Coding Time Windows in SAS

Authors: Rebecca Vick and Nathan Shippee

Current Version Date: 4/22/2026

Introduction

When analyzing data for research, it is sometimes necessary to code time windows. In this article we present several examples of programming common administrative healthcare data analytics that involve time in SAS.

Background

From time to time, researchers want to analyze health care events occurring within a specific time frame. Some examples include (but are not limited to):

- Assessing events occurring within a specified time window after an index event
- Connecting records occurring within the same time window
- Evaluating continuous enrollment

Although easy to conceptualize, it can seem daunting to program, especially to beginners. This article presents a few examples of coding time windows using administrative healthcare data. When coding, there are usually many ways to produce the same result. The examples here could potentially be coded in fewer, more complex lines of code but were constructed with coding beginners in mind. We used SAS software for these examples, but the steps could be translated into other scripting languages.

In the coding examples, SAS syntax is presented in the left-hand column, and notes describing the syntax in the right. Syntax keywords and functions are written in capital letters, and variables and datasets are in lowercase. Indentation was applied for readability and is good practice in your own programs.

Validation steps are mixed throughout with descriptions in red text. Always conduct validation throughout an analysis to ensure that your results contain the information you intended it to.

Before beginning any analysis, it's good to establish study parameters. In large real-world data sets you almost always come across anomalies and outliers. Defining parameters upfront can help avoid unwieldiness in your analysis due to these unusual cases.[1.1][SLJ1.2] Understanding your input variables, ranges, limitations, etc. can also help inform your choice of parameters. For example, when studying inpatient stays, limiting stay length to 30 days helps limit the time frame that must be searched for relevant physician claims, which are found in a separate file. Example 2 shows in detail how and why a parameter like this was established.

Examples

Example 1: Assessing events occurring within a specified time frame after an index event

When analyzing data for research, it is sometimes necessary to code time windows. In this article we present several examples of programming common administrative healthcare data analytics that involve time in SAS.

The first example shows one way to categorize and count hospital stays to calculate a 30-day readmission rate (preview of /the final product). We used MedPAR data, but the code can apply to many different files and claim types. Before embarking on an analysis, it's good to think about the best way to calculate a measure given your aims. A readmission rate requires a numerator (readmissions) and denominator (index admissions). What goes into each could differ depending on the project. For example, some researchers may want to exclude rehabilitation stays. Others may have the ability to identify planned admissions and exclude them from readmissions counts. There can be many possible approaches to an analytic question, and the researcher must decide what makes the most sense for their project.

In this example, we imagine being limited to one calendar year's data. We count admissions occurring between January and November and associated 30-day readmissions occurring between February and December to achieve a readmission rate for the year. In Step 1, we limit the input data to stays we might want to include in the measure. In Step 2 we apply SAS's LAG and INTCK functions to compute the number of days between stays. Step 3 flags particular admission types. These flags are used in the last step – Step 4 - to exclude certain admissions from the numerator, denominator, or both (these types of technical decisions are up to each researcher depending on their aims).

Step 1: Select FFS short hospital stays, excluding rehabilitation stays. Sort by bene id and admission date, which will arrange the stays chronologically in preparation for the LAG function in Step 2.

Example 1, Step 1

IDENTIFY SHORT-STAYS IN MEDPAR	**Notes**
`PROC SQL;`	Begin SQL procedure
`CREATE TABLE short_stays AS`	Create new dataset, "short stays".

SELECT bene_id, medpar_id, admsn_dt, dschrg_dt, bene_death_dt, prvdr_num	Select variables to include
FROM medpar.medpar_2019	Source the data from existing data set "medpar_2019", which is in the MedPAR library.
WHERE ss_ls_snf_ind_cd='S' AND infrmtl_enctr_ind_sw = 'N' AND prvdr_num_spcl_unit_cd NOT IN ('R', 'T', 'Y')	Only include FFS (infrmtl_enctr_ind_sw='N') short stay (ss_ls_snf_ind_cd = 's') records that are not rehab stays (prvdr_num_spcl_unit_cd NOT IN ('R', 'T', 'Y')).
ORDER BY bene_id, admsn_dt;	Sort output in ascending order by the variables listed. Ascending order is the default (specify 'DESC' after variable name to sort in descending order).
QUIT;	End SQL Procedure

Step 2: Use a SAS data step and the LAG function to place date and provider variables from the beneficiary's previous stay onto the current stay. The INTCK function calculates the number of days between stays. The provider identifiers will be used to identify transfers in Step 3. Although transfers will be removed from final admission and readmission counts in Step 4, they are included in the calculation of the number of days between stays at the bottom of this step. This means that the discharge date on the transfer will mark the beginning of the 30-day readmission window.

