TECHNICAL DISCUSSION OF THE IMPLEMENTATION APPROACH/METHODS USED FOR DEVELOPING THE SUMMARY TABLE V2.0 PACKAGE

Version 0.3
June 24, 2016

Sentinel is sponsored by the U.S. Food and Drug Administration (FDA) to monitor the safety of FDA-regulated medical products. Sentinel is one piece of the Sentinel Initiative, a multi-faceted effort by the FDA to develop a national electronic system that complements previously existing methods of safety surveillance. Sentinel Collaborators include Data and Academic Partners that provide access to health care data and ongoing scientific, technical, methodological, and organizational expertise. The Sentinel Coordinating Center is funded by the FDA through the Department of Health and Human Services (HHS) Contract number HHSF223201400030I.

Table of Contents

Document History

The following table is a revision history for this document.

Date	Version	Comments
December 14, 2015	0.1	First draft version
December 15, 2015	0.2	Incorporate revisions from Robert Rosofsky
June 24, 2016	0.3	Add content, clean up language

I. INTRODUCTION

The purpose of this document is to shed light on the various technical approaches and methods that guided the design and development efforts for the v2.0 Summary Tables Package.

II. DESIGN GOALS

1. The code should be well modularized to support routine development, testing/debugging, and maintenance/enhancement activities, as well as to better support routine operational contingencies (e.g. allow selective module execution for restarts).
2. The code should execute as efficiently as possible in order to minimize long runtimes.

III. GOAL – IMPROVE MODULARIZATION OF CODE

A. OVERVIEW OF PACKAGE STRUCTURE AND EXECUTION FLOW

In the v2.0 package, the programs fall into four general types

1. **Main program** – Program contains user parms, is executed by user, and includes the "true" Master program *(located in./sasprograms/ subfolder)*
2. **Master program** – Program contains developer parms, and sequentially includes and executes the Module Code *(located in./inputfiles/ subfolder)*
3. **Module Code** – Individual program modules that collectively orchestrate the extraction and transformation of MSDD data into output summary tables with the support of utility macros to carry out many of the discrete tasks *(located in ./inputfiles/module_code/ subfolder)*
4. **Utility Code** – Individual macro programs (or programs with collections of "small" macros), that are included once by the Utility_Code module, and are then executed as necessary by modular programs or by other utility macros *(located in ./inputfiles/utility_ code/ subfolder)*

B. MAIN PROGRAM

The main program – **qt_main_program.sas** -- asks users to specify parameter values, retrieves ETL specific and DP specific metadata via Common Components, sets up all of the standard paths and SAS librefs used by the package for data and programs, and includes at the very end, the master program.

Parameters specified in the main program:

- MSCC – Specifies location of Common Components include program
- DPTemp – Specifies location of DPTemp library for temporary files that persist across modules
- ExportPath – Specifies location of path for exporting summary tables as text files and optionally as a MS Access database
- AccessFiles – Specifies whether to produce a MS Access database and a code that maps to the extension for the file (i.e., MDB or ACCDB)
- ACCESS_DBMS – Specifies a platform specific keyword required when exporting to a MS Access database

C. MASTER PROGRAM

There are two versions of the master program -- a "DEV" version (**900-dev_master.sas**) and a "PROD" version (**001-prod_master.sas**). The "DEV" version is intended for customization for development/QA/ ad-hoc runs while the "PROD" version is intended for production runs.

The master program specifies parameters that are useful for development and QA activities, includes a module that performs startup chores (000-startup.sas) in preparation for running the summary tables modules, includes and runs each summary table module in sequence, and at the end, includes a module that performs shutdown chores (999-shutdown.sas).

Parameters specified in this program for developer/QA programmer:

- DEBUG – Specifies whether to create additional QA related variables, messages to the LOG, and output datasets.
- CleanWork – Specifies whether to purge the WORK library at the start of each new module
- CleanDPTemp – Specifies whether to delete DPTemp library files as soon as they are no longer needed
- ModuleLogs – Specifies whether to create separate log files for each module
- TextFiles – Specifies whether to export summary tables as text files

D. MODULE CODE

There are thirty-one modules in the package which may be divided into two categories:

1. Summary table creation
2. Infrastructure support

The twenty-seven summary table modules are directly involved in the process of extracting and transforming MSDD data into summary tables while the four infrastructure support modules provide operational support. For more documentation, please see the Summary Tables specifications document (Sentinel - Summary Table V2 Programming Specifications v1.0.docx).

