Paper 1797-2014

A Paradigm Shift: Complex Data Manipulations with DS2 and In-Memory Data Structures

Shaun Kaufmann, Farm Credit Canada.

ABSTRACT

Complex data manipulations can be resource intensive, both in terms of development time and processing duration. However in recent years SAS® has introduced a number of new technologies that when used together can produce a dramatic increase in performance while simultaneously simplifying program development and maintenance. This paper presents a development paradigm that utilizes the problem decomposition capabilities of DS2, the flexibility of SQL and the performance benefits of in-memory processing using hash objects.

INTRODUCTION

As SAS programmers, performing data manipulations is fundamental to almost every assignment we receive. Distilled to its essence, most tasks can be thought of as simply extracting data from a source, transforming its state in some manner and storing the results. However, while fundamentally similar in principle, not every data manipulation task is equally challenging. For simple tasks there are often many acceptable approaches for solving the problem. Some tasks however exhibit a level of complexity that requires more from the programmer. In order to generate a result in an acceptable timeframe, a detailed and complete understanding of both the problem and the available toolset is required. It is these complex data manipulations that are the focus of this paper.

When addressing complex data manipulation tasks, the underlying theme should be one of efficiency. How can we reduce the time required for a query to extract data from our source tables? How can we reduce development time by quickly and accurately writing our code? How can we reduce the execution duration of our programs? An effective approach for tackling complex data manipulation tasks must address all three of these questions.

The remainder of this paper is organized as follows. First, we clarify the properties of a complex data manipulation. We then discuss how your specific computing architecture can influence your decision when selecting the best SAS procedure for tasks of this nature. Next we discuss the benefits of developing a standard approach for tackling these types of problems and suggest a process that employs the technique of top-down design. It is here that you will be introduced to the new SAS DS2 language. With its user-definable methods, DS2 lends itself well to the decomposition inherent in a top-down design approach to algorithm development. We then introduce the concept of in-memory techniques. In-memory processing is a hot topic these days so you may be surprised to learn how easily you can integrate the benefits of in-memory processing into your development process, providing the speed of execution necessary for tackling the most challenging data manipulation tasks. The paper concludes by presenting a comprehensive example to reinforce the concepts previously introduced. This walkthrough, while fictitious and simplified for ease of understanding, concretely demonstrates the proposed approach, thus transforming the theoretical to practical.

WHAT MAKES A DATA MANIPULATION COMPLEX?

What do we mean when we refer to a data manipulation as being complex? For the purposes of this paper we will consider a data manipulation task to be complex if it has at least one of the following properties:

- The problem itself is inherently complex (i.e. the logic required to solve the problem is difficult to understand and implement).
- The resulting algorithm is computational complex (i.e. the runtime does not scale well as the input size increases, resulting in unacceptable performance).

I find it helpful to consider each of these properties as a cost function, where the cost is measured in terms of time. Problems that are complex to code carry a higher cost in terms of development and maintenance time. In this case we seek to decrease the workload on the programmer. Alternatively, problems that are computationally complex carry a higher cost in terms of execution time. In this case we seek to decrease the duration of program runtime. Ideally, our development approach should address both types of costs and minimize them to the greatest extent possible.

SAS PROCEDURES AND YOUR COMPUTING ENVIRONMENT

Before beginning to address any specific complex data manipulation it is important to be mindful of two external factors:

- What is the architectural design of your computing environment?
- What are the performance characteristics of each specific SAS technology?

While these two factors are somewhat peripheral to the specific data manipulation task, they nonetheless can have a major impact on both development and execution duration. The interplay between a computing environment and a specific SAS procedure can play a significant role in determining the optimal approach for your complex data manipulation tasks. In particular, source data location and structure has a significant impact on the efficiency of the data extraction phase of a complex data manipulation. Whether a DATA step, a PROC SQL or a PROC FEDSQL is most appropriate for a data extraction exercise will be heavily influenced by your source data's location and structure. It is prudent to invest the time to determine the most efficient strategy for your infrastructure.

