2025-11-04

Side-by-side by Design: Pharma Data Handling with Merge, Join, Match, and Hash in R

Yutaka Morioka
Yuki Nakagawa

EPS corporation

morioka.yutaka038@eps.co.jp

Yutaka Morioka
Title: SAS Guru
Organization: EPS Corporation

A SAS programmer based in Japan and clinical data scientist. I actively disseminate SAS programming techniques from introductory to advanced levels and have presented at international and regional conferences including SAS Global Forum, CDISC Interchange Japan, Phuse Japan SDE, and PharmaSUG SDE. As an organizer of the SAS User's Group in Japan, I also focus on fostering collaboration between SAS and R programmers, aiming to create synergy across both communities.

Yuki Nakagawa
Title: SAS Programmer / Biostatistician
Organization: EPS Corporation

Since joining the Statistical Analysis Department of EPS Corporation in 2019, Yuki Nakagawa has been engaged in clinical trials service from SDTM to statistical analyses as the SAS Programmer and biostatistician.

Disclaimer and Disclosures

- The opinions expressed are those of the authors and do not necessarily reflect the official views or positions of their organizations.

- This presentation introduces various approaches to data joining**,**

but it does not intend to determine or recommend a single “best” method.

- The authors declare no conflicts of interest related to any of the methods, packages, or tools mentioned in this presentation..

Instruction

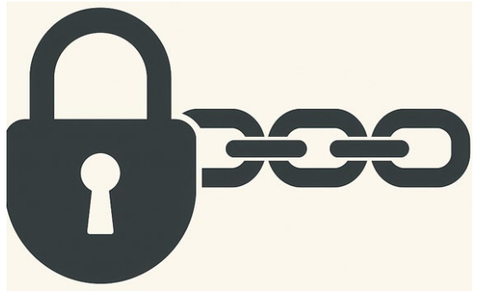
Instruction

In clinical trial data handling, data merging is arguably the most frequently used operation.

Especially in the derivation of SDTM and ADaM datasets, programmers often perform left joins using uniquely identifying keys such as USUBJID, SUBJID, VISIT, or TESTCD.However, few programmers take the time to re-examine what “joining” really means.

There are multiple ways to perform a join, each based on a different conceptual foundation. Moreover, the behavior of merge in SAS differs significantly from that in R, which often leads to confusion for programmers swiching from SAS to R.

Since misunderstanding how joins work can easily lead to unintended data errors, this presentation aims to take a deep and careful look at data joins from both a conceptual and practical perspective.



Base.R　merge function

TEST data - Base.R merge

```r
# DataFrame 11
df11 <- data.frame(
id = c(1, 2, 3),
value_A = c("A1", "A2", "A3"))
```

	id	value_A
1	1	A1
2	2	A2
3	3	A3

```r
# DataFrame 12
df12 <- data.frame(
id = c(2, 3, 4),
value_B = c("B2", "B3", "B4"))
```

	id	value_B
1	2	B2
2	3	B3
3	4	B4

TEST data(SAS dataset) -- Base.R merge

```
data wk11;
do id = 1 , 2, 3;
  value_A = choosec(id,"A1","A2","A3");
  output;
end;
run;
```

VIEWTABLE: Work.Wk11

	id	value_A
1	1	A1
2	2	A2
3	3	A3

```
data wk12;
do id = 2, 3, 4;
  value_B = choosec(id-1,"B2","B3","B4");
  output;
end;
run;
```

VIEWTABLE: Work.Wk12

	id	value_B
1	2	B2
2	3	B3
3	4	B4

Inner Join - Base.R merge

```
inner_joined <- merge(df11, df12, by = "id")
```

	id	value_A
1	1	A1
2	2	A2
3	3	A3

	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id	value_A	value_B
1	2	A2	B2
2	3	A3	B3

Full Outer Join - Base.R merge

```
full_outer_joined <- merge(df11, df12, by = "id", all = TRUE)
```

	id	value_A
1	1	A1
2	2	A2
3	3	A3

	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id	value_A	value_B
1	1	A1	*NA*
2	2	A2	B2
3	3	A3	B3
4	4	*NA*	B4

Left Outer Join - Base.R merge

```
left_joined <- merge(df11, df12, by = "id", all.x = TRUE)
```

	id	value_A
1	1	A1
2	2	A2
3	3	A3

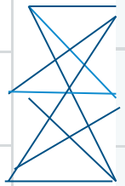
	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id	value_A	value_B
1	1	A1	*NA*
2	2	A2	B2
3	3	A3	B3

Cross Join - Base.R merge

```
cross_joined <- merge(df11, df12, by = NULL)
```

	id	value_A
1	1	A1
2	2	A2
3	3	A3



	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id.x	value_A	id.y	value_B
1	1	A1	2	B2
2	2	A2	2	B2
3	3	A3	2	B2
4	1	A1	3	B3
5	2	A2	3	B3
6	3	A3	3	B3
7	1	A1	4	B4
8	2	A2	4	B4
9	3	A3	4	B4

Base.R merge

```
merge(x, y,
      by = intersect(names(x), names(y)),
      by.x = by, by.y = by,
      all = FALSE,
      all.x = all, all.y = all,
      sort = TRUE,
      suffixes = c(".x", ".y"),
      incomparables = NULL,
      ...)
```

Argument	Description	Default
x, y	Data frames (or compatible objects) to join.	Required
by	Column name(s) to use as keys. If omitted, uses all common names: intersect(names(x), names(y)). If multiple, all are used as a composite key.	Common column names
by.x, by.y	Key column name(s) in x and y respectively—use when the key names differ between the two data frames.	Same as by
all	Logical. If TRUE, performs a full outer join; if FALSE, an inner join.	FALSE
all.x	Logical. If TRUE, keeps all rows from x (left join).	Same as all
all.y	Logical. If TRUE, keeps all rows from y (right join).	Same as all

Argument	Description	Default
sort	Logical. If TRUE, sorts the result by the key columns; if FALSE, preserves row order as much as possible.	TRUE
suffixes	Length-2 character vector with suffixes appended to non-key columns that have the same name in x and y.	c(".x", ".y")
incomparables	Values that should not match when merging (passed to match()). For example, set NA here to prevent NA values from matching each other.	NULL
...	Additional arguments passed to methods. Typically unused for data.frame but available for other methods.	

