Paper CT15

Don't Loop—Hash! Slashing Clinical Merge Times with Five-Line In-Memory SAS Lookups

Hayk Nazaryan, Gurus CRO, Yerevan, Armenia

ABSTRACT

Legacy SDTM and ADaM pipelines frequently rely on PROC SORT, DATA step MERGE, or PROC SQL joins to combine large clinical datasets with relatively small reference or decode tables. While these approaches are robust and familiar, they often introduce unnecessary disk I/O and sorting overhead, especially when millions of observations are processed repeatedly across development and validation cycles. This paper demonstrates how SAS hash objects provide a simple, in-memory alternative that can significantly reduce runtime without requiring changes to hardware, licenses, or validated infrastructure. Using three common clinical programming scenarios—one-to-many reference range joins and last-value carry-forward (LOCF) derivations—we compare traditional approaches with hash-based solutions. The paper highlights a minimal five-line hash pattern, discusses performance implications, and provides practical guidance for safe adoption in regulated clinical environments.

INTRODUCTION

Clinical trial data processing routinely involves joining large SDTM or ADaM datasets with smaller lookup, reference, or mapping tables. Typical examples include attaching laboratory reference ranges, decoding categorical variables, or applying standard terminology mappings. In many legacy programs, these joins are implemented using PROC SORT followed by a DATA step MERGE, or by PROC SQL joins. Although these techniques are well understood and widely accepted, they often require multiple passes over large datasets and incur significant disk I/O due to sorting.

As clinical datasets grow in size and complexity, particularly in late-phase studies and integrated analyses, performance bottlenecks become more visible. Even modest inefficiencies can translate into minutes of additional runtime when programs are executed repeatedly during development, quality control, and submission preparation. This paper explores SAS hash objects as a targeted optimization technique for these scenarios.

OVERVIEW OF SAS HASH OBJECTS

A SAS hash object is an in-memory data structure that can be created and manipulated within a DATA step. Hash objects store key–value pairs and provide constant-time access to stored data based on defined keys. Unlike traditional MERGE or SQL joins, hash objects do not require input datasets to be sorted and do not rely on disk-based intermediate files.

Hash objects are particularly effective when one dataset is relatively small and can be fully loaded into memory, while the other dataset is large and processed sequentially. This pattern is common in clinical programming, where reference tables often contain hundreds or thousands of records, compared to millions of observations in SDTM or ADaM datasets.

THE FIVE-LINE HASH PATTERN

A key advantage of hash objects is that their basic usage can be expressed using a concise and repeatable coding pattern. In its simplest form, a hash object is declared and populated once at the start of a DATA step, and then queried for each observation of the main dataset.
The following example illustrates this minimal pattern:

```
if _n_ = 1 then do; ①
  declare hash h(dataset:"lookup"); ②
  h.defineKey("KEY"); ③
  h.defineData(all:"YES"); ④
  h.defineDone(); ⑤
end;

rc = h.find(); ⑥
```

Notes:
① Executes this block only once and ensures the hash table is built a single time. Prevents rebuilding the hash for every observation.

② Creates a hash object named h, loads the 'lookup' dataset into memory. This happens at DATA step runtime, not at compile time.

③ Defines the lookup key. Hash lookups are performed by key value and key must uniquely identify rows (in most use cases).

④ Specifies which variables to return from the lookup. all:"YES" will bring all variables from the lookup table. You can also specify only selected variables for safety.

⑤ Finalizes the hash definition. After this, the hash object is ready for use.

⑥ Performs the in-memory lookup. Searches the hash using the current key value. RC will be 0 if the match is found and will be non-zero if there is no match.

SCENARIO 1: ONE-TO-MANY REFERENCE RANGE JOINS

One common clinical programming task is the attachment of laboratory reference ranges to individual lab records. In this scenario, a large laboratory dataset contains many records per TESTCODE and SEX combination, while the reference range table contains exactly one record per that key. This represents a classic one-to-many join: a single reference-range record is reused across multiple laboratory observations. Please note that the datasets below are dummy datasets just for example.