In my particular case, source data is stored almost exclusively in an Oracle-based dimensional data warehouse. In this situation using a PROC SQL procedure with Explicit SQL Pass-Through yields excellent execution performance for data extraction tasks that require joining and subsetting. This is due to the fact that the enterprise data warehouse is implemented on a powerful server configured (e.g. indexed, parallelized, etc.) to be extremely efficient for joining and subsetting operations. Using the explicit SQL pass-through version of a PROC SQL procedure allows me to fully leverage this power.

At this point is also pertinent to mention the introduction of the DS2 language, which becomes a full production technology as of the SAS 9.4 release. The DS2 Language Reference provides the following introduction:

"DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with Base SAS and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods, and packages. Several DS2 language elements accept embedded FedSQL syntax and the runtime-generated queries can exchange data interactively between DS2 and any supported database. This allows SQL preprocessing of input tables which effectively combines the power of the two languages."

While this introduction can be a little overwhelming, it is important to be aware that you can use as much or as little of DS2's features as you need. For the purposes of this paper we will focus on the newly provided ability to write user-defined methods. For further information on the DS2 language please consult the SAS 9.4 DS2 Language Reference.

COMPLEX DATA MANIPULATIONS: A STANDARD APPROACH

While it is unlikely that any two programming tasks will be identical, it is useful to develop a general method for tackling complex data manipulations. Having a standard approach to follow significantly decreases development time by reducing churn in the face of a new challenge. Additionally, a standard approach shared by all team members is of great benefit when the time comes to maintain each other's code. Base SAS provides its programmers with a diverse toolset and because personal preferences differ greatly, it increases the difficulty in supporting other developers' code.

As the complexity of a programming task increases, it becomes beneficial to segregate the logical design (the algorithm) from the actual program implementation (the coding).

An algorithm can be thought of as a recipe for solving a problem. This recipe lays out the series of steps that will be followed in order to accomplish the computational task. While many programmers simply start coding based on their experience and intuition, it is almost always better to design the algorithm first in natural language. This approach simplifies the construction of the logic of the process without initially worrying about the implementation details.

ALGORITHM DESIGN: A TOP-DOWN APPROACH

Algorithm design is central to the field of computer science and as such has been rigorously studied. This is of great benefit to us as we can simply leverage the learnings of others in developing our approach for tackling complex data manipulations. Specifically I find it beneficial to apply a top-down approach whereby a complex problem is broken down into smaller, more easily solvable, parts (sometimes referred to as decomposition).

Documenting this initial description of the program logic can be accomplished using what is referred to as pseudocode, which consists of short natural language (e.g. English) phrases used to explain the specific tasks within a program's algorithm. Consider the following business requirement:

Given a table of monthly sales transactions, compute the appropriate sales commission for each sales transaction where the sales commission is dependent on the month-to-date dollar value of sales sold by the individual salesperson on the transaction.

Assuming that the input table is already sorted in chronological order, the pseudocode representation of the solution could be expressed as in Figure 1.

```
For each transaction
       Get the current month-to-date sales for the associated salesperson
       Determine the appropriate commission factor
       Calculate the commission for this transaction
       Update the salesperson's month-to-date sales
```

Figure 1. Pseudocode

Once the required logic has been captured in pseudocode, the next step is to translate that pseudocode into a skeleton program. This skeleton program resembles pseudocode but must be capable of running without generating errors. It can be thought of as an intermediate step between the pseudocode design and the completed implementation. It is here that we can utilize the ability to write user-definable methods inherent in SAS DS2.

