Paper 19-2011

Teradata_® for the SAS_® Programmer: Ordered Analytical Functions, Hash Objects, ANSI SQL: 2003 Rick Andrews, Centers for Medicare and Medicaid Services

Jignesh Miyani, Teradata Corporation

ABSTRACT

The manner and speed in which information is hashed and indexed makes Teradata a valuable tool for satisfying time sensitive Congressional requests at the Centers for Medicare and Medicaid Services (CMS). Ordered Analytical Functions enable On-Line Analytical Processing (OLAP) requests containing ANSI SQL: 2003 window functions, which greatly enhance the functionality of the Structured Query Language (SQL). This paper will discuss how SAS 9.1 programming techniques can be imitated within the Teradata 12 environment using SAS\ACCESS® passthough queries and the simulation of By-Group processing with dot notation. The methods discussed will help users make an easier transition from traditional SAS programming to a Massively Parallel Processing (MPP) Teradata platform running shared nothing architecture.

TOPICS

- **Ordered Analytical Functions**
- Structured Query Language
- ANSI SQL:1999 vs. 2003
- By-Group Processing
- **Binary Search Trees** •
- SAS Hash Objects
- Teradata Hash Buckets
- In-database Processing •
- Massively Parallel Processing

INTRODUCTION

Using the right tool for the right job is the main reason for learning more about the Teradata environment. By using MPP, the system can utilize the latest advances in data processing technologies. In computer architecture, the designation MPP refers to a computing system with many independent units or entire microprocessors that run in parallel. Within this class of computing, all units of processing elements are connected to become one very large computer. This is in contrast to the distributed computing system whereby

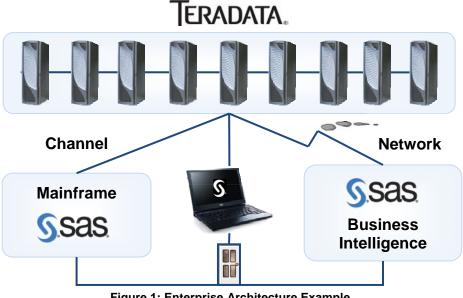


Figure 1: Enterprise Architecture Example

a massive number of separate computers are used to solve a single problem. In shared nothing architecture, a symmetric multiprocessing node is independent and self-sufficient, and no single point of contention exists across the entire system, creating one of the fastest means of data processing.

ANSI STANDARDS

Learning Teradata also means learning SQL. The SQL language was first created by IBM® in the early 1970's and was called SEQUEL for "Structured English Query Language". SQL was adopted as a standard by the American National Standards Institute (ANSI) in 1986 and into the SAS System's SQL procedure in 1990. The standard used in SAS 9.1 is SQL: 1999, also known as SQL3, the fourth revision of the SQL database guery language. Teradata 12 uses SQL: 2003, which contains the Ordered Analytical Functions that are also known as window functions and do not currently exist in PROC SQL. The latest revision of the standard is SQL: 2008, which is not discussed.

NOMENCLATURE

The tables to the right display some typical nomenclature for SAS and SQL terminology. These values will be used interchangeably throughout this paper. SAS and SQL code will be presented in uppercase letters while usersupplied variable names and hard-coded values will be presented in mixed case.

Table 1: Concepts				
SAS	SQL			
Data Set	Table			
Observation	Row			
Variable	Column			
Program	Query			

Table	2:	Coding
-------	----	--------

	SAS	SQL	
	DATA	CREATE	
SET		FROM	
	MERGE	JOIN	
	IF	CASE	

SAS HASH OBJECT

In SAS, the hash object is used to create an Adelson-Velskii and Landis (AVL) tree, a self-balancing binary search tree. In an AVL tree, the heights of the two child sub-trees of any node differ by at most one; therefore, it is said to be height-balanced (Sedgewick, 1983). The simple AVL tree shown in Figure 2 identifies how a search of the tree on the left would require a traverse of two limbs to obtain data located in node number 3. After the tree is heightbalanced, it takes only one traverse.

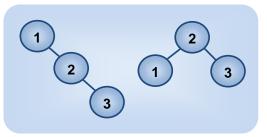


Figure 2: Simple AVL Tree

4

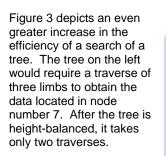
5

3

6

7

2



The SAS hash object is an illustration of how data can be balanced into Random

Figure 3: Complex AVL Tree

7

1

Access Memory (RAM) in order to read only the information required to meet a specific request. Using this method has been shown to greatly increase the speed of match-merges and lookup operations, though it is limited to the amount of available memory.