SAS Merge

SAS merge – Inner Join

```
proc sort data=wk11;
 by id;
run;

proc sort data=wk12;
 by id;
run;

data inner_join;
 merge wk11(in=in11)
       wk12(in=in22);
 by id;
 if in11 and in22;
run;
```

VIEWTABLE: Work.Wk11

	id	value_A
1	1	A1
2	2	A2
3	3	A3

VIEWTABLE: Work.Wk12

	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id	value_A	value_B
1	2	A2	B2
2	3	A3	B3

SAS merge – Full Outer Join

```
proc sort data=wk11;
 by id;
run;

proc sort data=wk12;
 by id;
run;

data outer_join;
 merge wk11
       wk12;
 by id;
run;
```

VIEWTABLE: Work.Wk11

	id	value_A
1	1	A1
2	2	A2
3	3	A3

VIEWTABLE: Work.Wk12

	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id	value_A	value_B
1	1	A1	
2	2	A2	B2
3	3	A3	B3
4	4		B4

SAS merge – Left Outer Join

```
proc sort data=wk11;
 by id;
run;

proc sort data=wk12;
 by id;
run;

data left_outer_join;
 merge wk11(in=in11)
       wk12(in=in22);
 by id;
 if in11;
run;
```

VIEWTABLE: Work.Wk11

	id	value_A
1	1	A1
2	2	A2
3	3	A3

VIEWTABLE: Work.Wk12

	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id	value_A	value_B
1	1	A1	
2	2	A2	B2
3	3	A3	B3

SAS – Cross Join

```
data cross_join;
set wk11;
 do i=1 to wk12obs;
  set wk12 nobs=wk12obs point=i;
  output;
 end;
run;
```

VIEWTABLE: Work.Wk11

	id	value_A
1	1	A1
2	2	A2
3	3	A3

VIEWTABLE: Work.Wk12

	id	value_B
1	2	B2
2	3	B3
3	4	B4

	id	value_A	value_B
1	2	A1	B2
2	3	A1	B3
3	4	A1	B4
4	2	A2	B2
5	3	A2	B3
6	4	A2	B4
7	2	A3	B2
8	3	A3	B3
9	4	A3	B4

R.Base mege VS SAS merge

```r
df21 <- data.frame(
id = c(1, 2, 3),
score = c(90, 80, 70)
)

df22 <- data.frame(
id = c(2, 3, 4),
score = c(85, 75, 65)
)

merged <- merge(df21, df22, by = "id")
```

	id	score
1	1	90
2	2	80
3	3	70

	id	score
1	2	85
2	3	75
3	4	65

	id	score.x	score.y
1	2	80	85
2	3	70	75

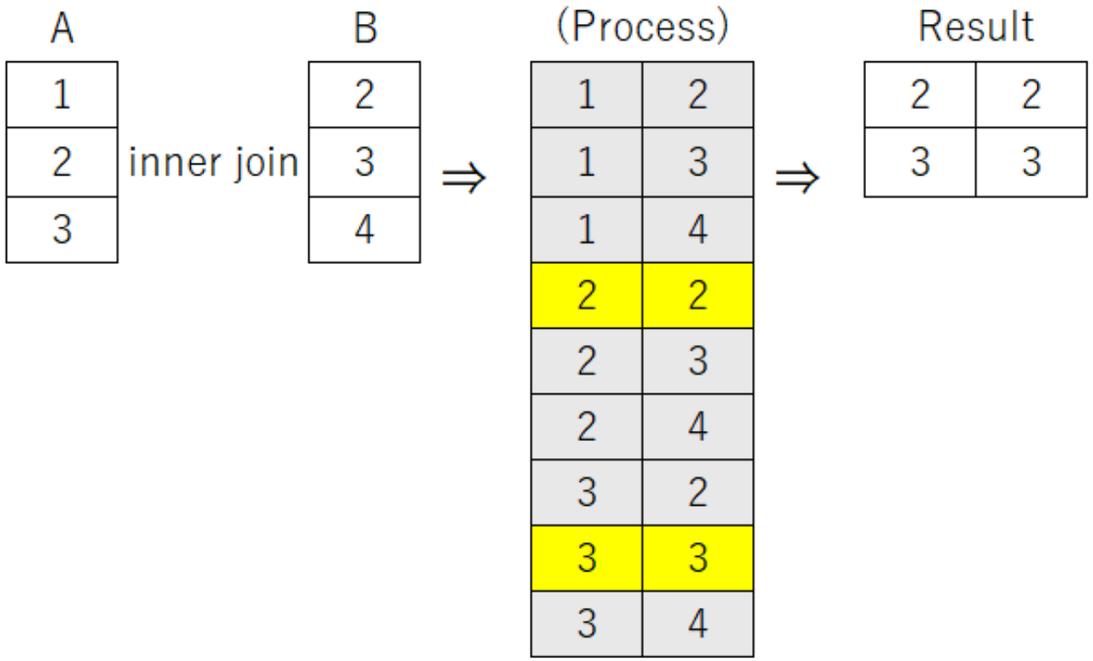
```sas
data wk21;
do id = 1 , 2, 3;
  score= choosen(id,90,80,70);
  output;
end;
run;
data wk22;
do id = 2, 3, 4;
  score = choosen(id-1,85,75,65);
  output;
end;
run;
proc sort data=wk21;
 by id;
run;
proc sort data=wk22;
 by id;
run;

data sas_merged;
 merge wk21(in=in11)
       wk22(in=in22);
 by id;
 if in11 and in22;
run;
```

VIEWTABLE: Work.Sas_merged

	id	score
1	2	85
2	3	75

Concepts of Internal Processing in SAS and R

R.merge Function
SQL Join



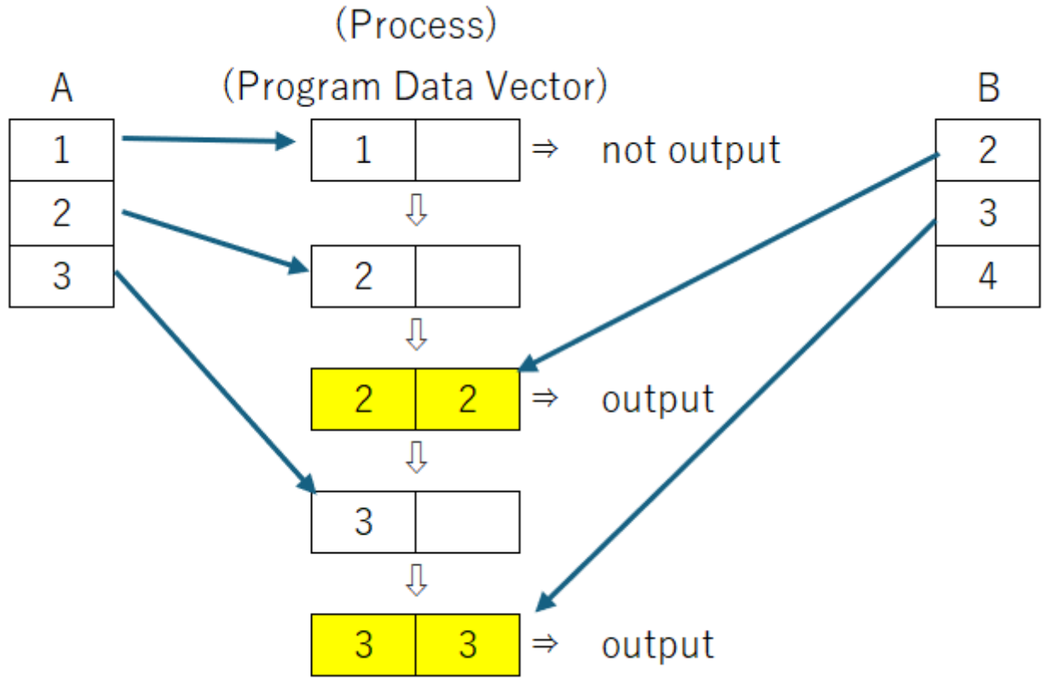
A
1
2
3

B
2
3
4

(Process)	
1	2
1	3
1	4
2	2
2	3
2	4
3	2
3	3
3	4

Result	
2	2
3	3

SAS



“Sequential processing” refers to the concept of implicitly looping through all records by reading one record, processing it, and then outputting one record — repeating this sequence for every record.