First we can directly migrate the pseudocode into the run method of the DS2 program as seen in Figure 2.

```
proc ds2;
       Data _null_;
              method run();
                     set mydata.monthly_sales_transactions;

                     getMonth_To_Date_Sales();
                     getCommission_Factor();
                     Calculate_Commission();
                     updateMonth_To_Date_Sales();
              end;
       enddata;
run;
quit;
```

Figure 2. DS2 Skeleton with Method Invocations

Next, in order for this code to execute without error, we must add a method body for each method that we have invoked above. Here we can simply leave the method definition empty. However, it is important that each method is defined before it is invoked as demonstrated in Figure 3.

```
proc ds2;
       Data _null_;

              Method getMonth_To_Date_Sales();
                     /* Implement code here.  */
              End;

              Method getCommission_Factor();
                     /* Implement code here.  */
              End;

              Method Calculate_Commission();
                     /* Implement code here.  */
              End;

              Method updateMonth_To_Date_Sales();
                     /* Implement code here.  */
              End;

              method run();
                     set mydata.monthly_sales_transactions;

                     getMonth_To_Date_Sales();
                     getCommission_Factor();
                     Calculate_Commission();
                     updateMonth_To_Date_Sales();
              end;
       enddata;
run;
quit;
```

Figure 3. Skeleton Code with Method Definitions Added

Having completed the skeleton program, two points should have become clear:

- First, the code very closely resembles the pseudocode representation. This suggests that moving from pseudocode to skeleton code is a relatively easy transition.
- Second, the run method of the program effectively represents the logic of the program. Therefore the basic algorithm can be quickly understood.

IMPLEMENTATION

Having laid out the high level logic for our algorithm, we can set about implementing each DS2 method. Each data manipulation task has its particular challenge. For the example above, the challenge is how to maintain a running total of the sales for each salesperson and there exist a variety of ways this could be accomplished. Some possible options include:

- Pre-sorting the dataset by salesperson and transaction date, then employing the FIRST.salesperson and LAST.salesperson BY variables provided as part of BY-group processing to determine when to reset the running total.
- Utilize DS2's capablity for dynamically creating and executing SQL queries from within the implicit loop of the DS2 run method. This approach would allow us to maintain running totals within an ordinary table through a combination of update and select queries.
- Utilize a SAS Hash Object to maintain the running total.

While each of these options could yield a suitable solution, this is an excellent opportunity to implement an in-memory approach by utilizing the power and ease of use of a SAS Hash Object.

IN-MEMORY TECHNIQUES FOR PERFORMANCE IMPROVEMENT

As datasets grow larger and larger there is an increasing need for a technique to process them efficiently. A major theme in this regard is a shift toward in-memory techniques with the rationale that RAM is faster than disk. However, it is important to note that not all in-memory approaches provide equal performance benefits. While it is true that a sequential search of a recordset in RAM is faster than a sequential search of a recordset on disk, some in-memory data structures can provide an even greater performance increase. For example, a hash object can retrieve/store a value in constant time regardless of the size of the dataset. This characteristic makes hash tables an excellent approach for in-memory processing. Since the introduction of the hash object in SAS 9, this feature has been steadily evolving to address the increasing need for in-memory techniques for performance improvement. As of SAS 9.4, the hash object is a mature SAS component and an extremely efficient approach for in-memory processing.

PUTTING IT ALL TOGETHER: AN END TO END EXAMPLE

To illustrate how the aforementioned techniques can be brought together to create a standard approach for solving complex data manipulation tasks, we will expand on the example introduced earlier. Figure 4 presents the expanded business requirements.

Your company (XYZ Inc.) is a large retailer producing 100 million retail transactions per month with a sales force of approximately 10,000 employees. The company is considering adopting a commission structure where sales commissions are based on both the tiered month-to-date sales of the salesperson and the store category (warehouse or retail store). As part of the impact analysis you are tasked with computing the hypothetical sales commission for each transaction for the December 2013 time period.

Figure 4. Expanded Business Requirements

Additional details include:

- Historical sales transaction data is stored in a non-SAS based (Oracle) dimensional data warehouse. Specifically, store category data is stored in a separate table and therefore a join or merge step is required.
- The sales transaction table contains an entire year of data (1.2 billion rows), so subsetting is required.
- The sales force is large and the exact number varies over time (approximately 10,000).
- The commission structure is:
  - 1% up to $25,000 of sales for warehouse staff, 1.5% on any additional sales.
  - 2% up to $10,000 of sales for retail staff, 3% on any additional sales.