6

5

2

TERADATA HASH BUCKET

In Teradata, information is processed through a sophisticated hashing algorithm and automatically distributed across Access Module Processes (AMPs), which control database processing. Each AMP is assigned an equal portion of the data and will execute all necessary database functions for the information allocated to it - each with independent file systems maintaining separate data structures, space management, cache management, and concurrency control functions. The file system interface allows for accessing and manipulating rows one at a time, and manipulating single data blocks, and entire tables or sub-tables (Clark 2000).

The ability to attach additional nodes makes Teradata extremely scalable. Parallelism is built deep into the Teradata solution, with each parallel unit acting as a self-contained minidatabase management system. This local autonomy eliminates extra CPUs often required by other parallel systems, where databasespecific functions are consciously coordinated, split up, or collected (Ballinger, 2001).

Scalable Performance Data Server® is the SAS implementation of this type of parallel processing architechure, which focuses on partitioning, parallel processing, unique indexing technology, query optimization, the SAS SQL Pass-Through facility, data management and security.

Hashing is a major reason for the speed of Teradata, which is a key motive for using

the platform. Another rationale is the use

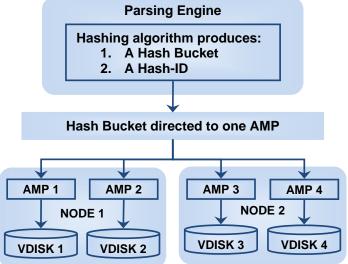


Figure 4: Teradata Database Architecture

of Ordered Analytical Functions contained within ANSI SQL: 2003, which eases the burden of additional code generation. These functions can be used for a variety of operations and can make programming much easier. A final incentive is the SAS in-database initiative in concert with the Teradata partnership, to be discussed later.

STRUCTURED QUERY LANGUAGE

Most analytical users of SQL utilize these six			
basic clauses, though many others exist for			
use by database administrators and data			
modelers. A rudimentary knowledge of these			
clauses is assumed and will not be discussed.			

ORDERED ANALYTICAL FUNCTIONS

Ordered Analytical Functions include ANSI SQL-2003-compliant window functions. A window function is similar to an aggregate function, though it is applied to a partition of a result set in lieu of the entire table. A window is specified by the OVER phrase, which can include the following phrases: PARTITION BY, ORDER BY, and ROWS {BETWEEN}.



SELECT

HAVING

FROM WHERE

- RANK
- REGR_AVGX

GROUP BY - classifies a collection or "group" of data

ORDER BY - ascertains the sort order for the final output

- REGR_AVGY
- REGR_COUNT
- REGR_COUNT

- statement for selecting desired variables

- criteria for limiting the data being read

- identifies the data sets to be merged together

- filters records resulting from the GROUP BY clause

- REGR_INTERCEPT
- REGR_R2
- REGR_SLOPE
- REGR SXX
- REGR_SXY
- REGR_SYY
- ROW_NUMBER
- STDDEV_POP
- STDDEV_SAMP
- SUM
- VAR_POP
- VAR_SAMP

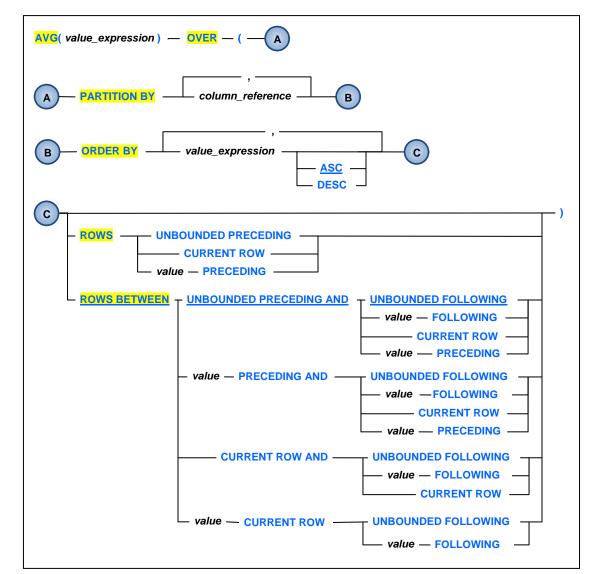


Figure 5: Window Function Diagram

SUMMARY EXAMPLES

The following SELECT statement contains a window function, which does not include a PARTITION BY phrase. The OVER clause, containing no phrases, computes a summary of all records, whereby the total paid amount of all rows is created and placed on each observation.

