Fred Vellinga BI Services

SAS Merge SQL Equivalent Object Model

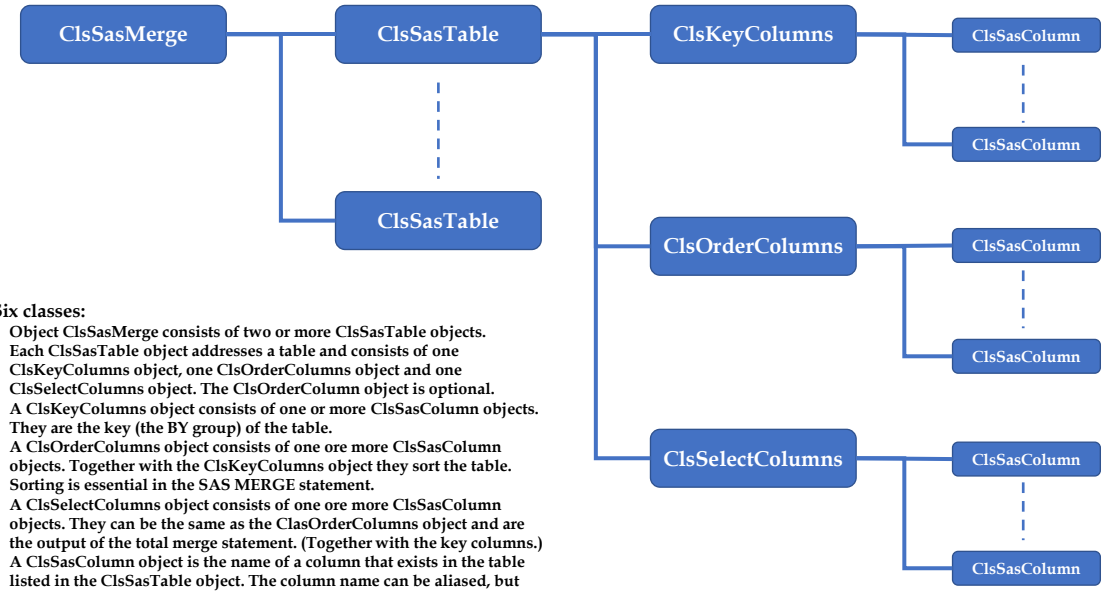
De macro Ctr+L maakt een streepjes lijst.
De macro Ctr+G maakt een grid/tabel.

1 SAS Merge Equivalent Object Model SQL 3
1.1 Classes and Objects 4
2 Class ClsSasColumn 5
3 Class ClsKeyColumns 6
4 Class ClsOrderColumns 7
5 Class ClsSelectColumns 8
6 Class ClsSasTable 9
7 Class ClsSasMerge 10
8 Appendix A (Additional processing) 12
9 Appendix B (Code) 13
9.1 SAS test data set 13
9.2 PostgreSQL test dataset 15

1 SAS Merge Equivalent Object Model SQL

The SAS Merge SQL Equivalent Object model simulates the SAS **MERGE** statement using SQL. It generates one SQL statement that is equivalent to the SAS merge behaviour. The object model consists of six classes written in Python with Visual Studio 2017 and tested in PostgreSQL. But it is standard SQL so should work everywhere. The model looks as follows:

SAS Merge SQL Equivalent Object Model



Six classes:

- **Object ClsSasMerge consists of two or more ClsSasTable objects.**
- **Each ClsSasTable object addresses a table and consists of one ClsKeyColumns object, one ClsOrderColumns object and one ClsSelectColumns object. The ClsOrderColumn object is optional.**
- **A ClsKeyColumns object consists of one or more ClsSasColumn objects. They are the key (the BY group) of the table.**
- **A ClsOrderColumns object consists of one ore more ClsSasColumn objects. Together with the ClsKeyColumns object they sort the table. Sorting is essential in the SAS MERGE statement.**
- **A ClsSelectColumns object consists of one ore more ClsSasColumn objects. They can be the same as the ClasOrderColumns object and are the output of the total merge statement. (Together with the key columns.)**
- **A ClsSasColumn object is the name of a column that exists in the table listed in the ClsSasTable object. The column name can be aliased, but this makes only sense for the ClsSelectColumns object.**

When you feed the returning SQL statement to a SQL data provider you can then iterate through the rows to do something with the data and you can fully emulate, or come close, the SAS data step technique.