Example 1, Step 2

AGGREGATE EACH LINE INTO ONE STAY	**Notes**
DATA short_stay_lags;	Create new data set, "short_stay_lags".
SET short_stays;	Use "short_stays" dataset created in Step 1 as the input.
BY bene_id;	Group by beneficiary
FORMAT dschrg_dt_prev DATE9.;	Initialize and format new date variable.
FORMAT prvdr_num_prev $10.;	Initialize and format new character variable.

`dschrg_dt_prev = IFN(NOT(FIRST.bene_id),LAG(dschrg_dt),.);`	IFN returns a numeric value based on whether the condition is true, false, or missing. FIRST takes the first record in the BY group specified above (bene_id). LAG takes the value of the variable specified (dschrg_dt) from the previous record (default). This line of code says if current record is not the first one for the beneficiary (as sorted), then set dschrg_dt_prev to the dschrg_dt of the previous record, otherwise set to blank.
`prvdr_num_prev = IFC(NOT(FIRST.bene_id),LAG(prvdr_num),.);`	IFC returns a character value based on whether the condition is true, false or missing. Do the same as the previous line of code but for prvdr_num.
`IF dschrg_dt_prev ^=. THEN time2readmit = INTCK('DAYS', dschrg_dt_prev, admsn_dt, 'C');`	If there is a previous record for this beneficiary, this field will not be blank and use the INTCK() function to calculate the number of days between the end of the previous stay and the start of the current stay.
`RUN;`	

Validation: Double click the icon for the short_stay_lags dataset (on the left side of the screen under the Project Tree, or in your SAS work library under the Server directory). Visually inspect it for reasonableness. Specifically, spot check that the newly constructed variables are as intended. Oftentimes, problems can be spotted immediately when doing this simple step.

NOTE: The LAG function can be adjusted to look back further than one row. Specify the number of look-back rows after the word LAG. For example, LAG3 (dschrg_dt,3) will return the discharge date from the record three rows preceding the current one. The default (i.e., nothing specified) looks back one row. **WARNING:** The LAG function may not return the results you desire when used within a conditional statement (e.g., IF Var1=0 THEN Var3=LAG(Var2)). The statement will execute only when the condition is true, therefore the lagged value from the 'previous' record will be the value stored the last time the statement was executed.

Step 3: Create indicator variables for potential index admissions, readmissions, transfers, and deaths, which will allow flexibility in choosing final index admissions (denominator) and readmissions (numerator) in the next step.

Example 1, Step 3

CREATE INDICATORS FOR REHABILITATION AND TRANSFERS	Notes
PROC SQL;	
CREATE TABLE admission_inds AS	
SELECT *,	Keep all variables
CASE WHEN dschrg_dt<= '01DEC2019'd THEN 1 ELSE 0 END AS index_time_ind,	Flag potential index admissions. Set new variable "index_time_ind" equal to 1 when dschrg_dt<='01DEC2019'd, otherwise, set to 0. This date limit allows for 30 days run-out to (input dataset only goes through Dec. 31st, 2019).
CASE WHEN time2readmit BETWEEN 0 AND 30 AND dschrg_dt_prev <= '01DEC2019'd THEN 1 ELSE 0 END AS within30days_ind,	Flag admissions occurring within 30 days of an index admission
CASE WHEN bene_death_dt = dschrg_dt THEN 1 ELSE 0 END AS death_ind,	When death date is equal to the discharge date, the patient died during the stay so set to 1, otherwise 0.
CASE WHEN time2readmit IN (0,1) AND prvdr_num ^= prvdr_num_prev THEN 1 ELSE 0 END AS transfer_ind	When time to readmission is 0 or 1 day and the provider numbers are different (i.e. patient moved to a different hospital), flag as a transfer (the gap in days for categorizing stays as transfers is up to the researcher).
FROM short_stay_lags;	
QUIT;	

Validation:

1. Open the admissions dataset and spot check newly constructed variables to ensure they are working as intended.
2. Do a "gut check" on the counts of each indicator variable. Sum each one and compare it to the total record count and to each other. Do they look unbelievably high or low given what you know about each category? If yes, check for coding errors.