It is important to note that the business requirements presented in Figure 4 in no way invalidate the pseudocode and skeleton program prepared previously. However, the additional details do indicate the magnitude of the processing task (100 million transactions to process, with a large and dynamic sales force) as well as the environment in which the task will be performed.

As an initial step, it is useful to explicitly state the three main challenges inherent in this process:

1. How can we most efficiently extract the source data into SAS? ←(hint: SQL can help)
2. What is an appropriate algorithm for solving this problem? ←(hint: DS2 can help)
3. How will I efficiently implement my solution? ←(hint: Hashes can help)

We will now address the issues presented for each individual challenge as we work towards the solution to the larger problem as a whole.

CHALLENGE ONE: EFFICIENT EXTRACTION.

Having an enterprise data warehouse (EDW) that resides outside of SAS is not uncommon. While this can introduce additional challenges if not utilized properly, it also presents an opportunity to leverage the power of the underlying infrastructure. As mentioned earlier, EDWs are usually implemented on powerful servers and it is in our best interest to make the most of the opportunity this provides us with. Here, the infrastructure can be best leveraged by utilizing the Explicit SQL Pass-Through functionality of the PROC SQL procedure. The complete PROC SQL query can be seen in Figure 5.

```
proc sql;

    CONNECT to Oracle (path=EDWP1 user=KAUFMANN password=PASSWORD);

    CREATE TABLE SASUSER.SGF2014_Sales_Transactions_100M AS SELECT * FROM connection to Oracle
   (Select A.Transaction_ID, A.Transaction_Date, A.Transaction_Amount, A.Salesperson_ID,
        B.Store Category
    FROM Sales_Transactions_2013 A, Store_Sales_Classification B
    WHERE A.Store_ID = B.Store_ID
    AND A.Transaction_Date
        BETWEEN TO_DATE ('2013/12/01', 'yyyy/mm/dd') AND TO_DATE ('2013/12/31', 'yyyy/mm/dd' )
    ORDER BY A.Transaction_Date ASC);

    DISCONNECT FROM Oracle;

quit;
```

Figure 5. PROC SQL with Explicit Pass-Through.

It is important to appreciate the hidden subtleties of the enterprise data warehouse and the benefits that they provide to us. Specifically, both tables are already indexed on the join keys, greatly increasing the speed of the join operation. Additionally, the Transaction_Date column is also indexed, resulting in more efficient subsetting.

This single PROC SQL block performs many functions simultaneously, including:

- Performs a join between the Sales_Transactions_2013 and the Store_Sales_Classification tables.
- Subsets the resulting dataset for the appropriate sales transaction date range so only the required records are transferred from ORACLE to SAS.
- Utilizes ORACLE specific SQL functions (BETWEEN and TO_DATE)
- Orders the resulting dataset in ascending order based on the transaction date timestamp

All of the above operations are performed using the powerful infrastructure of the enterprise data warehouse with the resulting table residing in the SASUSER library.

CHALLENGE TWO: ALGORITHM DESIGN.

The good news is that this challenge has already been met. As mentioned earlier, the pseudocode and skeleton program presented in Figures 1, 2 and 3 remain unchanged. It is a testament to this approach that the logic required to accurately solve the problem can be isolated in this manner. The logical approach (algorithm) is unaffected by details such as source location, transactional volume, etc. All that remains is to fill in the implementation details for each method definition.

CHALLENGE THREE: DETAILED IMPLEMENTATION.

Reiterating the business requirements, this assignment requires the programmer to process 100 million sales transactions. Left unsaid in the problem specification is that we would like to process these 100 million records in the least amount of time possible. From an implementation standpoint, the main challenge centers around how to efficiently maintain the running totals of month-to-date sales for approximately 10,000 employees. It has been my experience that for situations of this nature, hash objects provide the efficiency required while simultaneously

simplifying the implementation process. While there are other options, the following implementation will center around a SAS Hash Object based in-memory approach.