1. Infrastructure support modules

There are three modules that provide startup support prior to running any of the business related modules:

1. 000-startup.sas – This module sets up SAS options, defines global parameters, and includes the modules Utility_Code module and Validate_Parms.

2. Utility_Code.sas – This module includes all of the supporting utility macros, formats, and user-functions

3. Validate_Parms.sas – This module validates parameter values for parms defined in the main and master programs.

There is also a module that provides shutdown support that executes after all of the modules have completed:

1. 999-shutdown.sas – Calls macro to create overall signature file and return log control to main log

E. UTILITY CODE

There are twelve utility programs that, like the module code, come in two categories:

1. Business algorithm support
2. Infrastructure support

Utility macros that provide business algorithm support manipulate MSDD data according to a set of business rules. An example is the macro that bridges enrollment spans (bridge_enr_spans.sas).

In contrast, infrastructure support utility programs provide general services that have potentially broader use. An example is the macro that controls setup and cleanup chores related to including modules (inc_next.sas)

2. Inc_next

The macro inc_next, which is used by the master program to include each summary table module, is worth highlighting; it orchestrates the following chores for each module run:

- Log control
- Error control
- Cleanup of the WORK library
- Cleanup of the DPTemp library
- Execution of the specified module itself
- Update MSOC.ModuleRuntimes dataset with module level runtime statistics

IV. GOAL – IMPROVE RUNTIME EFFICIENCY

Techniques are chosen based on their ability to influence at least one of the following:

1. Reduce I/O processing
2. Increase efficiency of memory usage
3. Increase efficiency of multi-threaded processing
4. Reduce CPU processing (of the non-threaded kind)
5. Reduce disk storage size requirements

A. USE OF DATA-REDUCTION TECHNIQUES

- Reduces the number of page-reads from disk and page-writes to disk by increasing the number of observations that fit into a page of memory, thus reducing I/O.
- Improves the odds of being able to read data from memory rather than disk, reducing I/O. This can occur because reducing file sizes improves file-caching system efficiency (e.g. Files > 2G are often not cached or not very efficiently cached[i]).

Methods include:

1. Swap out large-byte variables and replace with small-byte variables, reducing I/O:
   A. Create small-byte numeric sequential-key variables to replace large-byte character MSDD variables:
      - PatID is replaced with PatKey
      - DrugClass is replaced with DrugClassKey
      - GenericName is replaced with GenericNameKey

      Patient ids are often stored as 16-byte or even 35-byte character fields which can be reduced down to a 4-byte or 5-byte numeric field. Similarly, drug class names and generic names are stored as 35-byte and 70-byte character description fields which can be reduced down to 3-byte or 4-byte numeric fields.

   B. Create small-byte numeric decimal variables to replace large-byte binary character string variables:
      - Enrollment coverage by type (e.g. DrugCov and MedCov) and calendar-year is in some places represented by a binary character string variable in order to, in a single variable, identify multiple years of coverage. But rather than load the large binary string into hash tables, the much smaller decimal equivalent is loaded instead. In one module, the decimal value is created and in another value it is converted back to a binary string value to enable checking coverage types by year.
        For example, a data partner with 15 years of enrollment data would need a binary character string of 15-bytes to store patient level coverage information by calendar-year. The decimal equivalent needs only 4-bytes of storage for each patient.
        A maximum of 64 years of calendar-year enrollment coverage data can be converted from binary to decimal format and vice versa.

2. Use statements to limit data to only the variables and records needed, reducing I/O.

   Testing shows that the DATA-step extraction processing modules run faster using subsetting-IF statements rather than subsetting-WHERE statements. This is because the majority of observations are kept in extraction and DATA-step WHERE-clause processing incurs a small incremental cost per record evaluated to see if it should be read into the PDV.

3. Join aggregated strata to long descriptions as late as possible

   All of the diagnosis, procedure, and drug summary tables have long text descriptions as part of their structure. These variables vary from 30 to 70 characters in length. The event data are processed and aggregated through use of only codes and/or key values first. Then these aggregations are linked to lookup tables to obtain the event descriptions as late in the process as possible. This greatly reduces I/O processing that would occur had the descriptions been carried in the files earlier in the processing.