Example 1, Step 3 VALIDATION

STEP 3 VALIDATION - COUNT TOTAL DUPLICATE CARRIER CLAIMS	Notes
PROC SQL;	

CREATE TABLE ind_totals AS	
SELECT COUNT(*) AS total_recs,	Count all records in dataset
SUM(index_time_ind) AS index_time_recs,	Sum index_time_ind, which will sum the number of records in this category
SUM(within30days_ind) AS within30days,	Sum within30days_ind, which will sum the number of records in this category
SUM(death_ind) AS deaths,	Sum death_ind, which will sum the number of records in this category
SUM(transfer_ind) AS transfers	Sum transfer_ind, which will sum the number of records in this category
FROM admission_inds;	
QUIT;	

Results of "ind_total" passed into Excel. Percentages of total were calculated in the second row:

total_recs	index_time_recs	readmits	deaths	transfers
9,700,173	8,914,006	1,718,534	309,801	187,230
100.0%	91.9%	17.7%	3.2%	1.9%

Table caption: Percentage of all records attributed to 1. index admissions (91.9%), 2. readmissions (17.7%), 3. deaths (3.2%), and 4. transfers (1.9%).

None of the category percentages look unbelievable. We expect index_time_rec to be around 8% lower than the total record count because it does not include stays ending in December. And based on what we know or can deduce about the other categories, nothing looks unreasonable.

Step 4: Using combinations of indicators created in Step 3, set final index admissions and readmissions to 1 and sum together for final counts.

Example 1, Step 4

COUNT TOTAL NUMBER OF ADMISSIONS AND READMISSIONS	Notes
PROC SQL;	
CREATE TABLE admissions_summary AS	
SELECT	

`SUM(CASE WHEN index_time_ind = 1 AND transfer_ind = 0 AND death_ind = 0 THEN 1 ELSE 0 END) AS index_admissions,`	Set all index admissions, excluding transfers and stays during which the patient died, to 1 and sum together
`SUM(CASE WHEN readmit_ind = 1 AND transfer_ind = 0 THEN 1 ELSE 0 END) AS readmissions`	Set all readmissions, excluding transfers, to 1 and sum together
`FROM admission_inds;`	
`QUIT;`	

Finalizing results in Excel can sometimes be easier than doing it in SAS. The figure below shows the results from the above copied and pasted into Excel with final a calculation and format applied.

Final readmission rate calculated in Excel using output from "admissions_summary":

index admissions	readmissions	readmission rate
8,458,248	1,546,891	18.3%

Table caption: Final readmission count (1.5 million) and rate (18.3%) calculated in Excel.

Jump to beginning of Example 1

Example 1 Step Recap:

1. Select admissions
2. Calculate days between stays and prepare for identification of transfers
3. Create indicators for index admissions, readmissions, deaths, and transfers
4. Refine index admissions and readmissions using indicators from step 3, tally both, and calculate readmission rate

Example 2: Connecting records occurring within the same time window

The second example links different record types from a single healthcare event: Carrier (professional/physician) and Inpatient (facility) records from the same hospital stay. The final product will be a dataset that contains one row per stay with a unique stay id (MedPAR ID) and a field containing the total Carrier allowed amount associated with the stay. The stay id could then be used to attach the Carrier amounts onto the completed stay record.

Carrier and Inpatient claims are housed in separate files that lack a pre-made unique key for linking them together. In this example, we link using a logical set of variables and conditions that works to a high (but not absolute) degree of accuracy. Linkage accuracy should be judged and adjusted by the researcher depending upon the aims of their project.

Before beginning an analysis, it is important to understand the contents of the data and to think through how it may impact a project. In this example, we use annual MedPAR files for the IP records, and monthly Carrier files for the professional records. Inclusion in an annual MedPAR file is based upon the year of discharge (regardless of year of admission); and in Carrier files, it is based upon the month of the claim through date (regardless of claim from date, but for Carrier claims, the two dates usually match). For example, inpatient stays with a discharge date in 2019 will be included in the 2019 MedPAR file; and Carrier claims with a claim through date in March 2019 will be included in the March 2019 Carrier file.