Readers inexperienced with SAS Hash Objects will be pleasantly surprised by their ease of use. The SAS Hash Object interacts with the program data vector (PDV), so the only step necessary after declaration and instantiation is to associate the desired PDV variables with the hash's key and data elements. Having made this association, calling the hash's find() method will use the key value from the PDV, retrieve the associated data elements for that entry in the hash object and update the PDV with their values. The add() and replace() methods operate similarly. For a more complete introduction to SAS Hash Objects, please consult the SAS 9.4 Language Reference.

The first step in using a SAS Hash Object is declaration and instantiation. The code presented in Figure 6 creates a hash object called h_Month_To_Date_Sales.

```
declare package hash h_Month_To_Date_Sales();
```

Figure 6. Hash Declaration and Instantiation

Next, we must define the hash key and data variables using the defineKey() and defineData() methods. This code must be executed only once, so the definitions are added within the DS2 init() method (Figure 7). For readers unfamiliar with DS2, the init() method is akin to using "`if _n_=1 then`" in a DATA step. Calling the defineDone() method completes the hash definition. We are also required to add a declaration for the month_to_date_sales variable. This is necessary since this variable is not part of the input dataset and must be created in the PDV before being used in the hash definition.

```
declare float month_to_date_sales;

method init();
      h_Month_To_Date_Sales.defineKey('salesperson_id');
      h_Month_To_Date_Sales.defineData('salesperson_id');
      h_Month_To_Date_Sales.defineData('month_to_date_sales');
      h_Month_To_Date_Sales.defineDone();
end;
```

Figure 7. Hash Definition

Additionally, we need to declare several working variables (Figure 8); commission_factor and commission_amount variables are required to hold the appropriate values for each transaction, as is a variable to catch the return codes (rc) for various hash operations.

```
declare float commission_factor;
declare float commission_amount;
declare integer rc;
```

Figure 8. Variable Declarations

With the necessary declarations out of the way, we can move on to the task of implementing our four user-defined methods. Figure 9 presents the code for the getMonth_To_Date_Sales() method. First the find() method of the h_Month_To_Date_Sales hash is called. This takes the current value of the salesperson_id variable from the PDV and checks to see if it exists as a key in the hash. If it does exist, then the return code is set to zero and the month_to_date_sales value associated with this salesperson_id is loaded into the PDV. If an entry for this salesperson_id is not found in the hash, then a non-zero return code is generated. In this case we then add an entry to the hash for this salesperson_id, having the value of zero for month_to_date_sales.

```
method getMonth_To_Date_Sales();
        rc = h_Month_To_Date_Sales.find();
        if rc <> 0 then
                do;
                        month_to_date_sales = 0;
                        h_Month_To_Date_Sales.add();
                end;
end;
```

Figure 9. The getMonth_To_Date_Sales() Method

Figures 10 and 11 present the implementation of the getCommission_Factor() and Calculate_Commission() methods. While there is nothing particularly interesting about the code itself, it is important to note the value provided by encapsulating this logic within separate methods. In more complicated real world examples, segregating logical units of code lends itself to re-use. Additionally program maintenance is simplified and code changes are likely to introduce fewer errors.

```
method getCommission Factor();
        if Store_Category = 'retail' then
                do;
                        if month_to_date_sales < 10000 then commission_factor = 0.02;
                        else commission_factor = 0.03;
                end;

        if Store_Category = 'warehouse' then
                do;
                        if month_to_date_sales < 25000 then commission_factor = 0.01;
                        else commission_factor = 0.015;
                end;
end;
```

Figure 10. The getCommission_Factor Method

```
method Calculate_Commission();
        commission_amount = Transaction_Amount * commission_factor;
end;
```

Figure 11. Calculate_Commission Method

The updateMonth_To_Date_Sales() method takes the current value of the month_to_date_sales variable (obtained previously by the getMonth_To_Date_Sales() method and adds to it the amount of this particular sales transaction. Then the hash's replace() method is called to update the value within the hash (Figure 12).