B. USE OF HORIZONTAL DATA SPLITTING AND LOOPING TECHNIQUES

Horizontal table splitting, which creates "bite-sized" files, in conjunction with looping through the splits sequentially:

- Increases the odds of file caching, reducing I/O
- Increases the odds that a file that is loaded into memory during sorting/aggregating steps, which reduces I/O; it also increases the efficiency of multi-threaded sorts (i.e. when data are processed in memory, the CPUs are not kept waiting for I/O, resulting in real-time less than CPU time).
- Increases the odds of avoiding sorts altogether, reducing I/O (e.g. SQL may choose a hash-join if data are not usefully sorted/indexed and the smallest table fits into memory)
- Reduces the amount of disk and memory resources needed at any given moment, reducing the odds of job/system crashes.

Two types of horizontal table splits and loop processing are employed:

A. With measures of prevalence, we do not need to look across multiple calendar-years at any given time; we need to only consider the data within a single year at a given time. So splitting tables by calendar-year suits the needs of prevalent summary table processing.

B. With measures of incidence, we do need to look across multiple years of data at any given time; incidence-event look-back periods and follow-up periods are relative to each patient and frequently cross calendar-years. So splitting tables by patient partition suits the needs for incidence summary table processing.
The ideal number of patient-partitions needs to be dynamically calculated for each Data Partner's MSDD, such that the size of each split is small enough to be easily cached by the operating system (e.g. < 2G) and easily sorted in memory by threaded-sort procedures (which is dependent upon system configurations).
Note that patient-partitioning is helpful not only for patient-oriented processing tasks; it is also very helpful whenever the processing is particularly intensive and/or complex. For example, when faced with complex business algorithms involving many logical steps, a desirable work-flow to handle the complexity is to break up the programming logic into several discrete SAS DATA/PROC steps, each of which are easier to code, and since each of the steps process "bite-sized" files that are easily cached, performance holds up comparatively well.

C. USE OF VIEWS FOR TWO-STEP EXTRACTION PROCESSING

In the scenario of two-step extraction processing of large tables read sequentially, if the first step outputs results to a view instead of to a static dataset, the total number of I/O cycles may be reduced. The total number of data-pages read from disk and written to disk may be reduced because instead of the first step writing out the results back out to disk, the first step hands those data-pages while still in memory over to the second step to process. In other words, we are in effect reading data-pages from the large table from disk just once. (Note: if the table read in the first step is small enough to be efficiently cached, using a view may not yield any performance benefits).

Why not forgo the view entirely and put all extraction logic in a single step, which presumably would be even faster? The reason is that the two-step extraction process better supports the goals of modularization. The first step implements logic that is structurally common to producing all of the summary tables while the second step is customized depending on the type of summary tables it supports.

One example of this use of views are the *dx_etl.sas* and *px_etl.sas* modules which each create a view, followed by the respective *dx_code_files.sas* and *px_code_files.sas* modules, that utilize the views to write out a set of separate static datasets. Thus the MSDD tables are in actuality, read only once.

D. USE OF IN-MEMORY HASH-TABLE LOOKUPS TECHNIQUES FOR LINKING DATA BETWEEN TABLES

Unless the data are already usefully sorted by the linking keys, it is faster to load a smaller table into memory as a hash-table and to scan (i.e. sequentially read) the larger table for matches.

For 1:1 and M:1 relationship between data, a DATA-step hash-table provides the SAS programmer with the most control and possibly the best performance. However, for M:M relationship between data, PROC SQL hash-table joins are easier to program and possibly execute more efficiently[ii].

In some cases, it is practical and more efficient to flatten-out M:M relationship into M:1 relationship so that a DATA-step hash-table may be used. This technique is used in reviewing patient enrollment coverage by calendar-year information during extraction (note the TableExtract macro) as well as for assigning drug classes and generic names to NDC codes. We used a DATA-step hash-table in this case because the linkages are implemented during the initial data extraction steps of large MSDD tables, which involve complex control flow logic not easily done in SQL.