The SAS overlay aspect is **not** implemented; column names with the same names are not overwritten. Also the **MERGE** equivalent without a **BY** statement is **not** supported. Also the position of a column in the **SELECT** clause is rather fixed.

SAS overlay means that same name columns are over written from right to left. The most right table in the **MERGE** statements overwrites columns in the left table having the same name. This means the order of appearance in the SAS **MERGE** statement is relevant, just as the ordering of a table.

When you do not want to iterate through the result data set, can also use the returning SQL query as in inner query:

```
SELECT Tx.<what_ever>
FROM (
  SELECT <sas_merge_query> ) Tx
WHERE <whatever>
```

Or more flexible, use the **WITH** clause:

```
WITH Tx as (
  <sas_merge_query>
),
<any other query>
SELECT Tx.<whatever>
FROM Tx
WHERE <whatever>
```

Supporting pointers per table (the **IN** option) are available telling you when there was a match or not. The **WHERE** clause can be used to keep only the rows you want. The **WITH** clause gives you the most power to further manipulate the data.

The SQL equivalent has no limits. The limits are the data provider and target database.

1.1 Classes and Objects

An object is an instance of class. The class is the code, the object is the instance in memory. You can have unlimited instances, or objects, of the same class in memory, all containing different information. Regard an object as a pointer variable pointing to a memory location.

Some objects in this model are collections of other objects. You can see that as parent-child relations. Because all child objects are referenced by value to the parent object you can re-use an object as many times as you like. Any parent object never references to the same child object, even when all child objects originated from one object. All objects are unique.

A class consists of a constructor (initializes the object variable), properties (variables local to the class which can be set and read), methods (the functions in a class) and events, also referred to as a call-back function. Not all of these elements are necessary in a class. Only the constructor is mandatory.

2 Class ClsSasColumn

The *SasColumn* object defines a column name you want to use in your **MERGE** statement. A column can be used in the **BY** part (in this model called the key), in the **ORDER** part of the table (table sorting is essential) and in the output part.

When next is your **MERGE** statement:

```
DATA Tmerge (keep=col1 col2 col3 d1 d2 d3);
  MERGE Ta(in=A) Tb(in=B) Tc(in=C)
  BY col1 col2 col3
  <body>
RUN;
```

then you have to be sure the three tables are sorted as you wanted. The minimal sort is by *col1-col3*, but there might be more. And you must know at forehand what your output columns will be. The SQL equivalent does the sorting for you, and the rest, but you have to tell the model which columns you will take from the tables for whatever purpose; key, sort or output. For that you use the *SasColumn* class.

Constructor	**pName**	The column name as it exists in a table you are using.
	pAlias	The column alias name as you want the column be named in the output. Only relevant for the output part.
	(The constructor parameters are optional.)	
Properties	**p_error_msg**	Returns the error message generated by this class.
	p_name	Set or returns the column name as it exists in a table you are using.
	p_alias	Set or returns the column alias name as you want the column be named in the output. Only relevant for the output part.
Methods		This class has no methods.
Events	**Event_handler**	This event is fired when you enter wrong data types in the property values when setting them. It is a sort of debugging feature. Once fired it executes the call back function specified here.

Example (without events):

```
col        = ClsSasColumn()
col.p_name = "k1"
```

3 Class ClsKeyColumns

The *KeyColumn* object contains a collection of *SasColumn* objects and forms the **BY** part of the **MERGE** statement for a given table that is part in the **MERGE** statement. Because this is SQL, there is no need that the key or **BY** variable names in the tables must have the same name.
Because this class is a collection class, in Python called a sequence class, the class has more elements then the *SasColumn* class. This is needed to iterate through the collection. Here I list only the elements necessary to turn on the SQL functionality.