SELECT Bene_Id, Thru_Dt, CIm_Type, Paid_Amt, SUM(Paid_Amt) OVER () AS Total_Paid FROM Mdcr_CIm	
---	--

Figure 6: Window Function without Clauses

For comparison, Figure 7 is used to show one method of performing this function using a typical SAS 9.1 data step. The END option of the SET statement is used to identify the end of the file. The paid amount is accumulated into the total paid variable and is output when the end of the file is found. The second data step retains the total paid value on each

DATA Deri ved_Table; SET Mdcr_Clm END=EndOfFile; Total_Paid + Paid_Amt; IF EndOfFile THEN OUTPUT; RUN; DATA Mdcr_Final; SET Mdcr_Clm; IF _N_=1 THEN SET Derived_Table; RUN;

Figure 7: OVER () Simulation using SAS 9.1

PARTITION BY PHRASE

The PARTITION BY clause determines the group, or groups, over which the ordered analytical function executes. If there is no PARTITION BY phrase, then the entire result set, delivered by the FROM clause, is output as a single group, or partition, over which the ordered analytical function executes (Teradata, 2010a).

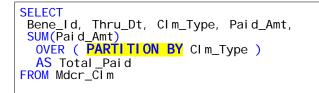


Figure 9: Window Function with Partition-By

The example in Figure 9 is a partition of the variable claim type, which corresponds to the groups created in Table 4. Notice the Inpatient (IP) records are now summarized with a total paid of \$1,250 with no regard to the beneficiary.

Figure 10 is a simulation of the output in Table 4 using "By-Group" processing in SAS 9.1. When the first record of a By-Group is found, the accumulator variable is set to zero, then incremented by the paid amount until the last record of the By-Group is found and then output into a derived table. The records are then merged back into the original table.

Table 3: OVER Clause without Phrases					
Bene Id	Thru Date	Clm Type	Paid Amt	Total Paid	
1	01-Jan-11	IP	\$500	\$2300	
1	05-Jan-11	SNF	\$300	\$2300	
1	30-Jan-11	PHY	\$175	\$2300	
2	02-Feb-11	IP	\$750	\$2300	
2	14-Feb-11	SNF	\$300	\$2300	
2	21-Feb-11	PHY	\$125	\$2300	
2	17-Mar-11	HHA	\$150	\$2300	

record. The example in Figure 8 also produces the same result using the ANSI SQL: 1999 standard. The derived table contains total paid and places it on each record by way of the inner join, implied by the comma within the FROM clause.

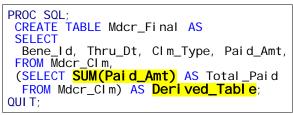


Figure 8: Simulation using ANSI SQL: 1999

Table 4:	Partition	by (Claim	Туре
----------	-----------	------	-------	------

Bene Id	Thru Date	Clm Type	Paid Amt	Total Paid
2	17-Mar-11	HHA	\$150	\$150
1	01-Jan-11	IP	\$500	\$1,250
2	02-Feb-11	IP	\$750	\$1,250
1	30-Jan-11	PHY	\$175	\$300
2	21-Feb-11	PHY	\$125	\$300
1	05-Jan-11	SNF	\$300	\$600
2	14-Feb-11	SNF	\$300	\$600

```
PROC SORT DATA=Mdcr_CIm;
  BY CIm_Type;
```

DATA Deri ved_Table; SET Mdcr_Clm; BY CIm_Type; KEEP CIm_Type Total_Paid; IF <mark>FIRST</mark>.CIm_Type THEN Total_Paid=0; Total_Paid + Paid_Amt; IF **LAST**. CIm_Type THEN OUTPUT; DATA Final Table; MERGE Mdcr_CIm Derived_Table; BY CIm_Type; RUN:

Figure 10: Partition-By Simulation using SAS 9.1

ORDER BY PHRASE

The ORDER BY phrase specifies how the rows are ordered in a partition, which determines the sort order of the rows over which the function is applied (Teradata, 2010a). Figure 11 depicts an example of a window function containing a PARTITION BY phrase without an ORDER BY. The results

in Table 5 are partitioned by beneficiary, though arbitrarily ordered by the paid amount within each partition.

SELECT Bene_Id, Thru_Dt, Clm_Type, Paid_Amt, SUM(Paid_Amt) OVER (PARTITION BY Bene_Id) AS Total_Paid FROM Mdcr_Clm

Figure 11: Partition without an Order-By

There are times when the order might be important. The next example introduces the ORDER BY phrase to the window function to arrange the output in sequence by the thru date for each beneficiary. The total paid amounts are the same in Tables 5 and 6, only the order has changed.