In other cases, a DATA-step hash table approach is not as practical. For example, for incident measures processing, we need to link in bridged enrollment spans to detailed event data, and this type of linkage involves a M:M relationship that is not very efficiently stored or processed in a flattened M:1 structure (e.g. the maximum number of bridged enrollment spans per patient can vary widely between Data Partners and a single patient with lots of span records can easily blow up the memory requirements of the hash-table to unreasonable sizes).

In the above case, we use a SQL hash-join to perform the linking. However, in order for SQL to select a hash-join, we need to first split the enrollment data and event data horizontally into patient-partitions, and then we need to loop through the patient partitions sequentially to perform the SQL joins.

E. USE OF EFFICIENT SORTING/AGGREGATING TECHNIQUES

While sorts may sometimes be avoided altogether when linking data between tables, it is often impractical to avoid sorts when aggregating data by stratification. For example, while PROC SUMMARY can aggregate data by stratifications without it being pre-sorted, the procedure is only efficient if the data fits into memory, which is impractical for large tables.

Ways to improve sorting/aggregation efficiency:

- Split tables horizontally prior to sorting/aggregating. (See section on horizontal table splitting for discussion of performance benefits).
- Prior to aggregation, diagnosis and procedure events are further split out into separate event datasets by specific code type/code level (e.g. icd9 3-digit, 4-digit, and 5-digit diagnosis codes, HCPC/CPT codes, icd9 3-digit, and 4-digit procedure codes). This speeds up the aggregation steps by excluding up front records that do not qualify for a particular aggregation.
- Arrange sort-keys/group-by keys in the order of most-number-of-levels to least-number-of-levels, reducing I/O and CPU time. For example, there are thousands of unique ICD9 diagnosis code values but only a handful of unique gender values. In this case, putting diagnosis code before gender in a GROUP BY SQL clause will result in faster sorts/aggregations. (Note the GROUP BY clause in the first PROC SQL step of the *prev_summ_finalize* macro as an example.)

- For the diagnosis and procedure measures, aggregation is required at both the individual care-setting level and across all care-settings levels. By flattening the care-setting stratification levels into Booleans and summarizing those Booleans, we eliminate the need to take a second sort/ aggregation pass at the event level data. From the all care-settings aggregation produced, we are able to split out separate aggregation rows for each of the individual care-setting levels. (Note the DATA step as the second part of the *prev_summ_finalize* macro as an example.)
- Leverage existing SORTEDBY data set attributes where possible. Data Partners' MSDD tables may already have sort orders that can be beneficial to the summary table processing. As SAS does not automatically "carry" SORTEDBY data set attributes from one step to another, we explicitly determined a beneficial SORTEDBY attribute and handed it off to a subsequent file when possible. (Note the *TableExtract* macro for examples.)

F. USE OF CONTROL-FLOW INTERRUPTER STATEMENTS

Judicious use of LEAVE and RETURN statements avoids unnecessary execution of statements within a DATA-step, reducing CPU time. The LEAVE statement breaks out of the current DO loop or SELECT group and proceeds with the next statement following the DO loop or SELECT group while the RETURN statement returns control to the top the DATA-step. (Note the modules *Dx_code_files, Px_code_files,* and *Rx_attach_keys for examples)*

G. USE OF DISK-STORAGE USAGE REDUCTION TECHNIQUES

Aggressive cleanup of temporary files minimizes the disk storage footprint and reduces the chances of job/system crashes due to running out of disk storage.

This is accomplished by:

- Purging files in the WORK storage area at the start of each new module
- Purging files in the DPTemp storage area as soon as they no longer needed
- Arranging the execution order of the modules to minimize the storage requirements for the DPTemp files.

  For example, instead of executing all extraction-related modules sequentially, which would write to disk many large files long before they are needed, we instead sequentially organize all of the modules needed to produce all measures for given type of data (e.g. diagnosis, procedure, dispensing), which allows us to aggressively delete files from DPTemp.

Aggressive cleanup of temporary files has another benefit. It improves the efficiency of file caching by freeing up the cache for new data.

---

[i] "Flexibility by Design: A Look at New and Updated System Options in SAS® 9.4", Jan Squillace, SAS Technical Support, Cary, NC

[ii] We did not compare the efficiency of M:M in-memory hash table lookups using DATA-step non-unique keys hash tables with a hash iterator object versus PROC SQL-step fuzzy hash joins.