SAS MERGE vs R merge()

Aspect	SAS MERGE (DATA step)	R merge() (Base R)
Number of datasets	Can merge multiple datasets at once: MERGE d1 d2 d3;	Always two datasets only (for 3+, chain merges)
Key specification	BY required for keyed merges; without BY, matches by row number (one-to-one)	by optional: if omitted → all common column names are used as keys; if no common columns → cross join
Pre-sorting	Required: input datasets must be sorted (or indexed) by BY variables	Not required: keys matched internally; default output sorted by key (sort=TRUE)
Without BY/by	Row-number match (1st obs with 1st, 2nd with 2nd, etc.)	Equi-join on all common columns; if none → Cartesian product
Same variable names (non-keys)	Later dataset values overwrite earlier ones	Both retained with suffixes (.x, .y by default, customizable)
Many-to-many keys	Not SQL-like: sequential matching, can produce unexpected carry-over values	SQL-like: all pairwise combinations are generated

SAS MERGE vs R merge()

Aspect	SAS MERGE (DATA step)	R merge() (Base R)
Join types	By default, similar to full outer join (with missing values filled). Inner/left/right emulated using IN= dataset option + conditional logic	Explicit arguments: inner (default), all=TRUE (full), all.x=TRUE (left), all.y=TRUE (right)
Identifying unmatched rows	IN= option creates dataset flags (presence indicators)	Unmatched rows → NA; must create flags manually if needed
Row order	Determined by input order / BY-group order (depends on sort)	Default sorted by key; set sort=FALSE to preserve input order
Cross join	Not supported (BY omitted → row-number merge, not Cartesian product)	Supported via by=NULL (true Cartesian product)
Semi/anti join	Implemented with IN= flags + conditional logic	Not directly available; must filter after merge

dplyr

dplyr

Overview

dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

https://dplyr.tidyverse.org/

Test data – dplyr

```
library(dplyr)

df31 <- tibble(
id = c(1, 2, 3),
value_A = c("A1", "A2", "A3")
)

df32 <- tibble(
id = c(2, 3, 4),
value_B = c("B2", "B3", "B4")
)
```

	id	value_A
1	1	A1
2	2	A2
3	3	A3

	id	value_B
1	2	B2
2	3	B3
3	4	B4

Join – dplyr

```
inner_join <- inner_join(df31, df32, by = "id")
```

	id	value_A	value_B
1	2	A2	B2
2	3	A3	B3

```
full_join <- full_join(df31, df32, by = "id")
```

	id	value_A	value_B
1	1	A1	*NA*
2	2	A2	B2
3	3	A3	B3
4	4	*NA*	B4

```
left_join <- left_join(df31, df32, by = "id")
```

	id	value_A	value_B
1	1	A1	*NA*
2	2	A2	B2
3	3	A3	B3

```
cross_join <- cross_join(df31, df32)
```

#Cross Join (Before dplyr v1.1.0)

```
cross_join2 <- df31 %>%
mutate(dummy = 1) %>%
inner_join(df32 %>% mutate(dummy = 1), by =
"dummy") %>%
select(-dummy)
```

	id.x	value_A	id.y	value_B
1	1	A1	2	B2
2	1	A1	3	B3
3	1	A1	4	B4
4	2	A2	2	B2
5	2	A2	3	B3
6	2	A2	4	B4
7	3	A3	2	B2
8	3	A3	3	B3
9	3	A3	4	B4

dplyr:full_join ≠ SAS merge

```r
library(dplyr)
wk1 <- tibble(id = 1:3, val = c("A", "B", "C"))
wk2 <- tibble(id = 2:4, val = c("D", "E", "F"))
out1 <- full_join(wk1, wk2, by = "id")
```

```
data sas_out1 ;
merge wk1 wk2;
by id;
run;
```

	id	val
1	1	A
2	2	B
3	3	C

	id	val
1	2	D
2	3	E
3	4	F

	id	val.x	val.y
1	1	A	*NA*
2	2	B	D
3	3	C	E
4	4	*NA*	F

≠

VIEWTABLE: Work.Sas_out1

	id	val
1	1	A
2	2	D
3	3	E
4	4	F

dplyr:full_join ≠ SAS merge

```r
out2 <- rows_upsert(wk1, wk2, by = "id")
```

	id	val
1	1	A
2	2	D
3	3	E
4	4	F

In this case, `rows_upsert` behaves similarly to a SAS merge rather than a full join.

sassy

For SAS® programmers, encountering R for the first time can be quite a shock.

- Where is the log?
- Where are my datasets?
- How do I do a data step?
- How do I create a format?
- How do I create a report?

All these basic concepts that were so familiar and easy for you are suddenly gone. How can R possibly be a replacement for SAS®, when it can't even create a decent log!

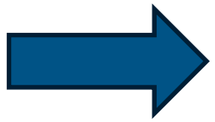
https://sassy.r-sassy.org/index.html

sassy – merge

```
wk1 <- tibble(id = 1:3, val = c("A", "B", "C"))
wk3 <- tibble(id = 2:4, val = c("D", "E", "F"),val2 = c("D", "E", "F") )
out3 <- rows_upsert(wk1, wk3, by = "id")
```

	id	val
1	1	A
2	2	B
3	3	C

	id	val	val2
1	2	D	D
2	3	E	E
3	4	F	F

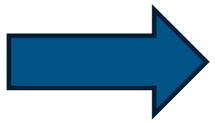


Console | Terminal | Background Jobs

R 4.5.1 · ~/

```
Error in `rows_upsert()`:
! All columns in `y` must exist in `x`.
i The following columns only exist in `y`: `val2`.
Run `rlang::last_trace()` to see where the error occurred.
```

```
library(sassy)
out4 <- datastep(
merge(wk1, wk3, by = "id", type = "full"),
{}
)
```



	id	val.x	val.y	val2
1	2	B	D	D
2	3	C	E	E

Even when using the sassy package to replicate SAS behavior in R, it does not fully replicate the merge statement.

“Perfect replication” is difficult because the SAS merge ... by ...; relies on a unique execution model:

- **PDV (Program Data Vector)**: values from the other dataset are retained and carried forward.
- **Sequential matching within BY groups:** duplicates are matched in order, not as a SQL-style Cartesian join.
- **Overwriting of same-named variables:** variables from later datasets overwrite earlier ones, while at the same time special flags like IN=, FIRST., and LAST. are available.