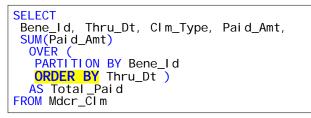


Figure 12: Partition with an Order-By

ROWS & ROWS BETWEEN PHRASES

	Table 5: Partition by Beneficiary					
Bene Id	Thru Date	Clm Type	Paid Amt	Total Paid		
1	30-Jan-11	PHY	\$175	\$975		
1	05-Jan-11	SNF	\$300	\$975		
1	01-Jan-11	IP	\$500	\$975		
2	21-Feb-11	PHY	\$125	\$1325		
2	17-Mar-11	HHA	\$150	\$1325		
2	14-Feb-11	SNF	\$300	\$1325		
2	02-Feb-11	IP	\$750	\$1325		

....

Table 6: Order by Claim Type

		-		
Bene Id	Thru Date	Clm Type	Paid Amt	Total Paid
1	01-Jan-11	IP	\$500	\$975
1	05-Jan-11	SNF	\$300	\$975
1	30-Jan-11	PHY	\$175	\$975
2	02-Feb-11	IP	\$750	\$1325
2	14-Feb-11	SNF	\$300	\$1325
2	21-Feb-11	PHY	\$125	\$1325
2	17-Mar-11	HHA	\$150	\$1325

The ROWS phrase defines the rows over which the aggregate function is computed for each observation in the partition. If ROWS is specified, the computation of the aggregate function for each record in the partition includes only the subset of rows in the ROWS phrase. If there is no ROWS phrase, the computation includes everything in the partition (Teradata, 2010a).

- **ROWS** The starting point for the partition is the first record in the group. The aggregation group end is always the current row.
- **ROWS BETWEEN** The aggregation group-start and end, which defines a set of rows relative to the current row in the ordering of the rows within the partition. The row specified by the group start must precede the row specified by the group end.
 - UNBOUNDED PRECEDING The entire partition preceding the current row.
 - **UNBOUNDED FOLLOWING** The entire partition following the current row.
 - o CURRENT ROW
- The start or end of the aggregation group as the current row.
 The number of rows preceding the current row.
- value PRECEDING
 value FOLLOWING
 - The number of rows following the current row.

The default action of a window function with no other phrases is to evaluate all rows PRECEDING and FOLLOWING the current row as shown in Figure 13. This is the same as having the OVER () clause with no other phrases as depicted in Figure 6 with the same results as Table 3.

```
SELECT
Bene_Id, Thru_Dt, CIm_Type, Paid_Amt,
SUM(Paid_Amt)
OVER (
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING )
AS Total_Paid
FROM Mdcr_CIm
```

Figure 13: Default Action of Window Function

Total

Paid

\$175

\$475

\$975

\$1,100

\$1,250

\$1,550

\$2,300

UNBOUNDED PRECEDING & CURRENT ROW

Figure 14 illustrates the accumulation of the paid amount from all preceding rows thru the current row shown in Table 7. Note the partitions by beneficiary appear to be arbitrarily sorted by the paid amount.

Bene

ld

1

1

1

2

2

2

2

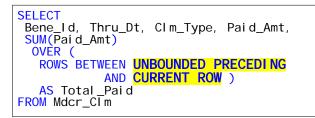


Figure 14: Accumulating Total Dollar Amounts

ROWS, PARTITION, & ORDER BY USAGE

The next example describes the use of PARTITION BY and

ORDER BY in conjunction with the ROWS phrase. The result will be a partition, or grouping, of the beneficiary sorted by the thru date. The ROWS phrase causes the accumulation of the paid amount until the next beneficiary is encountered, whereby the value is reset to zero. Table 8: Output of Figures 15 & 16

SELECT Bene_Id, Thru_Dt, CIm_Type, Paid_Amt, SUM(Paid_Amt) OVER (
PARTITION BY Bene_Id
ORDER BY Thru_Dt
ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW)
AS Total _Paid
FROM Mdcr Clm

Bene Id	Thru Date	Clm Type	Paid Amt	Total Paid		
1	01-Jan-11	IP	\$500	\$ 500		
1	05-Jan-11	SNF	\$300	\$ 800		
1	30-Jan-11	PHY	\$175	\$ 975		
2	02-Feb-11	IP	\$750	\$ 750		
2	14-Feb-11	SNF	\$300	\$1050		
2	21-Feb-11	PHY	\$125	\$1175		
2	17-Mar-11	HHA	\$150	\$1325		

Table 7: Accumulation Example

Clm

Туре

PHY

SNF

IP

PHY

HHA

SNF

IP

Paid

Amt

\$175

\$300

\$500

\$125

\$150

\$300

\$750

Thru

Date

30-Jan-11

05-Jan-11

01-Jan-11

21-Feb-11

17-Mar-11

14-Feb-11

02-Feb-11

Figure 16 uses By-Group processing in SAS 9.1 to create

Figure 15: Accumulation using Order By

the total paid amount using FIRST dot notation. The data must first be sorted by beneficiary and thru date to put the data in the correct order. The BY statement in the DATA step creates two temporary variables called FIRST.Bene Id and LAST.Bene Id. They are boolean expressions identifying the first and last records within the group, which can be used to conditionally control output.