The date of service on a physician bill that was incurred during an inpatient stay should fall between the inpatient admit and discharge dates, therefore we need a range of Carrier files that will cover all dates between the minimum IP admit date and the maximum IP discharge date. In large, real-world data sets you almost always encounter outliers and anomalies. For example, 0.03% of MedPAR hospital short stay records with a discharge date in January 2019 have an admit date prior to December 1st, 2018, with the earliest being March 25, 2017! Unless the answer to a question is likely affected by these types of cases (e.g., long or high-cost stays), capturing all of the associated carrier claims would be burdensome yet have no significant effect on results. Defining parameters upfront can help to avoid unwieldiness in your data. In this first example, we limit the data to FFS inpatient short stays lasting 30 days or less with discharge dates between January and March 2019. These parameters make the required date range for the input data known and manageable.

Step 1: Select inpatient (IP) records of interest. In this example, fee-for-service (FFS) (infrmtl_enctr_ind_sw = 'N') short stays (ss_ls_snf_ind_cd = 'S') lasting 30 days or less (dschrg_dt - admsn_dt <= 30) with a discharge date in the first quarter of 2019.

Example 2, Step 1

CREATE MEDPAR TABLE	**Notes**
`PROC SQL;`	Begin SQL procedure
`CREATE TABLE short_stays AS`	Create new dataset (short stays)
`SELECT bene_id, medpar_id, admsn_dt, dschrg_dt`	Select variables to include
`FROM medpar.medpar_2019`	Source the data from existing dataset, “medpar_2019”, which is in the MedPAR library.

WHERE ss_ls_snf_ind_cd = 'S' AND dschrg_dt - admsn_dt <= 30 AND infrmtl_enctr_ind_sw = 'N' AND dschrg_dt BETWEEN '01JAN2019'd AND '31MAR2019'd;	Only include records that meet these conditions
QUIT;	End SQL Procedure

Step 2: Stack monthly Carrier files one onto the other vertically (i.e., concatenate). In this example, it is easy to do because the monthly files have the exact same format (it would be more complex if variable names or formats differed among the files). Note that the 100% Carrier files are very large, which increases runtime. To minimize size, only carry necessary variables through.

Example 2, Step 2

CREATE CARRIER TABLE	**Notes**
DATA carrier_claims (KEEP=bene_id clm_id clm_from_dt clm_thru_dt nch_carr_clm_alowd_amt);	Create a new table (carrier_claims). List the variables separated by spaces (not commas. This is typical in DATA steps, as opposed to PROC SQL, which uses commas).
SET rif2018.bcarrier_claims_12 rif2019.bcarrier_claims_01 rif2019.bcarrier_claims_02 rif2019.bcarrier_claims_03;	List the names of the input files separated by spaces. Notice the inclusion of the December 2018 file for physician bills from stays that began prior to January 2019. Although not an issue in this example, note that the SET command keeps all variables from the input files even if they are not common to all files. You can achieve the same result using the PROC SQL OUTER UNION command, but it would require more lines of code and more specifications to preserve uncommon variables.
RUN;	End SQL Procedure

Step 3: Attach Carrier records to the IP stays from step 1. The LEFT JOIN (as opposed to RIGHT or FULL JOIN) preserves all records from the "left" table. Records in the "right" table without a matching record on the left will be dropped. This is a one-to-many merge due to there being multiple carrier records per stay, therefore the record count will be greater than that from Step 1.

Validating the merged table is an important part of this step. In our example, we find Carrier claims that join to multiple stays. If not discovered, those Carrier amounts would have unknowingly been double counted.

Example 2, Step 3

MERGE MEDPAR AND CARRIER TABLES	Notes
`PROC SQL;`	
`CREATE TABLE ss_carrier_merge AS`	
`SELECT a.bene_id, a.medpar_id, a.admsn_dt, a.dschrg_dt, b.*`	Select variables from input datasets. The 'a.' and 'b.' before variable names indicate the source dataset. These abbreviations are assigned in the FROM statement. The asterisk in 'b.*' means that all variables in dataset b (carrier_claims) will be included.
`FROM short_stays a`	Label the short_stays data set, 'a'.
`LEFT JOIN carrier_claims b`	Join the short_stays and carrier_claims datasets together. Label carrier_claims 'b'. A left join preserves all records in the 'FROM' table (short_stays). Records in carrier_claims that do not match at least one record in short_stays on the variables specified will be dropped.
`ON a.bene_id=b.bene_id AND b.clm_from_dt BETWEEN a.admsn_dt AND a.dschrg_dt`	Join carrier claims that occurred for the same beneficiary within the same time window as the inpatient record.
`ORDER BY a.bene_id, a.medpar_id, a.admsn_dt, a.dschrg_dt, b.clm_from_dt, b.clm_thru_dt;`	Sort output in ascending order by the variables listed. Ascending order is the default (specify 'DESC' after variable name to sort in descending order).
`RUN;`	