SQL

SQL – Join

```
/*Inner join*/
 select coalesce(wk11.id, wk12.id) as id
          ,value_A
          , value_B
        from wk11 inner join wk12 on wk11.id = wk12.id;
/*full outer join*/
 select coalesce(wk11.id, wk12.id) as id
          ,value_A
          , value_B
        from wk11 full outer join wk12 on wk11.id = wk12.id;
/*left outer join*/
 select coalesce(wk11.id, wk12.id) as id
          ,value_A
          , value_B
        from wk11 full outer join wk12 on wk11.id = wk12.id;
/*cross join*/
 select coalesce(wk11.id, wk12.id) as id
          ,value_A
          , value_B
        from wk11 cross join wk12 ;
```

id	value_A	value_B
2	A2	B2
3	A3	B3

id	value_A	value_B
1	A1	
2	A2	B2
3	A3	B3
4		B4

id	value_A	value_B
1	A1	
2	A2	B2
3	A3	B3

id	value_A	value_B
1	A1	B2
1	A1	B3
1	A1	B4
2	A2	B2
2	A2	B3
2	A2	B4
3	A3	B2
3	A3	B3
3	A3	B4

Various Ways to Implement SQL in R

```
library(sqldf)
sqldf <- sqldf("SELECT a.id, a.value_A, b.value_B
FROM df11 a
INNER JOIN df12 b
ON a.id = b.id")
```

1. sqldf packageThe most well-known methodAllows direct handling of dataframes with SQLUses SQLite internally for processing

```
library(DBI)
library(RSQLite)
con <- dbConnect(RSQLite::SQLite(), ":memory:")
dbWriteTable(con, "df11", df11)
dbWriteTable(con, "df12", df12)
dbi1 <- dbGetQuery(con, "SELECT a.id, a.value_A, b.value_B
FROM df11 a
INNER JOIN df12 b
ON a.id = b.id")
```

2. DBI + dplyr/dbplyrCombining R's database connection API (DBI) with dbplyr—an extension of dplyr—enables you to convert dplyr code into SQL for execution on databases.Furthermore, using DBI::dbGetQuery() allows you to write raw SQL directly.

```
library(duckdb)
con <- dbConnect(duckdb())
dbWriteTable(con, "df11", df11)
dbWriteTable(con, "df12", df12)
duckdb <- dbGetQuery(con, "SELECT a.id, a.value_A, b.value_B
FROM df11 a
INNER JOIN df12 b
ON a.id = b.id")
```

3. duckdb PackageUse DuckDB, an ultra-fast embedded DB engine, in RRun SQL queries directly on dataframesIt's becoming a staple for large-scale data processing

Base.R match

Base.R match

The match function compares vectors and returns an index.

```
x <- c("B", "C", "A", "D")
table <- c("A", "B", "C")
match(x, table)
```

```
> x <- c("B", "C", "A", "D")
> table <- c("A", "B", "C")
> match(x, table)
[1]  2  3  1 NA
```

Base.R match

```
df41 <- data.frame(
ID = c(101, 102, 103, 104),
Name = c("Alice", "Bob", "Carol", "Dave")
)

df42 <- data.frame(
ID = c(103, 101, 105),
Score = c(88, 85, 90)
)

match <- df41
match$Score <- df42$Score[ match(df41$ID, df42$ID) ]
```

	ID	Name
1	101	Alice
2	102	Bob
3	103	Carol
4	104	Dave

	ID	Score
1	103	88
2	101	85
3	105	90

	ID	Name	Score
1	101	Alice	85
2	102	Bob	*NA*
3	103	Carol	88
4	104	Dave	*NA*

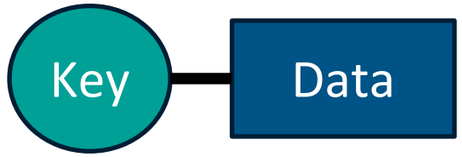
In short, match() is a “position-return” type join method a simplified version of SQL JOIN (limited to many-to-one, left joins)useful for fetching values in a lookup-like manner.

hash

https://cran.r-project.org/web/packages/hash/index.html

hash

```
library(hash)

# Creating a Hash
h <- hash(keys = c("a", "b", "c"), values = c(1, 2, 3))

# Reference to a Value
h["a"] # 1
h["b"] # 2

# Add new elements
h["d"] <- 4
h["z"]

# Existence check with has.key()
has.key("a", h) # TRUE
has.key("x", h) # FALSE

# Safe lookup: return NA if not exist
k <- "x"
if (has.key(k, h)) {
print(h[[k]])
} else {
print(NA)
}
```




```
> # Reference to a Value
> h["a"]   # 1
[1] 1
> h["b"]   # 2
[1] 2
>
> # Add new elements
> h["d"] <- 4
>
> h["z"]
NULL
```

```
> has.key("a", h)   # TRUE
   a
TRUE
> has.key("x", h)   # FALSE
    x
FALSE
```

```
[1] NA
```

hash

```
# Lookup Table（Country Code, Country Name）
lookup <- data.frame(
code = c("JP", "US", "FR"),
name = c("Japan", "United States", "France")
)
```

	code	name
1	JP	Japan
2	US	United States
3	FR	France

```
# Target Dataset
df <- data.frame(
subject = 1:5,
code = c("JP", "US", "CN", "FR", "JP")
)
```

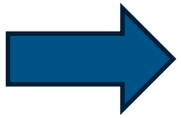
	subject	code
1	1	JP
2	2	US
3	3	CN
4	4	FR
5	5	JP

hash

```
library(hash)
# Create Hash
h <- hash(keys = lookup$code, values = lookup$name)

df$name <- vapply(df$code,
function(k) if (has.key(k, h)) h[[k]] else NA_character_,
FUN.VALUE = character(1))
```

	subject	code
1	1	JP
2	2	US
3	3	CN
4	4	FR
5	5	JP

	code	name
1	JP	Japan
2	US	United States
3	FR	France



	subject	code	name
1	1	JP	Japan
2	2	US	United States
3	3	CN	*NA*
4	4	FR	France
5	5	JP	Japan

Summary of the R hash package

- Fundamentally, it is a dictionary structure of “**one key → one value.**”
- While the value can be a vector or a list, it is not natural to define **multiple variables as keys** or to **manage multiple variables together as data.**
- Therefore, unlike the SAS hash object, it does not support something like **defineKey / defineData method** with multiple variables.
- Intuitively, its behavior is closer to a **proc format lookup table** in SAS (mapping a code to a label).

data.table

data.table

`data.table` provides a high-performance version of base R's `data.frame` with syntax and feature enhancements for ease of use, convenience and programming speed.

The `data.table` project uses a custom governance agreement and is fiscally sponsored by NumFOCUS. Consider making a tax-deductible donation to help the project pay for developer time, professional services, travel, workshops, and a variety of other needs.

NUMFOCUS
OPEN CODE = BETTER SCIENCE

https://github.com/Rdatatable/data.table?tab=readme-ov-file

data.table

Key Features of data.table in R

High Performance

- Extremely fast operations even with tens of millions of rows.
- Optimized algorithms for joins and aggregation.
- Reference-based operations minimize memory consumption.

Concise Syntax ([i, j, by])

Core structure: DT[i, j, by]
i: row filtering (similar to SQL WHERE)
j: column operations or aggregations (similar to SQL SELECT)
by: grouping (similar to SQL GROUP BY)
Enables SQL-like data manipulation in a very compact form.