.

```
method updateMonth_To_Date_Sales();
        month_to_date_sales = month_to_date_sales + Transaction_Amount;
        h_Month_To_Date_Sales.replace();
end;
```

Figure 12. updateMonth_To_Date_Sales Method

Figure 13 presents the complete code. In regards to performance, this implementation results in a runtime of approximately 3 minutes for 100 million rows when executed on a Late 2011 MacBook Pro running Windows 7 in a virtual machine.

Additionally, notice that the run() method quickly conveys the high level algorithm. As well, the code for each specific step is encapsulated in a separate method, making its logic obvious. These factors lend themselves to ease of maintenance, especially within a team of programmers where the person performing the maintenance isn't the original author.

```
proc ds2;

data Commissions(overwrite=yes);

        keep Transaction_ID commission_amount;

        declare package hash h_Month_To_Date_Sales();

        declare float month to date sales;
        declare float commission_factor;
        declare float commission_amount;
        declare integer rc;

        method init();
                h_Month_To_Date_Sales.defineKey('salesperson_id');
                h_Month_To_Date_Sales.defineData('salesperson_id');
                h_Month_To_Date_Sales.defineData('month_to_date_sales');
                h_Month_To_Date_Sales.defineDone();
        end;

        method getMonth_To_Date_Sales();
                rc = h Month To_Date_Sales.find();
                if rc <> 0 then
                        do;
                                month_to_date_sales = 0;
                                h_Month_To_Date_Sales.add();
                        end;
        end;

        method getCommission Factor();
                if Store_Category = 'retail' then
                        do;
                                if month_to_date_sales < 10000 then commission_factor = 0.02;
                                else commission_factor = 0.03;
                        end;

                if Store_Category = 'warehouse' then
                        do;
                                if month_to_date_sales < 25000 then commission_factor = 0.01;
                                else commission_factor = 0.015;
                        end;
        end;

        method Calculate_Commission();
                commission_amount = Transaction_Amount * commission_factor;
        end;

        method updateMonth_To_Date_Sales();
                month_to_date_sales = month_to_date_sales + Transaction_Amount;
                h_Month_To_Date_Sales.replace();
        end;

        method run();
                set SASUSER.SGF2014_Sales_Transactions_100M;
                getMonth_To_Date_Sales();
                getCommission_Factor();
                Calculate_Commission();
                updateMonth_To_Date_Sales();
        end;

enddata;
run;
quit;
```

Figure 13. Complete DS2 Implementation

CONCLUSION

Complex data manipulations can be resource intensive, both in terms of development time and processing duration. In these instances, choosing the appropriate SAS technology is essential to producing a viable solution. Making appropriate choices allow you to get the most out of your resources by using them to their fullest potential. Additionally, taking the time to formalize the development approach within your team will ensure that each team member can achieve the same high level of results.

Your approach should seek to minimize duration in all areas: data extraction, program development and program execution. To this end we have suggested an approach for each: SQL for efficient extractions, DS2 for efficient program development, and SAS Hash Objects for efficient program execution.

ACKNOWLEDGMENTS

I'd like to thank Christine Giambattista for her efforts in editing this paper.

REFERENCES AND RECOMMENDED READING

- *SAS® 9.4 DS2 Language Reference*
- *SAS® 9.4 Language Reference*
- *SAS® 9.4 Components Objects Reference*
- *SAS® 9.4 SQL Procedure User's Guide*
- *SAS® 9.4 FedSQL Language Reference*

CONTACT INFORMATION

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

Shaun Kaufmann
Farm Credit Canada
1800 Hamilton Street
Regina, SK, Canada S4P 4L3

(306) 550-9104:
Shaun.Kaufmann@gmail.com