LAB:

ID	SEX	TESTCODE	VISIT	ORRES
1	m	BLPRESS	Visit1	120
1	m	BLPRESS	Visit2	125
1	m	BLPRESS	Visit3	122
1	m	BLPRESS	Visit4	140
1	m	BLPRESS	Visit5	135
1	m	BLPRESS	Visit6	138
1	m	BLPRESS	Visit7	130
1	m	BLPRESS	Visit8	150
1	m	BLPRESS	Visit9	120
1	m	HRTRATE	Visit1	78
1	m	HRTRATE	Visit2	80
1	m	HRTRATE	Visit3	68
1	m	HRTRATE	Visit4	85
1	m	HRTRATE	Visit5	80
1	m	HRTRATE	Visit6	82
1	m	HRTRATE	Visit7	79
1	m	HRTRATE	Visit8	90
1	m	HRTRATE	Visit9	65
1	m	TEMP	Visit1	36.6
1	m	TEMP	Visit2	37.2
1	m	TEMP	Visit3	36.8
1	m	TEMP	Visit4	38.3
1	m	TEMP	Visit5	37.4

RANGE:

SEX	TESTCODE	LOWRANGE	HIGHRANGE
m	TEMP	36.7	38.5
m	HRTRATE	68	85
m	BLPRESS	125	140
f	TEMP	36.6	38.2
f	HRTRATE	65	38.2
f	BLPRESS	122	135

Joining these 2 datasets by data step merge:

```
/* Sorting the LAB dataset */
proc sort data=LAB;
  by SEX TESTCODE;
run;
```

```
/* Sorting the RANGE dataset */
proc sort data=RANGE;
  by SEX TESTCODE;
run;

/* Merging the LAB and RANGE datasets */
data final;
  merge LAB(in=a) RANGE;
  by SEX TESTCODE;
  if a;
run;
```

Joining these 2 datasets with the hash objects:

```
data final_hash;
  if 0 then set RANGE(keep=LOWRANGE HIGHRANGE);

  if _n_=1 then do;
    declare hash h(dataset:"RANGE");
    h.defineKey("SEX","TESTCODE");
    h.defineData("LOWRANGE","HIGHRANGE");
    h.defineDone();
  end;

  set LAB;
  rc = h.find();

  if rc ne 0 then do;
    LOWRANGE = .;
    HIGHRANGE = .;
  end;

  drop rc;
run;
```

Traditional implementations require sorting both datasets and performing a DATA step MERGE or SQL join. Using a hash object, the reference range table can be loaded into memory once, and each laboratory record can retrieve the corresponding ranges via a key-based lookup. This approach avoids sorting the large laboratory dataset and reduces overall I/O.

SCENARIO 2: LAST-VALUE CARRY-FORWARD DERIVATIONS

Beyond static lookups, hash objects can also be used to maintain state during DATA step execution. A practical example is last-value carry-forward (LOCF) derivations, where the most recent non-missing value for a subject and parameter must be retained and applied to subsequent records.

LAB:

ID	SEX	TESTCODE	VISIT	ORRES
1	m	BLPRESS	Visit1	120
1	m	BLPRESS	Visit2	125
1	m	BLPRESS	Visit3	122
1	m	BLPRESS	Visit4	.
1	m	BLPRESS	Visit5	135
1	m	BLPRESS	Visit6	138
1	m	BLPRESS	Visit7	130
1	m	BLPRESS	Visit8	150
1	m	BLPRESS	Visit9	.
1	m	HRTRATE	Visit1	78
1	m	HRTRATE	Visit2	80