Flexible and Fast Joins

Key-based joins with on= are highly efficient.
Supports equi-joins, non-equi joins, rolling joins, and multi-key joins.
Can reproduce INNER, LEFT, and FULL OUTER JOIN behavior.

data.table

```
library(data.table)

dt51 <- data.table(id = c(1,2,3), val1 = c("A","B","C"))
```

	id	val1
1	1	A
2	2	B
3	3	C

```
dt52 <- data.table(id = c(2,3,4), val2 = c(10,20,30))
```

	id	val2
1	2	10
2	3	20
3	4	30

Inner & Left Outer Join - data.table

```
#Inner Join
dt_out1 <- dt51[dt52, on = .(id), nomatch=0]
```

	id	val1	val2
1	2	B	10
2	3	C	20

```
#Left Outer Join (dt1 on the left side as the reference)
dt_out2 <- dt52[dt51, on = .(id)]
```

	id	val2	val1
1	1	*NA*	A
2	2	10	B
3	3	20	C

Full Outer & Cross Join - data.table

```
#Full Outer Join
dt_out3 <- merge(dt51, dt52, by="id", all=TRUE)
```

```
data.table::merge.data.table(dt51, dt52, by = "id", all = TRUE)
```

	id	val1	val2
1	1	A	*NA*
2	2	B	10
3	3	C	20
4	4	*NA*	30

```
#Cross Join
dt51_ <- dt51[, key := 1]
dt52_ <- dt52[, key := 1]
dt_out4 <- dt51_[dt52_, on=.(key)
, allow.cartesian=TRUE][, key := NULL]
```

	id	val1	i.id	val2
1	1	A	2	10
2	2	B	2	10
3	3	C	2	10
4	1	A	3	20
5	2	B	3	20
6	3	C	3	20
7	1	A	4	30
8	2	B	4	30
9	3	C	4	30

Rolling Join - data.table

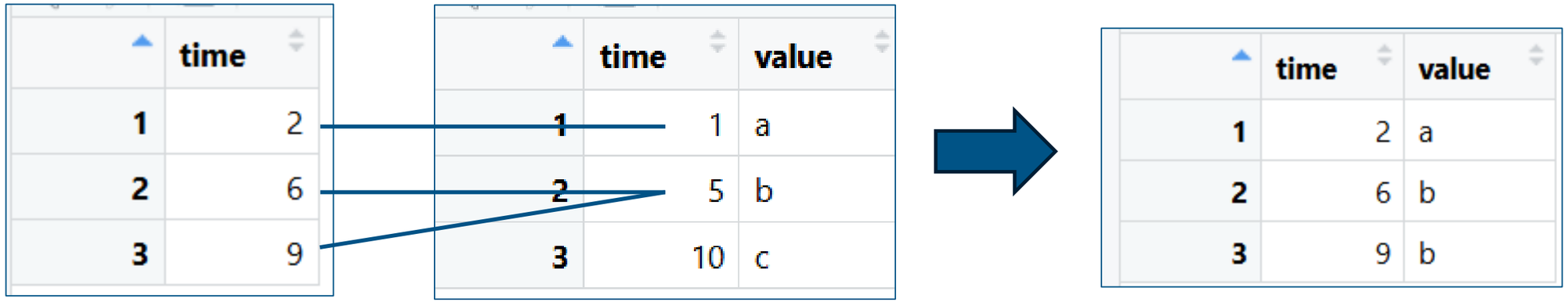
```
#Rolling Join
dt_query <- data.table(time = c(2,6,9))
```

	time
1	2
2	6
3	9

```
dt_time <- data.table(time = c(1,5,10)
, value = c("a","b","c"))
```

	time	value
1	1	a
2	5	b
3	10	c

```
#Join the most recent past value
dt_out5 <- dt_time[dt_query, on=.(time), roll=TRUE]
```



Rolling Join - data.table

Rolling join is an ordered join that fills in matches using the closest previous (or next) key value, when an exact match is not found.

For each value in the query table, find the closest matching key in the source table that does not exceed it (when roll=TRUE).

In data.table's Rolling Join, when keys are duplicated (tied),the value from the last occurrence (the final row) is adopted.

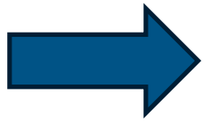
Option	Description
roll = TRUE	Carry the most recent past value forward (LOCF)
roll = Inf	Carry the most recent past value forward (LOCF)
roll = -Inf	Carry the next future value backward (NOCB)
roll = "nearest"	Use whichever value is closest (past or future)

Non-equi Join - data.table

```
# Non-equi Join (point-in-interval join)
dt_range <- data.table(start = c(1,5), end = c(3,10),
label=c("low","high"))
dt_point <- data.table(x = 1:10)

# Combine elements where x is within the range start <= x <= end
dt_out6 <- dt_range[dt_point, on=.(start <= x, end >= x), nomatch=0]
```

	x
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

	start	end	label
1	1	3	low
2	5	10	high



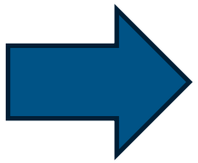
	start	end	label
1	1	1	low
2	2	2	low
3	3	3	low
4	5	5	high
5	6	6	high
6	7	7	high
7	8	8	high
8	9	9	high
9	10	10	high

Non-equi Join - data.table

```
# Non-equi Join
# Get the nearest previous or next value (without rolling)
X <- data.table(time = c(1, 5, 10), value = c("a", "b", "c"))
Y <- data.table(time = c(2, 6, 9))
setkey(X, time)
# Nearest previous (time < y)
prev <- X[Y, on = .(time < time), mult = "last"]
```

	time	value
1	1	a
2	5	b
3	10	c

	time
1	2
2	6
3	9



	time	value
1	2	a
2	6	b
3	9	b

Non-equi Join - data.table

```
# Non-equi Join
#Matching within a +-tolerance band
#Use non-equi conditions to find all matches inside [u – tol, u + tol],
#then pick the closest one (dist = abs(t – u)).
# Match within the tolerance band
X <- data.table(id=1, t=c(1,5,9,12), val=c("A","B","C","D"))
Y <- data.table(id=1, u=c(2,6,11), tol=c(2,1,3))
setkey(X, id, t)
# Precompute join columns
Y[, lower := u - tol]
Y[, upper := u + tol]
# Now perform the join safely
hits <- X[Y, on = .(id, t >= lower, t <= upper),
nomatch=0,.(id, u, tol, t, val, dist = abs(t - u))]
# Keep only the closest match per (id,u)
nearest <- hits[order(id, u, dist)][, .SD[1], by=.(id, u)]
```