PROC SORT DATA=Mdcr_Clm; BY Bene_Id Thru_Date; RUN;
DATA Derived_Table; SET Mdcr_Clm; BY Bene_Id; IF FIRST.Bene_Id THEN Total_Paid = 0; Total_Paid + Paid_Amt; OUTPUT; RETURN; RUN;

Table 9: First Dot Variables

Bene Id	Thru Date	Clm Type	FIRST. Bene Id	LAST. Bene Id
1	01-Jan-11	IP	1	0
1	05-Jan-11	SNF	0	0
1	30-Jan-11	PHY	0	1
2	02-Feb-11	IP	1	0
2	14-Feb-11	SNF	0	0
2	21-Feb-11	PHY	0	0
2	17-Mar-11	HHA	0	1

Figure 16: First Dot Notation using SAS 9.1

HAVING CLAUSE

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions. It was not added as part of ANSI SQL: 2003, but is used below to create an example of the new QUALIFY phrase discussed later. It includes a predicate used to filter rows resulting from the GROUP BY clause. Because it acts on the results of the GROUP BY clause, aggregation functions can be used in the HAVING clause predicate.

3y-Group

The HAVING clause is often desired in the healthcare industry to keep only those beneficiaries having a full twelve months of coverage. The months of coverage must first be counted and only those records having a count of twelve are kept.

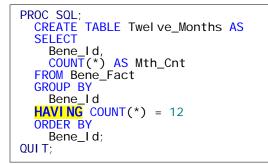


Figure 17: HAVING Clause ANSI SQL: 1999

Figure 18 depicts FIRST dot, LAST dot programming used in By-Group processing within SAS. When the first record of a By-Group is found, the counter is set to zero, then incremented by one until the last record is found. The observation is output only if the count equals twelve.

PROC SORT DATA= Bene_Fact; BY Bene_Id; RUN;
DATA Twelve_Months; SET Bene_Fact; BY Bene_Id; IF FIRST .Bene_Id THEN Mth_Cnt = 0; Mth_Cnt + 1; IF LAST .Bene_Id AND Mth_Cnt = 12; RUN;

Figure 18: Basic Window Function

The results of the Ordered Analytical Function shown in Table 12 have the cumulative total on every record, unlike the incremental accumulation in Table 10 with the addition of the count for the months within each state.

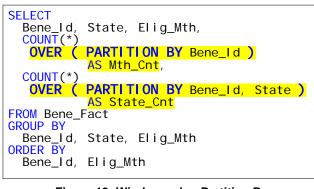


Figure 19: Window using Partition By

Table 12 shows the first beneficiary, known as a snowbird, resided in Florida during the winter months and in New York for the remainder of the year. There

are times when a researcher is interested in recoding the primary state of residence for an individual, while maintaining the total number of eligible months. The following will demonstrate new functionality for this task.

	-				
Bene Id	Elig Mth	Mth Cnt			
1	Jan-08	1	\sum		
1	Feb-08	2			
1	Mar-08	3			
1	Apr-08	4			
1	May-08	5			
1	Jun-08	6			
1	Jul-08	7	$\left(\right)$		
1	Aug-08	8			
1	Sep-08	9			
1	Oct-08	10			

11

12

1

2

18

Table 10: Bene Count by Month

Table	11: Oı	utput	of Fig	jures	17 &
				_	

Nov-08

Dec-08

Jan-08

Feb-08

1

1

2

2

Bene Id	Mth Cnt
1	12

The HAVING clause and By-Group methods have been shown to illustrate the use of grouping records for aggregation. With Ordered Analytical Functions in SQL: 2003, the term *window* is in many ways related to the function of By-Group processing in SAS.

