight) as wgt_sum
  from sashelp.class
 group by sex;
```

Because we have more non-aggregate columns on the SELECT than are found in the GROUP BY, SAS will generate the NOTE below and very helpfully merge the summary field(s) back into the detail data.

```
NOTE: The query requires remerging summary statistics back with the original data.
```

	Name	Sex	Age	Weight	Height	wgt_sum
9	Janet	F	15	112.5	62.5	811
10	Philip	M	16	150	72	1089.5

Using the Database To Re-Merge Summary & Detail Data

However, the same syntax running against the database will generate an error because databases require that all non-aggregated columns must appear in the GROUP BY.

```
select * from connection to teradata (
      select name, sex, age, weight, height, sum(weight) as wgt_sum
        from sashelp_class
        group by sex
```

```
        );
```

ERROR: Selected non-aggregate values must be part of the associated group

How does one achieve the mixed SAS detail / summary result in a database query? One could code a sub-query, summing weight and grouping by sex and join the sub-query result set to the detail data by sex. Or, windowing functionality could be used:

```
select * from connection to teradata (
      select name, sex, age, weight, height
         ,   sum(weight)
                  over ( partition by sex )  as wgt_sum
        from sashelp_class
        order by 2,1
      );
```

In the example above:

- SUM() - aggregate function
- OVER - define windowing criteria
- PARTITION - defines grouping column(s)
- ROWS …. - specification is *not* included since we want to sum all rows in the partition
- GROUP BY ?? - not needed ☺

Cumulative Sums

When an insurance claim is opened, the adjuster establishes a reserve amount – setting aside a sum of money required to pay the company's estimated claim obligation. As time moves on the claim matures - additional details emerge, estimates are obtained, claim severity more accurately known and payments made. As each of the claim transactions occur, reserves are adjusted to reflect the anticipated outstanding payment amounts Because reserves are reported as a balance sheet liability, the bean counters are very interested in knowing the sum of outstanding reserves at any point in time. Hence, a requirement to report daily cumulative claims reserves by day.

Given the claims reserves transactions below, what are the total outstanding reserves on April 11, 2015 ?

```
data claims_reserves;
      claim_no=1; trans_dt='04apr2015'd; reserve_amt=  500; note='Open '; output;
      claim_no=1; trans_dt='12apr2015'd; reserve_amt= -300; note='Pymt '; output;
      claim_no=1; trans_dt='13apr2015'd; reserve_amt=  600; note='Add  '; output;

      claim_no=2; trans_dt='09apr2015'd; reserve_amt= 1200; note='Open '; output;
      claim_no=2; trans_dt='12apr2015'd; reserve_amt= -800; note='Pymt '; output;
      claim_no=2; trans_dt='13apr2015'd; reserve_amt= -400; note='Close'; output;

      format trans_dt date9.;
run;
```

Normally this exercise would require SAS datastep code to:

- create the cumulative outstanding reserve totals by transaction date
- use of LAG function to detect gaps in transaction dates
- DO loop to generate rows for missing transaction dates

The SAS data is loaded into Teradata using the libname / proc append method used in the previous examples. The steps below illustrate how daily outstanding reserves can be calculated exclusively in **Teradata** SQL. The solution below is presented as a series of separate steps for clarity, but they could easily be combined for efficiency. Note that the PERIOD data type and EXPAND ON are features found only in Teradata v13+.

In the first query, cumulative reserve sums are created by transaction date.

```
      execute(
             create volatile table claims_reserves_cumulative as (

                  select claim_no, trans_dt
                     ,   sum(reserve_amt)
```

```
                    over ( partition by claim_no
                             order by trans_dt
                         rows unbounded preceding ) as os_reserve_amt
                    from claims_reserves
                         )
                with data primary index ( claim_no )
                   on commit preserve rows
                   ) by teradata;
```

In the example above:

- SUM() - aggregate function
- OVER - define windowing criteria
- PARTITION - defines grouping column as claim_no, i.e. cumulative sum is within claim_no
- ORDER BY - order rows by transaction date within the partition
- ROWS …. - all rows up to and including the current row are to be included in SUM()

The second query creates the **Teradata** PERIOD datatype field which is used to generate the daily outstanding reserve amounts. The PERIOD datatype contains a range of dates in a single DB column (Note: completely unusable if brought into SAS). The period_dt value will be the starting & ending date for the outstanding reserve amount.

```
execute(
 create volatile table claims_reserves_period as (

      select claim_no, os_reserve_amt
         ,   period(trans_dt,
               coalesce( min(trans_dt)
                         over ( partition by claim_no
                                  order by trans_dt
                                rows between 1 following and 1 following
                          ), current_date )) as period_dt
        from claims_reserves_cumulative
                    )
      with data primary index ( claim_no )
         on commit preserve rows
         ) by teradata;
```

In the example above:

- MIN() - aggregate function, looking for the next transaction date for this claim_no
- COALESCE - returns the next transaction date, or the current_date if no more transactions are found
- OVER - define windowing criteria
- PARTITION - defines grouping column as claim_no, i.e. next transaction date is within claim_no
- ORDER BY - order rows by transaction date within the partition
- ROWS …. - only consider the row following the current row

```
execute(
 create volatile table claims_reserves_daily as (
      select claim_no
          ,  begin(trans_dt2)    as trans_dt
          ,  os_reserve_amt
        from claims_reserves_period

      expand on period_dt as trans_dt2
             by interval '1' day
            for period ( cast ( '2015-04-01' as date ),
                         cast ( '2015-04-14' as date ) )
       )
      with data primary index ( claim_no )
         on commit preserve rows
         ) by teradata;
```

In the example above:

- BEGIN() - Teradata function to return the start period value of a PERIOD data type column
- EXPAND ON - creates a time series based on the period value in the input row
- BY INTERVAL - time series interval, begins with start value of PERIOD value, terminates with end value
- FOR PERIOD - limits the number of rows to be expanded

The final result is shown below. The total outstanding reserve on April 11, 2015 is $1,700.

```
            os_reserve_
 trans_dt            amt
----------------------
<snip>
10APR2015          1700
11APR2015          1700
12APR2015           600
13APR2015           800
```

CONCLUSION

Since SAS programmers are very comfortable with the flexibility of the familiar datastep, it's a real temptation to pull large amounts of detail data out of the database into SAS for additional processing that we might think is uniquely SAS functionality. But database capability continues to evolve and it's often possible to do most, if not all, data preparation and manipulation in the database. Leveraging the power of the database reduces data movement, network traffic and increases programmer and execution efficiency. **Do stuff where it makes sense !**

It's vital to maintain an inquisitive nature and be willing to stretch yourself to learn new and better techniques. While it's often easier to lapse into the tried and true methods of yesteryear, there's great advantage in exploiting the strength of your tools, especially those you've not dared to try before.

REFERENCES

SQL Data Manipulation Language, Release 14.0, July 2013, Teradata Corporation

http://www.postgresql.org/docs/current/static/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS

CONTACT INFORMATION

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

Name: Harry Droogendyk
Enterprise: Stratia Consulting Inc.
E-mail: conf@stratia.ca
Web: www.stratia.ca