	id	t	val
1	1	1	A
2	1	5	B
3	1	9	C
4	1	12	D

	id	u	tol
1	1	2	2
2	1	6	1
3	1	11	3

	id	u	tol	t	val	dist
1	1	2	2	0	A	2
2	1	6	1	5	B	1
3	1	11	3	8	C	3
4	1	11	3	8	D	3

	id	u	tol	t	val	dist
1	1	2	2	0	A	2
2	1	6	1	5	B	1
3	1	11	3	8	C	3

data.table vs. dplyr

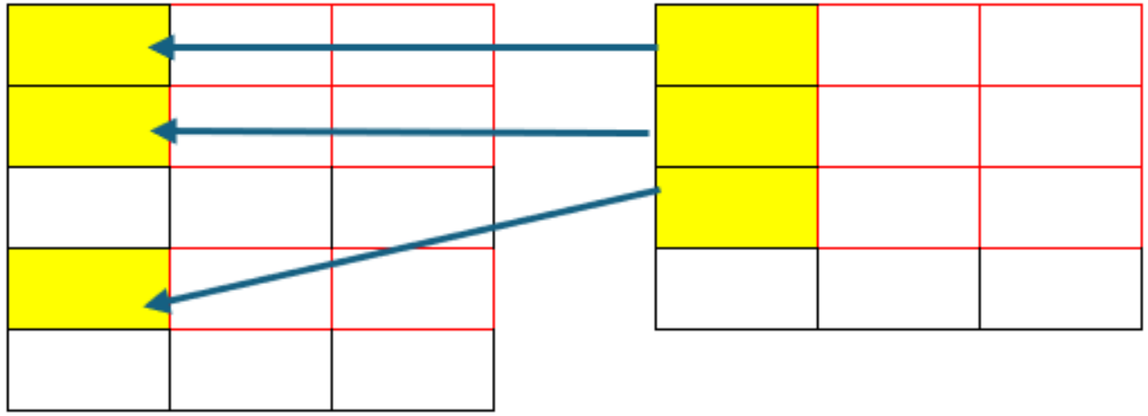
Aspect	dplyr::inner_join	data.table [i, on=]
Conceptual Model	SQL-like (declarative)	Hash / index-based (imperative)
Implementation Algorithm	Sort-based (merge join)	Hash join or key-based binary search
Execution Environment	R + C++ (vctrs, dplyr)	C (datatable.c)
Key Features	Readability, strict type consistency, tidyverse integration	High speed, low memory usage, supports non-equi joins
Database Integration	Translated to SQL via dbplyr	In-memory only (no direct SQL translation)

PharmaForest (SAS)

https://pharmaforest.github.io/

Clinical Trial Data Handling

In clinical trial data programming, particularly in the creation of CDISC-compliant SDTM and ADaM datasets, left joins account for the vast majority of data merge operations.Moreover, in ADaM dataset derivations, it is extremely common to create flag variables (e.g., “Y/N”) indicating whether a given key variable such as USUBJID exists in another dataset.



AE,MH,CM,DS,etc..

USUBJID	
A001	Y
A002	N
A003	Y
B001	Y
B002	N

USUBJID	
A001	
A001	
A003	
B001	
B004	

PharmaForest

https://github.com/PharmaForest

sashash

The sashash package provides powerful and efficient hash-based lookup and validation tools specifically designed for SAS programming. Leveraging the robust capabilities of SAS hash objects, this package enables rapid and dynamic key-based data retrieval and existence checking directly within a single data step. This significantly reduces the need for separate sort and merge steps, streamlining workflows and enhancing performance.

%kvlookup()

Enables efficient and dynamic retrieval of variables from a specified master dataset based on provided keys, directly within a single data step without separate sorting or merging.

PARAMETERS:

```
master     : (Required) Name of the maste
key        : (Required) Space-separated l
var        : (Required) Space-separated l
wh         : (Optional) SQL WHERE clause
warn       : (Optional) Y/N flag. If 'Y',
dropviewflg: (Optional) Y/N flag. If 'Y',
```

%keycheck()

Dynamically validates the existence of keys within a master dataset directly within a single data step. Ideal for rapid data integrity checks and immediate flagging of key existence or non-existence.

PARAMETERS:

```
master     : (Required) Name of the master dataset to check against.
key        : (Required) Space-separated list of key variables to check.
wh         : (Optional) SQL WHERE clause condition to subset the master dataset before loading into has
fl         : (Required) Name of the output variable indicating existence.
cat        : (Optional) Controls output format of existence indicator:
             - 'YN' (default): Returns 'Y' if key exists, 'N' otherwise.
             - 'NUM': Returns 1 if key exists, 0 otherwise.
dropviewflg: (Optional) Y/N flag. If 'Y', drops temporary SQL view created when the 'wh' parameter is u
```

%kvlookup - Sashash Package

```
data dm;
length SUBJID SEX $20.;
SUBJID="A001";SEX="MALE";AGE=14;output;
SUBJID="A002";SEX="FEMALE";AGE=13;output;
SUBJID="B001";SEX="MALE";AGE=13;output;
run;

data wk1;
length SUBJID $20.;
SUBJID="A001";output;
SUBJID="A002";output;
SUBJID="A003";output;
SUBJID="B001";output;
run;

data out1;
set wk1;
%kvlookup(master=dm,key=SUBJID,var=SEX AGE);
run;
```

VIEWTABLE: Work.Dm

	SUBJID	SEX	AGE
1	A001	MALE	14
2	A002	FEMALE	13
3	B001	MALE	13

VIEWTABLE: Work.Wk1

	SUBJID
1	A001
2	A002
3	A003
4	B001

VIEWTABLE: Work.Out1

	SUBJID	SEX	AGE
1	A001	MALE	14
2	A002	FEMALE	13
3	A003		.
4	B001	MALE	13

Kvlookup_dt [R]

```
kvlookup_dt <- function(x,
                        master,
                        key,            # character vector: key column names
                        var = NULL,     # character vector: variable names to retrieve (NULL = all columns except keys)
                        wh  = NULL,     # character scalar: data.table expression applied to master (e.g., quote(Age > 12) or "Age > 12")
                        warn = FALSE,   # whether to issue a warning when lookup keys are not found
                        overwrite = TRUE # whether to overwrite existing columns in x if names overlap
) {
  stopifnot(requireNamespace("data.table", quietly = TRUE))
  DTx <- data.table::as.data.table(x)
  DTm <- data.table::as.data.table(master)

  # --- Apply WHERE filter (wh) to master ---
  if (!is.null(wh) && nzchar(as.character(wh))) {
    # Allow both character and expression inputs for evaluation
    if (is.character(wh)) {
      wh_expr <- parse(text = wh)[[1]]
    } else {
      wh_expr <- wh
    }
    DTm <- DTm[eval(wh_expr)]
  }

  # --- Determine columns to keep (key + var) ---
  if (is.null(var)) {
    # If var is not specified, use all columns in master except keys
    var <- setdiff(names(DTm), key)
  }
  keep_cols <- unique(c(key, var))
  miss_cols <- setdiff(keep_cols, names(DTm))
  if (length(miss_cols)) {
    stop(sprintf("The following columns do not exist in master: %s", paste(miss_cols, collapse = ", ")))
  }
  DTm <- DTm[, ..keep_cols]

  # --- Add a matching flag to master (used to check hits after join) ---
  hit_col <- "__kv_hit__"
  while (hit_col %in% names(DTm)) hit_col <- paste0(hit_col, "_")
  DTm[, (hit_col) := TRUE]

  # --- Set key for faster join ---
  data.table::setkeyv(DTm, key)

  # --- Perform join (X[i] form; keep i = x to preserve its order and row count) ---
  # master[x, on=key] → keeps the number and order of i (=x) rows,
  # with X (=master) columns first
  RES <- DTm[DTx, on = key]

  # --- Extract target columns (var) and merge them into x ---
  # Handle name conflicts
  dup_in_x <- intersect(var, names(DTx))
  if (length(dup_in_x) && !overwrite) {
    stop(sprintf("The following columns already exist in x (overwrite=FALSE): %s",
                 paste(dup_in_x, collapse = ", ")))
  }
  # Add new columns or overwrite existing ones (similar to SAS DATA step behavior)
  for (vn in var) {
    DTx[, (vn) := RES[[vn]]]
  }

  # --- Issue warnings (warn=TRUE when keys are non-NA but not found) ---
  if (isTRUE(warn)) {
    # "matched" = whether the master-side flag is TRUE
    matched <- isTRUE(RES[[hit_col]])
    matched[is.na(matched)] <- FALSE

    # "key_any_present" = at least one key is non-missing
    key_df <- RES[, ..key]
    key_any_present <- apply(!is.na(as.data.frame(key_df)), 1, any)

    not_found <- (!matched) & key_any_present
    if (any(not_found)) {
      # Log the missing key values
      msg_keys <- apply(as.data.frame(key_df[not_found]), 1, function(rw) {
        paste(sprintf("%s=%s", names(key_df), ifelse(is.na(rw),
"NA", as.character(rw))), collapse = ", ")
      })
      warning(sprintf("kvlookup_dt: %d key(s) not found in master. Examples:¥n%s",
                      sum(not_found),
                      paste(utils::head(msg_keys, 5), collapse = "¥n")))
    }
  }

  # --- Remove temporary working column ---
  RES[, (hit_col) := NULL]

  # The result is a data.table where var columns are added to x
  return(DTx[])
}
```