Validation:

1. Open the ss_carrier_merge dataset and visually inspect it. Oftentimes problems can be spotted immediately when taking this simple action. For example, did IP records successfully link to Carrier claims? If yes, does it appear to be to a degree that seems reasonable? For links, does the clm_from_dt fall between the admsn_dt and dschrg_dt?
2. Check for Carrier claims that link to multiple stays (i.e., duplicates). We know that when a hospital patient is transferred to a different unit (e.g., rehab), the transfer will show up in a separate record from the initial stay. Knowing this, we hypothesize that a doctor bill incurred on the transition day would be joined to both the initial and the rehab records, so it is necessary to see how often this happens before moving forward.

Example 2, Step 3 VALIDATION

STEP 3 VALIDATION - COUNT TOTAL DUPLICATE CARRIER CLAIMS	Notes
`PROC SQL;`	
`CREATE TABLE carrier_counts AS`	
`SELECT clm_id,`	
`COUNT(*) AS car_cnt`	Count how many times a unique Carrier claim id is encountered, i.e., duplicates.
`FROM ss_carrier_merge`	
`WHERE clm_id ^= .`	Ignore records that do not contain a Carrier join.
`GROUP BY 1;`	Group by CLM_ID
`QUIT;`	
`PROC FREQ DATA=carrier_counts;`	Generate frequency distribution of car_cnt.
`TABLE car_cnt/MISSING;`	Specify variable(s) for frequency table and include missing values. For a crosstab, list two variables separated by an asterisk (x*y).
`RUN;`	

Results of Proc Freq:

car_cnt	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	29,742,053	99.12	29,742,053	99.12
2	263,696	0.88	30,005,749	100
3	365	0	30,006,114	100

Table caption: Frequency of carrier records that link to one hospital stay (99.12%), two stays (0.88%) and three stays (<0.01%).

STEP 3 VALIDATON - REVIEW IMPACT OF DUPLICATES ON REIMBURSEMENT	Notes

PROC SQL;	
CREATE TABLE ss_carrier_merge_2 AS	
SELECT a.*,	
b.car_cnt	Attach carrier claim counts.
FROM ss_carrier_merge a	
LEFT JOIN carrier_counts b	Join merged table to carrier claim counts.
ON a.clm_id = b.clm_id;	Join by clm_id
QUIT;	
PROC SQL;	Quantify the dollar impact of the duplicates by totaling carrier payments by carrier claim count.
CREATE TABLE CAR_DOLLARS AS	
SELECT car_cnt,	
SUM (nch_carr_clm_alowd_amt) AS car_allwd_amt	Sum clm_pmt_amt
FROM ss_carrier_merge_2	
GROUP BY 1;	Sum amounts by carrier claim count.
QUIT;	

Pasting the results of the CAR_DOLLARS table into Excel allows us to quickly compute metrics that provide a sense of scale for duplicate links. As shown below, columns 1 and 2, and rows 1-4 contain the output of the SAS table. Columns 3-5, and row 5 contain calculations made in Excel. The shaded cells tell us the percentage of dollars associated with ambiguous links. It is up to each researcher to weigh the implications of issues like these and decide how to handle them.

car_cnt	car_allwd_amt	percent of total	deduped total	percent of deduped total
1	$ 3,862,293,463	97.4%	$ 3,862,293,463	98.7%
2	$ 103,272,715	2.6%	$ 51,636,358	1.3%
3	$ 209,595	0.0%	$ 139,730	0.0%
Total	$ 3,965,775,773	100.0%	$ 3,914,069,550	100.0%

Table caption: Percentage of total carrier claim costs attributable to single stay links (98.7%), double stay links (1.3%) and triple stay links (<0.01%).

Other variables (provider specialty, NPI, taxonomy codes, and others) could be added to help determine which of the ambiguous links are probably true. In this example, we decided to put dollar amounts from ambiguous links and unambiguous links in separate columns, which will provide flexibility in the analysis. For example, results could be tested for sensitivity with and without ambiguous link amounts.

Step 4: Aggregate Carrier allowed amounts separately for unambiguous and ambiguous links by stay.

