



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

Hayk Nazaryan
Gurus CRO

AGENDA

1. Traditional Approaches
2. What Is a SAS Hash Object?
3. The Five-Line Hash Pattern
4. Scenario 1: One-to-Many Reference Range Join
5. Scenario 2: LOCF (Stateful Derivation with Hash)
6. Advantages and disadvantages of hash objects
7. Q&A

Why Are Clinical Merges Still Slow?

- **SDTM** and **ADaM** datasets often contain millions of rows
- We repeatedly join them with small reference or decode tables
- Typical solutions:
 - PROC SORT + MERGE**
 - PROC SQL LEFT JOIN**
- These approaches introduce heavy disk I/O
- Sorting is often more expensive than the join itself

Traditional Approaches

- Large **SDTM** / **ADaM** datasets often require **PROC SORT** before merging
- Sorting creates large temporary files in **WORK**
- In shared or limited environments (Grid, AVD, SAS Studio):
WORK space is limited
Disk I/O is shared
- As data volume grows, sorting can:
become the dominant runtime cost
fail completely due to insufficient WORK space

```
ERROR: Insufficient space in file WORK.BIG_S.DATA.  
ERROR: Failure while merging sorted runs from utility file 1 to final output.  
ERROR: Failure encountered during external sort.  
ERROR: File WORK.BIG_S.DATA is damaged. I/O processing did not complete.
```

What Is a SAS Hash Object?

- Resides in memory
- Created and used inside a **Data Step**
- Accessed by key
- Lookup time does not depend on table size
- Requires no sorting of the large dataset

Think of it as:

A DATA-step dictionary stored in RAM

The Five-Line Hash Pattern

```
if _n_ = 1 then do; ← Executes this block only once
  declare hash h(dataset:"lookup"); ← Creates a hash object named h from 'lookup.'
  h.defineKey("KEY"); ← Defines the lookup key (by variables)
  h.defineData(all:"YES"); ← Specifies which variables to return from the lookup
  h.defineDone(); ← Finalizes the hash definition
end;

rc = h.find(); ← Performs the in-memory lookup
```

Scenario 1: One-to-Many Reference Range Join

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

RANGES

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	85
f	BLPRESS	122	135

```
/* 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;
```

Scenario 1: One-to-Many Reference Range Join

```
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;
```



➤ This line **does not execute**

- It forces SAS to “know” these variables exist in this DATA step (PDV)
- Prevents errors like: “Undeclared data symbol LBNRLO ...”

Scenario 1: One-to-Many Reference Range Join

```
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;
```

Defining the hash object and loading the RANGE dataset with LOWRANGE, HIGHRANGE variables



Scenario 1: One-to-Many Reference Range Join

```
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;
```

Performs the hash lookup using the current values of: SEX, TESTCODE variables

Match found if RC = 0 and the LOWRANGE, HIGHRANGE variables will be filled

Scenario 1: One-to-Many Reference Range Join

```
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;
```

To handle 'no match' cases:

If no reference range record exists
for that key combination: keep
LOWRANGE/HIGHRANGE missing

Scenario 2: LOCF (Stateful Derivation with Hash)

LAB

sex	testcode	visit	id	aval
m	BLPRESS	visit1	1	120.0
m	BLPRESS	visit2	1	125.0
m	BLPRESS	visit3	1	122.0
m	BLPRESS	visit4	1	.
m	BLPRESS	visit5	1	135.0
m	BLPRESS	visit6	1	138.0
m	BLPRESS	visit7	1	130.0
m	BLPRESS	visit8	1	150.0
m	BLPRESS	visit9	1	.
m	HRTRATE	visit1	1	78.0
m	HRTRATE	visit2	1	80.0
m	HRTRATE	visit3	1	68.0
m	HRTRATE	visit4	1	85.0
m	HRTRATE	visit5	1	.
m	HRTRATE	visit6	1	82.0
m	HRTRATE	visit7	1	79.0
m	HRTRATE	visit8	1	90.0
m	HRTRATE	visit9	1	.
m	TEMP	visit1	1	.
m	TEMP	visit2	1	37.2
m	TEMP	visit3	1	36.8
m	TEMP	visit4	1	.
m	TEMP	visit5	1	37.4

```
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-missing 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

sex	testcode	visit	id	aval
m	BLPRESS	visit1	1	120.0
m	BLPRESS	visit2	1	125.0
m	BLPRESS	visit3	1	122.0
m	BLPRESS	visit4	1	122.0
m	BLPRESS	visit5	1	135.0
m	BLPRESS	visit6	1	138.0
m	BLPRESS	visit7	1	130.0
m	BLPRESS	visit8	1	150.0
m	BLPRESS	visit9	1	150.0
m	HRTRATE	visit1	1	78.0
m	HRTRATE	visit2	1	80.0
m	HRTRATE	visit3	1	68.0
m	HRTRATE	visit4	1	85.0
m	HRTRATE	visit5	1	85.0
m	HRTRATE	visit6	1	82.0
m	HRTRATE	visit7	1	79.0
m	HRTRATE	visit8	1	90.0
m	HRTRATE	visit9	1	90.0
m	TEMP	visit1	1	.
m	TEMP	visit2	1	37.2
m	TEMP	visit3	1	36.8
m	TEMP	visit4	1	36.8
m	TEMP	visit5	1	37.4

When Hash Pays Off

Hash helps most when:

- Big dataset is millions of rows
- Lookup table is small and reused
- Programs rerun many times (DEV/QC cycles)
- WORK is shared / limited (Grid, AVD, SAS Studio)
- Requires no sorting of the large dataset

Hash may not show big gains when:

- Data is small
- Everything already sorted/indexed
- Local WORK is very fast

When NOT to Use Hash Objects

- Many-to-many joins (multiple matches on both sides)
- Very large lookup tables (do not comfortably fit in memory)
- Poorly defined or unstable keys
- Simple one-off joins where SQL is clearer
- Situations where sorting is already required downstream

Q/A

Session

**THANK
YOU**