Using data.table, I reproduced the behavior of SAS's PharmaForest.sashash Package. kvlookup macro — one of the most commonly used data-handling techniques in clinical trial programming — with equivalent functionality in R.

Kvlookup_dt [R]

```
dm <- data.table(SUBJID =
c("A001","A002","B001"),
AGE = c(14,13,13),
SEX = c("MALE","FEMALE","FEMALE"))

wk1 <- data.table(SUBJID =
c("A001","A002","A003","B001"))


out <- kvlookup_dt(
wk1, dm,
key = "SUBJID",
var = c("AGE","SEX")
)
```

	SUBJID	AGE	SEX
1	A001	14	MALE
2	A002	13	FEMALE
3	B001	13	FEMALE

	SUBJID
1	A001
2	A002
3	A003
4	B001

	SUBJID	AGE	SEX
1	A001	14	MALE
2	A002	13	FEMALE
3	A003	*NA*	*NA*
4	B001	13	FEMALE

Keycheck_dt [R]

It was also quite straightforward to reproduce the behavior of SAS's %keycheck() macro using data.table in R.

```
keycheck_dt <- function(x,
                        master,
                        key,            # character vector: key column names
                        wh  = NULL,     # character or expression: filter applied to master (e.g., "AGE >= 15")
                        fl  = "exist_fl", # name of output flag column
                        cat = c("YN","NUM","Y")  # "YN"=Y/N, "NUM"=1/0, "Y"=Y/""
) {
  stopifnot(requireNamespace("data.table", quietly = TRUE))
  cat <- match.arg(cat)
  DTx <- data.table::as.data.table(x)
  DTm <- data.table::as.data.table(master)
  # --- apply WHERE filter (wh) on master ---
  if (!is.null(wh) && nzchar(as.character(wh))) {
    wh_expr <- if (is.character(wh)) parse(text = wh)[[1]] else wh
    DTm <- DTm[eval(wh_expr)]
  }
  # --- keep only key columns, unique keys are enough for existence check ---
  miss_cols <- setdiff(key, names(DTm))
  if (length(miss_cols)) {
    stop(sprintf("Columns not found in master: %s", paste(miss_cols, collapse = ", ")))
  }
  DTm <- unique(DTm[, ..key])
  # --- prepare master with a hit flag for join result ---
  hit_col <- "__key_hit__"
  while (hit_col %in% names(DTm)) hit_col <- paste0(hit_col, "_")
  DTm[, (hit_col) := TRUE]
  # --- set keys for fast join ---
  data.table::setkeyv(DTm, key)
  # --- left-join-like existence check against x (preserve x's order/rows) ---
  RES <- DTm[DTx, on = key]
  # FIX: vectorized match flag
  matched <- RES[[hit_col]]          # logical vector: TRUE or NA
  matched[is.na(matched)] <- FALSE     # NA -> FALSE
  # --- build the requested flag column on DTx ---
  if (cat == "YN") {
    DTx[, (fl) := ifelse(matched, "Y", "N")]
  } else if (cat == "NUM") {
    DTx[, (fl) := as.integer(matched)]  # 1 if exists, 0 otherwise
  } else if (cat == "Y") {
    DTx[, (fl) := ifelse(matched, "Y", "")]
  }

  invisible(DTx[])
}
```

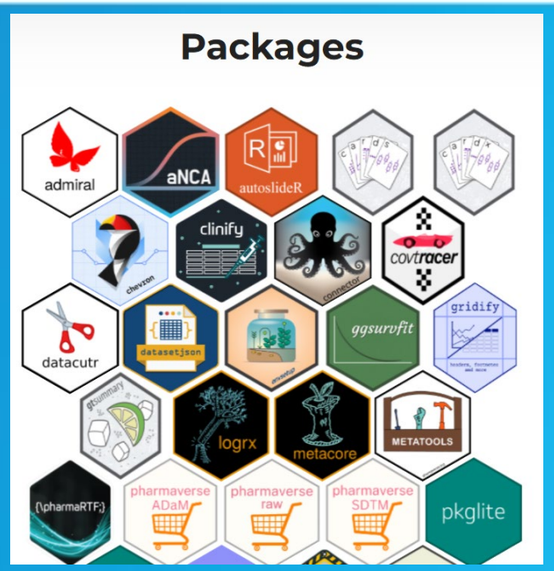
```
ae <- data.table(USUBJID = c("A001","A001","A003"),
AETERM = c("AE 1", "AE 2", "AE 1"))
dm <- data.table(USUBJID = c("A001","A002","A003","A004"))
out1 <- keycheck_dt(dm, ae, key="USUBJID", fl="AEFL", cat="YN")
```

	USUBJID
1	A001
2	A002
3	A003
4	A004

	USUBJID	AETERM
1	A001	AE 1
2	A001	AE 2
3	A003	AE 1

	USUBJID	AEFL
1	A001	Y
2	A002	N
3	A003	Y
4	A004	N

Pharmaverse



https://pharmaverse.org/

Representative join-related functions in the {admiral} package

phuse Home GitHub YouTube Blog E›

ADaM in R Asset Library

Purpose

To provide an open source, modularized toolbox that enables the pharmaceutical programming community to develop ADaM datasets in R.