Example 2, Step 4

SUM CLAIM PAYMENTS BASED ON UNAMBIGUOUS AND AMBIGUOUS LINKS	**Notes**
`PROC SQL;`	
`CREATE TABLE sum_carrier AS`	
`SELECT`	
`bene_id,`	
`medpar_id,`	
`SUM (CASE WHEN car_cnt<=1 THEN nch_carr_clm_alowd_amt ELSE 0 END) AS car_allwd_amt FORMAT COMMA18.2,`	Sum claim payments of unambiguous links and name result, "total_carrier_pmts".
`SUM (CASE WHEN car_cnt>1 THEN nch_carr_clm_alowd_amt ELSE 0 END) AS ambgs_car_allwd_amt FORMAT COMMA18.2,`	Sum claim payments of ambiguous links and name result, "ambgs_car_allwd_amt".
`FROM ss_carrier_merge_3`	
`GROUP BY bene_id, medpar_id;`	Sum carrier allowed amounts for each unique combination of bene_id and medpar_id (i.e., each unique stay).
`QUIT;`	

Step 5: Attach aggregated Carrier fields to MedPAR records. Validate final product.

Example 2, Step 5

ATTACH AGGREGATED CARRIER PAYMENTS TO MEDPAR	**Notes**

PROC SQL;	
CREATE TABLE short_stays_with_carrier AS	
SELECT A.*,	Select all variables from table 'a' (short_stays).
B.*	Select all variables from table 'b' (sum_carrier).
FROM short_stays a	
LEFT JOIN sum_carrier b	
ON a.bene_id = b.bene_id AND a.medpar_id = b.medpar_id;	
QUIT;	

Validation: Compare record counts in the first and final datasets to ensure they match.

Example 2, Step 5 VALIDATION

STEP 5 VALIDATON	**Notes**
PROC CONTENTS DATA=short_stays; RUN;	Describe contents of the dataset short_stays (including record count).
PROC CONTENTS DATA=short_stays_with_carrier; RUN;	Compare allowed amounts from the first carrier merged table, the second version of the same table, and the final table. We expect the total from the first table to be about 2.6% higher than the second and third because it includes duplicate and triplicate carrier amounts. The last two amounts should match. Notice how the amount from the second table is limited to carrier records with a single count, which we compare to the total of the unambiguous link amount in the final table.
PROC MEANS DATA=ss_carrier_merge SUM; VAR nch_carr_clm_alowd_amt; RUN;	Return sum of the variable nch_carr_clm_alowd_amt from the ss_carrier_merge dataset.

`PROC MEANS DATA=ss_carrier_merge_2 SUM; VAR nch_carr_clm_alowd_amt; WHERE car_cnt=1; RUN;`	Return sum of the variable nch_carr_clm_alowd_amt from the ss_carrier_merge_2 dataset for rows having a car_cnt=1
`PROC MEANS DATA=short_stays_with_carrier SUM; VAR car_allwd_amt; RUN;`	Return sum of car_allwd_amt – the total from unambiguous links – from the final dataset.

Results of code above:

The MEANS Procedure (table: ss_carrier_merge)

NCH Carrier Claim Allowed Charge Amount
Sum
5163487837

The MEANS Procedure (table: ss_carrier_merge_2)

NCH Carrier Claim Allowed Charge Amount
Sum
5030136001

The MEANS Procedure (table: short_stays_with_carrier)

car_allwd_amt
Sum
5030136001

Table caption: SAS Output showing that allowed charge totals (i.e., NCH Carrier Claim Allowed Charge Amount from ss_carrier_merge_2 table vs. car_allwd_amt from short_stays_with_carrier table) match appropriately.

Example 2 Step Recap

1. Select short stays from MedPAR file
2. Stack monthly Carrier files
3. Join Carrier and MedPAR records falling within the same time window
4. Aggregate Carrier allowed amounts by stay

5. Attach aggregated Carrier amounts to MedPAR records

Jump back to beginning of example 2

Example 3: Identifying members with continuous enrollment

To support a study's validity, it is sometimes necessary to require members to be enrolled in a certain type of coverage during the entire length of the study period, i.e., be **continuously enrolled**. This example shows various ways of identifying members who had continuous enrollment, depending on how defined, using the Master Beneficiary Summary File (MBSF).

The MBSF is an annual file that contains monthly enrollment information for each Medicare member enrolled for at least one day during the year. It contains several different arrays of monthly variables and variables with the total number of months for different types of coverage including Medicare Part A, Part B, Part D, Medicare Advantage and dual enrollment in both Medicare and Medicaid. For example, Part A and B monthly coverage is indicated in the twelve variables, MDCR_ENTLMT_BUYIN_IND_01 - MDCR_ENTLMT_BUYIN_IND_12. Additionally, there are two summary variables that provide the total number of months covered by Part A and Part B that year: BENE_HI_CVRAGE_TOT_MONS and BENE_SMI_CVRAGE_TOT_MONS, respectively.