Table 12: Output of Figure 19

Bene Id	Elig Mth	State Cd	Mth Cnt	State Cnt	
1	Jan-08	FL	12	4	
1	Feb-08	FL	12	4	
1	Mar-08	NY	12	8	
1	Apr-08	NY	12	8	
1	May-08	NY	12	8	
1	Jun-08	NY	12	8	
1	Jul-08	NY	12	8	
1	Aug-08	NY	12	8	
1	Sep-08	NY	12	8	
1	Oct-08	NY	12	8	
1	Nov-08	FL	12	4	
1	Dec-08	FL	12	4	
2	Jan-08	CA	2	2	
2	Feb-08	CA	2	2	

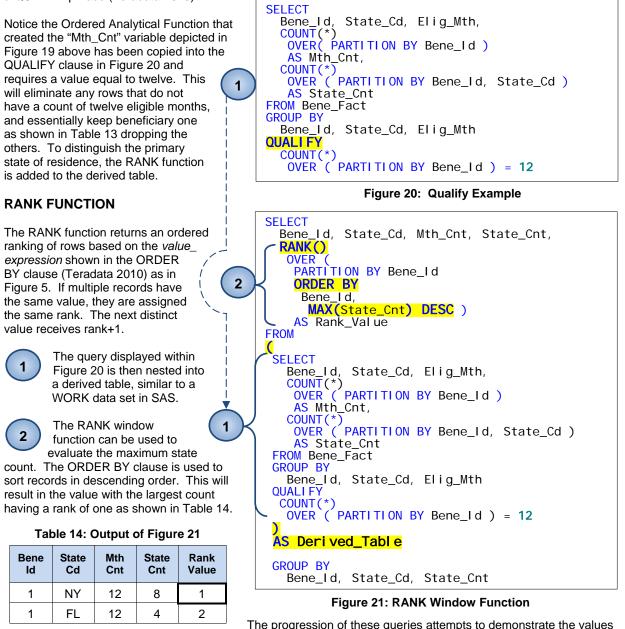
In this case the state of New York is the primary state and the number of eligible months is twelve. The challenge is how to keep only one record per beneficiary, while maintaining the proper count of months and keeping only those

members with twelve months of enrollment as shown in Table 13. Because Ordered Analytical Functions are not allowed in the HAVING clause, the QUALIFY phrase has been created in ANSI SQL: 2003 to allow control of output for window functions.

Table 13: Desired Results			
Bene	State	Mth	
ld	Cd	Cnt	
1	NY	12	

QUALIFY CLAUSE

The QUALIFY clause is analogous to the HAVING clause in ordinary SQL to eliminate unqualified rows. QUALIFY eliminates records based on the function value, returning a new quantity for each of the participating rows. An SQL query containing both Ordered Analytical Functions and Aggregate Functions can have both a HAVING clause and a QUALIFY phrase (Teradata 2010).



obtained by exploiting *window* functions. The desired results shown in Table 13 have yet to be achieved. By moving the RANK function into a QUALIFY clause, after the derived table, a condition can be added to keep the record with a rank value of one. The record will finally contain the number of twelve required eligible months and include the

primary state of residence.

PRIMARY STATE OF RESIDENCE

The RANK window function shown in Figure 21 has been moved into the QUALIFY clause in Figure 22 and *qualified* with a value of one. Along with the qualification of twelve continuous months of enrollment, beneficiary number one has been kept with a primary state of New York as shown in Table 15.

Table 15: Output of Figures 22 & 23

BENE ID	STATE CD	MTH CNT
1	NY	12

As with any program or query, there is usually more than one mechanism to obtain the same results. A more straight-forward or efficient process may exist.

In comparison, a SAS version of this process is shown in Figure 23, though many other SAS techniques exist. The program shown below is meant to provide an example of a method using traditional SAS programming.

PROC SORT DATA=Bene_Fact; BY Bene_Id State_Cd; RUN;

```
DATA Twelve_Months (
       KEEP= Bene_Id State_Cd Mth_Cnt )
     State_Counts (
       KEEP = Bene_ld State_Cd State_Cnt );
  SET Bene_Fact;
 BY Bene_Id State_Cd;
IF FIRST.Bene_Id THEN Mth_Cnt = 0;
     Mth_Cnt + 1;
  IF FIRST State_Cd THEN State_Cnt = 0;
     State_Cnt + 1;
  I F
     LAST. Bene_I d AND Mth_Cnt = 12
     THEN OUTPUT Twelve_Months;
  IF LAST. State Cd
     THEN OUTPUT State_Counts;
RUN:
PROC SORT DATA=State_Counts;
 BY Bene_Id State_Cnt;
RUN:
DATA Max_State;
  SET State_Counts;
  BY Bene_Id State_Cnt;
  IF LAST. Bene_Id;
RUN:
DATA Prim_State_of_Residence;
  MERGE
    Twelve_Months (IN=Keep_It)
    Max_State;
  BY Bene Id;
  IF Keep_It THEN OUTPUT;
RUN:
```