Representative join-related functions in the {admiral} package

derive_vars_merged()

```
library(admiral)
adae <- tibble(USUBJID = c("A001","A001","A003"),
AETERM = c("AE 1", "AE 2", "AE 1"))
```

	USUBJID	AETERM
1	A001	AE 1
2	A001	AE 2
3	A003	AE 1

```
adsl <- tibble(
USUBJID = c("A001","A002","A003","A004"),
TRTSDT = as.IDate(c("2024-01-10","2024-01-12","2024-01-15","2024-01-18"))
)
```

	USUBJID	TRTSDT
1	A001	2024-01-10
2	A002	2024-01-12
3	A003	2024-01-15
4	A004	2024-01-18

```
adae_1 <- derive_vars_merged(
dataset = adae,
dataset_add = adsl,
by_vars = exprs(USUBJID),
new_vars = exprs(TRTSDT)
)
```

	USUBJID	AETERM	TRTSDT
1	A001	AE 1	2024-01-10
2	A001	AE 2	2024-01-10
3	A003	AE 1	2024-01-15

Representative join-related functions in the {admiral} package

derive_var_merged_exist_flag()

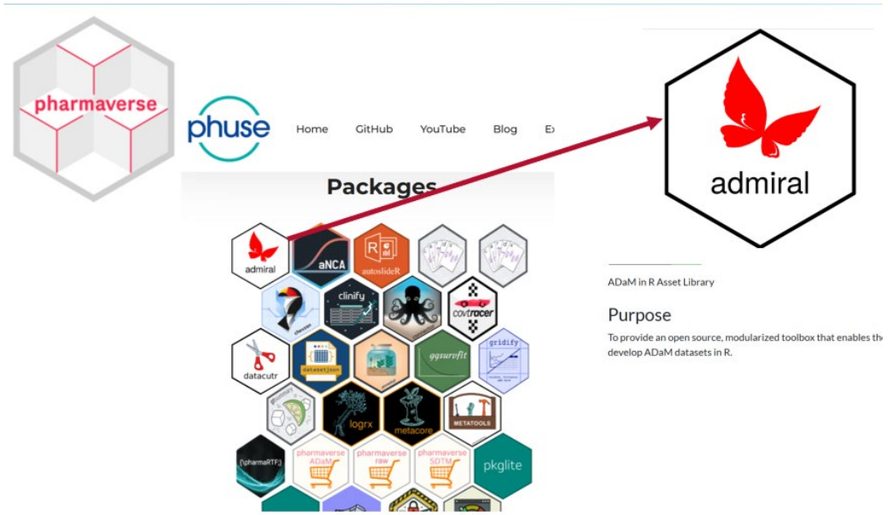
```
adsl_1 <- derive_var_merged_exist_flag(
dataset = adsl,
dataset_add = adae,
by_vars = exprs(USUBJID),
condition = TRUE,
new_var = AEFL,
true_value = "Y",
false_value = "N" ,
missing_value= "N"
)
```

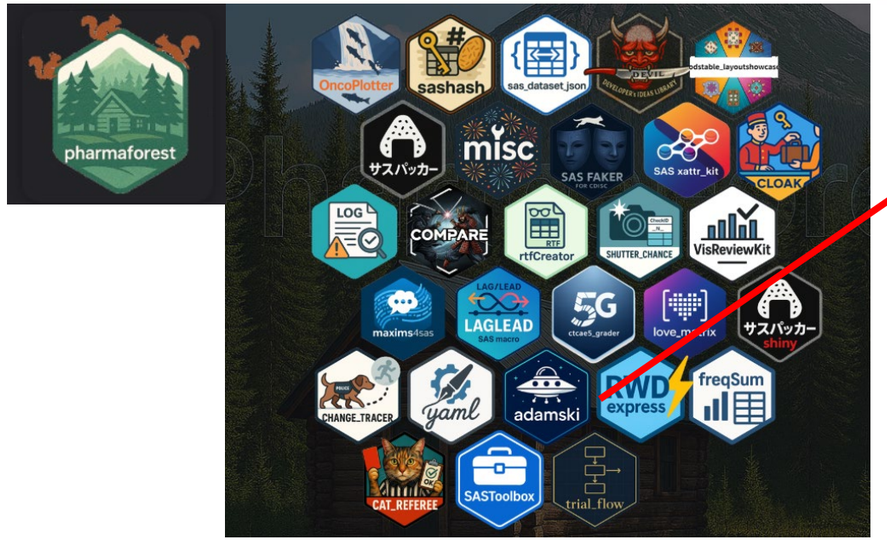
	USUBJID	TRTSDT
1	A001	2024-01-10
2	A002	2024-01-12
3	A003	2024-01-15
4	A004	2024-01-18

	USUBJID	AETERM
1	A001	AE 1
2	A001	AE 2
3	A003	AE 1

	USUBJID	TRTSDT	AEFL
1	A001	2024-01-10	Y
2	A002	2024-01-12	N
3	A003	2024-01-15	Y
4	A004	2024-01-18	N

Pharmaforest {Adamski} package





%derive_var_merged_exist_flag()

Purpose:

Creates a character flag variable indicating whether the current DATA step row's key(s) exist in another dataset.

Parameters:

```
- `dataset_add` (required) : Dataset to check for existence (e.g., `SDTM.AE`).
- `by_vars`     (required) : Space-separated list of key variables used for the lookup.
- `new_var`     (required) : Name of the output flag variable to create (character).
- `condition`   (optional) : WHERE clause (as text) applied to `dataset_add` before building  the hash
- `true_value`  (optional) : Value assigned to `new_var` when a match is found.
                             Default: `Y`.
- `false_value` (optional) : Value assigned to `new_var` when no match is found.
                             Default: (blank).
```

In the current PharmaForest adamski package project,we are gradually working on reproducing the functionality of pharmaverse's admiral in SAS,inspired by the concepts and design of admiral.

sdtm.oak

phuse Home GitHub YouTube Blog Exa

sdtm.oak

An EDC (Electronic Data Capture systems) and Data Standard agnostic solution that enables the pharmaceutical programming community to develop CDISC (Clinical Data Interchange Standards Consortium) SDTM (Study Data Tabulation Model) datasets in R. The reusable algorithms concept in 'sdtm.oak' provides a framework for modular programming. We plan to develop a code generation feature based on a standardized SDTM specification format, which has the potential to automate the creation of SDTM datasets.

Final Comparison and Summary

Summary Notes

Common but Subtle Differences in Join Behavior Across R Frameworks

Aspect	base::merge	dplyr::left_join	data.table X[i, on=]	hash package
Default Sorting	Yes (sort=TRUE)	No (preserves left table order)	Preserves order of *i* (right operand)	Not applicable
Non-equi Join	Not supported	Not supported	Supported (on=.(a <= b, ...))	Not applicable
Rolling / Nearest Join	Not supported	Not supported	Supported (roll=TRUE / mult)	Not applicable
Column Name Conflicts	Managed via suffixes	Appends .x / .y	Explicit reference with i. (overwrite using :=)	Not applicable
Many-to-Many Handling	Expands all combinations	Expands all combinations	Expands all combinations (requires unique() / aggregation if undesired)	Assumes one-to-one mapping
Performance on Large Data	△ Slow	△–○ Moderate (can leverage databases via dbplyr)	◎ Excellent (in-memory optimized)	○ Good (limited by single-key design)

Summary Notes

- base::merge() is simple and reliable but can reorder rows and is memory-heavy.
- dplyr::xxxx_join() emphasizes readability and consistency, suitable for tidy workflows and moderate datasets.
- data.table joins are the most efficient and flexible, supporting non-equi and rolling logic ideal for clinical data (e.g., SDTM/ADaM period merges).
- The hash package provides ultra-fast one-to-one lookups, best for code mapping or format-like replacements.
- In CDISC workflows, it is also recommended to use the built-in join functions provided by Admiral or oak, as they help standardize and simplify data merge management across ADaM and SDTM process

新手一生

Kōzō Masuda（升田 幸三）
Japanese Professional Shogi Player

The above kanji is a motto of a professional Shogi (Japanese chess) player,meaning
“Always try a new move.”

I’ve long been passionate about pioneering new techniques in SAS, and now, I’m ready to do the same with R.

イーピーエス株式会社