1	m	HRTRATE	Visit3	68
1	m	HRTRATE	Visit4	85
1	m	HRTRATE	Visit5	.
1	m	HRTRATE	Visit6	82
1	m	HRTRATE	Visit7	79
1	m	HRTRATE	Visit8	90
1	m	HRTRATE	Visit9	.
1	m	TEMP	Visit1	.
1	m	TEMP	Visit2	37.2
1	m	TEMP	Visit3	36.8
1	m	TEMP	Visit4	.
1	m	TEMP	Visit5	37.4

```
/* LOCF using hash object */
data final;
  set LAB;
  by id testcode;

  length last_aval 8;

  /* Ensure last_aval exists in PDV */
  if _n_=1 then do;
    declare hash h();
    h.defineKey("id","testcode");
    h.defineData("last_aval");
    h.defineDone();
  end;

  /* Retrieve last value for this subject+param (if exists) */
  rc = h.find();

  /* Apply LOCF: if AVAL missing, fill from last_aval */
  if missing(AVAL) and rc=0 then AVAL = last_aval;

  /* Update last_aval when we see a non-misisng value */
  if not missing(AVAL) then do;
    last_aval = AVAL;
    h.replace(); /* insert or update for this key */
  end;

  drop rc last_aval;
run;
```

Final:

ID	SEX	TESTCODE	VISIT	ORRES
1	m	BLPRESS	Visit1	120
1	m	BLPRESS	Visit2	125
1	m	BLPRESS	Visit3	122
1	m	BLPRESS	Visit4	122
1	m	BLPRESS	Visit5	135
1	m	BLPRESS	Visit6	138
1	m	BLPRESS	Visit7	130
1	m	BLPRESS	Visit8	150
1	m	BLPRESS	Visit9	150
1	m	HRTRATE	Visit1	78
1	m	HRTRATE	Visit2	80

1	m	HRTRATE	Visit3	68
1	m	HRTRATE	Visit4	85
1	m	HRTRATE	Visit5	85
1	m	HRTRATE	Visit6	82
1	m	HRTRATE	Visit7	79
1	m	HRTRATE	Visit8	90
1	m	HRTRATE	Visit9	90
1	m	TEMP	Visit1	.
1	m	TEMP	Visit2	37.2
1	m	TEMP	Visit3	36.8
1	m	TEMP	Visit4	36.8
1	m	TEMP	Visit5	37.4

By storing the last observed value in a hash keyed by subject and parameter, programmers can implement flexible LOCF logic within a single DATA step. This approach can simplify complex BY-group processing and enables additional customization when standard RETAIN-based logic becomes difficult to maintain. This example demonstrates an implementation technique; the statistical appropriateness of LOCF must always be justified by the analysis plan.

PERFORMANCE CONSIDERATIONS AND LIMITATIONS

The primary performance benefit of hash objects arises from eliminating the need to sort large datasets and reducing disk I/O. On small datasets, runtime differences may be negligible; however, on large clinical pipelines, avoiding repeated sorts can result in substantial time savings.

In shared environments such as SAS Grid, AVD, or SAS Studio, large PROC SORT steps may consume substantial WORK space and can fail when disk quotas are reached. Hash-based approaches avoid sorting the large dataset, reducing I/O pressure and the risk of WORK-space exhaustion.

Hash objects are not a universal replacement for MERGE or PROC SQL. They are less suitable for many-to-many joins, very large lookup tables that do not fit comfortably in memory, or scenarios with poorly defined or unstable keys. Careful evaluation is required before introducing hash logic into validated production code.

CONCLUSION

SAS hash objects provide a powerful yet accessible technique for optimizing common clinical data-processing tasks. They are not intended to replace PROC SORT, DATA step MERGE, or PROC SQL joins in all situations. However, for common patterns involving a large fact table and a small, reusable lookup table, hash-based approaches provide a robust alternative that avoids unnecessary sorting, reduces dependency on WORK space, and scales efficiently across repeated pipeline executions.

CONTACT INFORMATION

Contact the author at:
Author Name: Hayk Nazaryan
Company: Gurus CRO
Address: 51 Aharonyan Str., Yerevan, Armenia
Email: h.nazaryan@gurus.am