SELECT Bene_Id, State_Cd, Mth_Cnt FROM SELECT Bene_Id, State_Cd, Eliq_Mth, COUNT(*) OVER (PARTITION BY Bene_Id) AS Mth_Cnt, COUNT(*) OVER (PARTITION BY Bene_Id, State_Cd) AS State_Cnt FROM Bene_Fact **GROUP BY** Bene_Id, State_Cd, Elig_Mth QUALI FY COUNT(*) OVER (PARTITION BY Bene_Id) = 12 AS Deri ved_Tabl e

GROUP BY Bene_Id, State_Cd, Mth_Cnt

QUALIFY RANK() OVER (PARTITION BY Bene_Id ORDER BY Bene_Id, MAX(State_Cnt) DESC) = 1

1

2

3

4

Figure 22: Primary State ANSI SQL: 2003

1. The initial FIRST dot in Figure 23 sets a monthly counter to zero whenever a new beneficiary is encountered and then increments the variable by one. The second FIRST dot sets a state counter to zero whenever a new beneficiary is found and increments that variable by one.

2. The initial LAST dot will output to a data set when the monthly counter is equal to twelve for each beneficiary. The second LAST dot will output to a data set containing the state counter for each beneficiary by state.

3. The state counts data set is then sorted by beneficiary and state count. This will place the state with the largest count on the last record. The final LAST dot will keep the last record for each beneficiary.

4. The next step merges the two data sets and keeps only those matching the twelve month criteria.

Neither the SAS nor SQL process takes into account when a tie exists for state counts. Additional code is needed to evaluate the data.

Figure 23: Primary State SAS 9.1 By-Group

IN-DATABASE PROCESSING

Thus far, the methods described in this paper have been related to the latest version of SAS 9.1. In October 2007, SAS and Teradata Corporation announced a strategic partnership that enables exploitation of both companies' core strengths. As part of the in-database initiative, the foundation of the SAS and Teradata partnership enables end-users to run and optimize key aspects of SAS analytic processes within the Teradata database (Webb 2008).

Selective Base SAS and statistical procedures were made available for in-database processing in the first maintenance release of SAS 9.2. In the second and third maintenance releases, the following Base SAS, SAS Enterprise Miner™, SAS/ETS_®, and SAS/STAT_®

procedures are enhanced for in-database processing. Additional procedures are planned for incremental delivery in subsequent releases (Webb 2008). The procedures marked with an asterisk (*) require SAS Analytics Accelerator to run inside the database (SAS 2011a).

CORR*
CANCORR*
DMDB*
DMINE*
DMREG*
FACTOR*
FACTOR*
FREQ
FREQ<

- SORT • SUMMARY/MEANS
- TIMESERIES*
 - TABULATE
 - VARCLUS*

Predictive and descriptive model training typically requires many passes over the prepared, de-normalized data. SAS analytic and statistical functions, as well as SAS procedures, that execute repeatedly in model training can be moved inside the database. SAS Enterprise Miner procedures that are candidates for inclusion are DMDB (Data Mining Data Base), DMREG (Regression), NEURAL (Neural Network), DMVQ (Clustering), and others. Possible SAS/STAT software procedures include LOGISTIC, CATMOD, GENMOD, GLM, and many more (SAS 2007).

ANALYTICS ACCELERATOR

When performing in-database modeling, the SAS Analytics Accelerator for Teradata dynamically generates SQL, which is based on the procedure options and statements. It then submits the SQL code directly to the database. The code can be standard SQL that can be interpreted by any database, or it can be tuned specifically for Teradata. The choice for the type of SQL code is determined by the complexity of the required analysis (SAS 2011b).

FORMAT PUBLISHING AGENT

The process for exporting custom format definitions is called SAS Format Publishing. The SAS FORMAT procedure enables the creation of custom formats that replace raw data values with formatted character values. PROC FORMAT accomplishes this by creating a simple lookup table. SAS In-Database technology contains a tool for exporting these lookup tables from the SAS System to the Teradata system. This tool is delivered as part of the SAS/ACCESS Interface to Teradata and is known as the SAS Format Publishing Agent for Teradata (Webb 2008).

TABLE SERVER PROGRAMMING LANGUAGE

A further goal of SAS In-Database processing is to encapsulate the logic inside the DATA step implicit loop in a context that can be executed inside Teradata. Relative to the DATA step, Table Server Programming Language (TSPL) is a lightweight implementation that can be executed outside a SAS process. The SAS 9.2 releases of SAS In-Database technology include the tools needed to translate a user's existing DATA step program into TSPL-based functions (Webb 2008).