Method 1

If your study requires members to be enrolled in traditional fee-for-service Medicare Parts A and B (i.e., no managed care) for all 12 months of a given year, you can select members from that year's MBSF using the summary variables for Parts A, B, and managed care:

Example 3, Method 1

Use annual summary enrollment variables to select members with continuous annual enrollment over the entire period	**Notes**
`DATA cohort1(KEEP=bene_id);`	Create a new dataset (cohort1). Include only one variable – bene_id.
`SET MBSF.MBSF_ABCD_2019;`	Set MBSF_2019 as input
`IF bene_hi_cvrage_tot_mons=12 AND bene_smi_cvrage_tot_mons=12 AND bene_hmo_cvrage_tot_mons=0 THEN output;`	Select beneficiaries who were enrolled in Medicare Part A and Part B all twelve months of the year and were never enrolled in a managed care plan during the year.
`RUN;`	Run data step

Method 1 will include members with continuous enrollment who died in December. To exclude all members who died during the period, add "AND bene_dt_dt=." to the if statement. Method 2 below identifies members with continuous enrollment the entire year plus members with continuous enrollment up until the date they died that year.

Method 2

The logic in method 1 would exclude members who passed away prior to December. To include them, you can still use the summary enrollment variables in combination with the bene_death_dt variable:

Example 3, Method 2

Use summary enrollment variables to select members with continuous annual enrollment over the entire period while alive	**Notes**
`DATA cohort2(KEEP=bene_id);`	Create a new dataset called 'cohort2'. Include only one variable – bene_id.
`SET MBSF.MBSF_ABCD_2019;`	Set MBSF_2019 as input dataset
`IF MISSING(bene_death_dt)THEN alivemo=12;` `ELSE alivemo = MONTH(bene_death_dt);`	Create a new variable (alivemo) to indicate the number of months the member was alive during the year. If there is no date of death, assume the member was alive all 12 months; if date of death is present, set number of months alive by applying the MONTH function to the date of death.

`IF bene_hi_cvrage_tot_mons=`
`bene_smi_cvrage_tot_mons= alivemo AND`
`bene_hmo_cvrage_tot_mons=0 THEN OUTPUT;`
	Only include members who had A&B coverage and no managed care for the entire time they were alive during the year.

`RUN`	Run data step

Method 3

The logic in method 2 would exclude those who were newly enrolled during the year. To include them, you can still use the summary enrollment variables:

Example 3, Method 3

Use summary enrollment variables to select members with continuous annual enrollment while alive, including new enrollees	**Notes**

`DATA cohort3(KEEP=bene_id);`	Create a new dataset (cohort3). Include only one variable – bene_id.
`SET MBSF.MBSF_ABCD_2019;`	Set MBSF_2019 as the input dataset.
`IF MISSING(bene_death_dt)THEN alivemo=12;` `ELSE alivemo = MONTH(bene_death_dt);`	Create a new variable (alivemo) to indicate the number of months the member was alive during the year. If there is no date of death present, assume they were alive all 12 months; if date of death is present, set number of months alive by applying the MONTH function to the date of death.

`IF YEAR(covstart)=2019 THEN covmo=13-`
`MONTH(covstart);`
`ELSE covmo=12;`
	Create a new variable, (covmo) to indicate the number of months covered by Medicare assuming they were alive the whole year. If the coverage start date falls within the study year, subtract the start month (determined using the MONTH function and the coverage start date) from 13 to arrive at the total number of months covered; otherwise, assume they were covered all 12 months.

`IF bene_hi_cvrage_tot_mons=`
`bene_smi_cvrage_tot_mons= MIN(alivemo,covmo) AND`
`bene_hmo_cvrage_tot_mons=0 THEN OUTPUT;`
	Only select members who had A&B coverage and no managed care the entire time that they were enrolled during the year. Their total months of actual Medicare coverage is determined by taking the minimum of alivemo and covmo.