SCORING ACCELERATOR

The SAS Scoring Accelerator for Teradata embeds the robustness of SAS Enterprise Miner scoring models directly in the highly scalable Teradata database. By using the SAS In-Database technology and the SAS Scoring Accelerator for Teradata, the scoring process is done inside the data warehouse, and therefore does not require the transfer of data (SAS 2011a).

CONCLUSION

The addition of Ordered Analytical Functions and the speed of MPP database platforms, like Teradata, have greatly improved the ability for analysts to obtain information quickly and efficiently. Using the right tool for the right job is at the heart of these improved capabilities and all manner of research can benefit from using this new technology. In addition to the new SQL functions, SAS has begun a robust in-database project in partnership with Teradata Corporation that will undoubtedly prove to be a huge enhancement to the processes that exist today.

REFERENCES

Ballinger, C. (2007): "Born To Be Parallel", Why Parallel Origins Give Teradata an Enduring Performance Edge.

Clark, S. (2000): "Butler Group Research Paper", Research Paper Teradata, Butler 100101, October 2000.

Dorfman, P. (2009): "The SAS® Hash Object in Action", Proceedings of the SAS Global Forum, 2009, 153-2009.

Heaton (2008): "Many-to-Many Merges in the DATA Step", Proceedings of the SAS Global Forum, 2008, 81-2008.

Mann, R. (2009): "What's a DBA to do?", Teradata Magazine Online, Retrieved February 15, 2011, <u>http://www.teradatamagazine.com/tech2techtemplate.aspx?id=11188</u>

SAS (2007): SAS Institute, "SAS/ACCESS® 9.2 for Relational Databases", Reference, Fourth Edition

SAS (2011a): SAS Institute, "<u>SAS® In-Database Processing</u>", A Roadmap for Deeper Technical Integration with Database Management Systems.

SAS (2011b): SAS Institute, "SAS Analytics Accelerator for Teradata", Chapter 1

Secosky, J. (2007): "<u>The DATA step in Version 9: What's New?</u>", Proceedings of the Twenty-seventh Annual SAS Users Group International Conference, 27, dsv9.

Sedgewick R., Algorithms, Addison-Wesley, 1983, <u>ISBN 0-201-06672-6</u>, page 199, chapter 15: Balanced Trees. Adelson-Velskii, G.; E. M. Landis (1962).

Siemers, V. (2000): "<u>Removing Duplicates: Proc Sql Can Help You "See</u>"", Proceedings of the Twenty-fifth Annual SAS Users Group International Conference, 25, 106-25.

Teradata (2010a): Teradata Corporation, "SQL Reference: Functions and Operators", Release 12, B035-1145-067A.

Teradata (2010b): Teradata Corporation, "SQL Fundamentals: Teradata Database", Release 13, B035-1141-098A.

Webb, B. (2008): "SAS® In-Database Processing with Teradata", An Overview of Foundation Technology.

Weiming H., (2004): "<u>Top Ten Reasons to Use PROC SQL</u>", Proceedings of the Twenty-ninth Annual SAS Users Group International Conference, 29, 042-29.

Wikimedia Foundation Inc. 2005: Wikimedia Foundation, "Structured Query Language", Retrieved February 15, 2011, <u>http://en.wikipedia.org/wiki/SQL</u>.

Wikimedia Foundation Inc. 2005: Wikimedia Foundation, "Hash Table", Retrieved February 15, 2011, <u>http://en.wikipedia.org/wiki/Hash_table</u>.

ACKNOWLEDGMENTS

The authors would like to thank the following for their assistance: Kim Andrews, Rich Coyle, Yadira Sanchez, Greg Savord, and Julie Slater from CMS; the SAS Help Desk; and James Bourgeois, Kishore Elagandhala, Robert Jacob, Suren Kalathil, Mohit Kulkarni, and Thomas Warren from Teradata Corporation.

CONTACT INFORMATION

Rick Andrews

Office of the Actuary Centers for Medicare and Medicaid Services 7500 Security Boulevard Baltimore, MD 21244 Phone: (410) 786-4088 E-mail: <u>Richard.Andrews2@cms.hhs.gov</u> Jignesh Miyani Senior Architect Teradata Corporation 7175 Security Boulevard Baltimore, MD 21244 Phone: (301) 820-4521 E-mail: Jignesh.Miyani@Teradata.com

Teradata, IBM, and SAS and all other product or service names are registered trademarks or trademarks of their respective companies in the USA and other countries. ® indicates USA registration.