`RUN`	Run data step

Method 4

The logic in method 4 looks for members with a minimum of six months continuous enrollment in fee-for-service (FFS) Medicare Parts A and B (i.e., no managed care; includes those dually eligible for Medicare and Medicaid). This method requires scanning through an array of monthly enrollment variables versus relying on summary enrollment variables. It seeks a more limited minimum months of enrollment than Methods 1,2&3 while avoiding members who disenrolled and reenrolled during the period such that their longest span of continuous enrollment was less than six months. This method will include new enrollees and members who died if they had at least six months continuous FFS A&B enrollment.

Note: When copying and pasting the code below into a SAS program file, the single quote marks may trigger a processing error when running. To correct this, delete the quotes and retype them.

Example 3, Method 4

Use array of enrollment variables to sum months of continuous enrollment	Notes
`DATA cohort4(KEEP=bene_id);`	Create a new dataset (cohort4). Include only one variable – bene_id.
`SET MBSF.MBSF_ABCD_2019;`	Set the 2019 MBSF as the input dataset.
`ARRAY monthcheckab {12} mdcr_entlmt_buyin_ind_01-mdcr_entlmt_buyin_ind_12;`	Create an array (monthcheckab), which absorbs data from the 12 mdcr_entlmt_buyin_ind monthly variables. These variables indicate whether a member had Medicare Part A, B or both that month.
`ARRAY monthcheckhmo {12} hmo_ind_01-hmo_ind_12;`	Create an array (monthcheckhmo), which absorbs data from the 12 hmo_ind monthly variables. These variables indicate whether a member was enrolled in a managed care plan in that month.
`ARRAY month {12} month1-month12;`	Create an array of 12-month indicators which will be filled in during the next steps.
`DO i=1 to 12;`	Set up a DO loop that will be iterated 12 times using variable 'i' as the counter.
`IF monthcheckab{i} IN ('3','C') AND monthcheckhmo{i} IN ('0','4') THEN month{i}=1; ELSE month{i}=0;`	Summarize the information in the (monthcheckab) and (monthcheckhmo) arrays to flag members who had FFS Medicare A&B coverage in each month. Note how 'i' is used to indicate the month in which you are looking.
`end;`	End the DO loop
`count6=month{1};`	Initialize a new variable (count6) that will be used in the next step to count continuous months of FFS A&B enrollment. Set it equal to the value of month in January (1=FFS A&B, 0=not FFS A&B).
`DO i=2 TO 12 UNTIL (count6=6);`	Set up a DO loop that is iterated until count6 equals 6. Note how the counter 'i' starts equal to 2 (i.e., February)
`IF month{i}=1 AND month{i-1}=0 THEN count6=1;`	If the member had FFS A&B coverage in the current month {i} but *not* in the previous month {i-1}, set count6 equal to 1 (meaning, this is the first month of FFS A&B enrollment during our study period).

`ELSE IF month{i}=1 AND month{i-1}=1 THEN count6+1;`	If the member had FFS A&B coverage in the current {i} *and* the previous month {i-1}, increase the value of count6 by 1. We need to look at both the current and previous month to ensure members had *continuous* enrollment.
`ELSE count6=0;`	If neither of the previous two conditions hold true (i.e., the member did not have FFS A&B coverage this month) set count6 equal to 0.
`END;`	End do loop
`IF count6=6 THEN OUTPUT;`	If the member achieved 6 months of continuous FFS A&B enrollment, output them to the final dataset (cohort4).
`run;`	Run data step

Jump back to beginning of example 3

Downloads

You can download a SAS program file containing the SAS code presented in this article below.

Coding_time_windows_in_SAS

Resources

SAS Institute Inc. *Programming Documentation for SAS® 9.4 and SAS® Viya® 3.5* : https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/pgmsaswlcm/home.htm.

Chronic Conditions Warehouse (CCW) *MedPAR Codebook*: https://www2.ccwdata.org/web/guest/data-dictionaries.

Chronic Conditions Warehouse (CCW) *Medicare Beneficiary Summary File (MBSF) Base with Medicare Part A, B, C, and D Version 2 Codebook*: https://www2.ccwdata.org/documents/10280/19022436/codebook-ffs-claims.pdfhttps://www2.ccwdata.org/web/guest/data-dictionaries.

Chronic Conditions Warehouse (CCW) *Medicare Fee-for-Service (FFS) Claims (Version L) Codebook* : https://www2.ccwdata.org/documents/10280/19022436/codebook-ffs-claims.pdfhttps://www2.ccwdata.org/web/guest/data-dictionaries.

H. Ament's *Learning to love the SAS LAG function*: https://www.lexjansen.com/phuse/2011/cc/CC08.pdf.