Constructor	**pCol**	A *SasColumn* object. It is a pointer variable. But that is transparent. Fastest method to add one column to this object.
	(The constructor parameters are optional.)	
Properties	**p_error_msg**	Returns the error message generated by this class.
	p_columns	Returns the collection (a list) of *SasColumn* objects.
	p_index	Returns the index of the *SasColumn* object in the collection.
	p_count	Returns the column count; the number of *SasColumn* objects in this object.
Methods	**Append(Obj)**	Append a *SasColumn* object to this collection. The append order is important. When you want to do **BY** processing using **BY** ***k1 k2***, you first have to add ***k1*** and then ***k2***. The other way will also work but then you may get unintended output. When you initialize this object via the constructor, you can define a *SasColumn* also there. That column is then appended. When you need only one column for this class, that is the fastest method.
	Remove(Obj)	Removes the *SasColumn* object from this collection.
Events	**Event_handler**	This event is fired when you enter wrong data types in the method parameters. It is a sort of debugging feature. Once fired it executes the call back function specified here.

Example (without events):

```
col      = ClsSasColumn()
keys     = ClsKeyColumns()

col.p_name = "k1"; keys.append(col)
col.p_name = "k2"; keys.append(col)
```

Here you have added two columns (**k1** and **k2**) to the *KeyColumn* object. You have a collection of two *SasColumn* objects. The columns originate from the same *SasColumn* object but are in the *KeyColumn* object unique. (Python supports the ';' character as statement separator.)

4 Class ClsOrderColumns

The *OrderColumn* object contains a collection of *SasColumn* objects that are columns needed to additional sort the table. It is an optional object. Together with the *KeyColumn* object the *OrderColumn* object orders the table. So you can regard them as additional to the key columns. The class is identical to class *ClassKeyColumns*.

Constructor	**pCol**	A *SasColumn* object. It is a pointer variable. But that is transparent. Fastest method to add one column to this object.
	(The constructor parameters are optional.)	
Properties	**p_error_msg**	Returns the error message generated by this class.
	p_columns	Returns the collection (a list) of *SasColumn* objects.
	p_index	Returns the index of the *SasColumn* object in the collection.
	p_count	Returns the column count; the number of *SasColumn* objects in this object.
Methods	**Append(Obj)**	Append a *SasColumn* object to this collection. The append order is important. When you want to do additional column ordering with ***d1***, ***d2***, you first have to add ***d1*** and then ***d2***. The other way will also work but then you may get unintended output.
	Remove(Obj)	Removes the *SasColumn* object from this collection.
Events	**Event_handler**	This event is fired when you enter wrong data types in the method parameters. It is a sort of debugging feature. Once fired it executes the call back function specified here.

Example (without events):

```
col   = ClsSasColumn()
keys  = ClsKeyColumns()
order = ClsOrderColumns()

col.p_name = "k1"; keys.append(col)
col.p_name = "k2"; keys.append(col)
col.p_name = 'd1'; order.append(col)
col.p_name = 'd2'; order.append(col)
```

Here you have added two columns (**k1** and **k2**) to the *KeyColumn* object and two columns (**d1** and **d2**) to the *OrderColumn* object. You now have two collections each having two *SasColumn* objects. The columns originate from the same *SasColumn* object but are unique in both collections.

5 Class ClsSelectColumns

The *SelectColumn* object contains a collection of *SasColumn* objects that are columns that will be outputted. The class is identical to class *ClassKeyColumns*.

Constructor	**pCol**	A *SasColumn* object. It is a pointer variable. But that is transparent. Fastest method to add one column to this object.
	(The constructor parameters are optional.)	
Properties	**p_error_msg**	Returns the error message generated by this class.
	p_columns	Returns the collection (a list) of *SasColumn* objects.
	p_index	Returns the index of the *SasColumn* object in the collection.
	p_count	Returns the column count; the number of *SasColumn* objects in this object.
Methods	**Append(Obj)**	Append a *SasColumn* object to this collection. The append order is important. When you do additional column ordering with ***d1***, ***d2***, you first have to add ***d1*** and then ***d2***. The other way will also work but then you may get unintended output.
	Remove(Obj)	Removes the *SasColumn* object from this collection.
Events	**Event_handler**	This event is fired when you enter wrong data types in the method parameters. It is a sort of debugging feature. Once fired it executes the call back function specified here.

Example (whithout events):

```
col    = ClsSasColumn()
keys   = ClsKeyColumns()
order  = ClsOrderColumns()
select = ClsSelectColumns()

col.p_name = "k1"; keys.append(col)
col.p_name = "k2"; keys.append(col)
col.p_name = 'd1'; order.append(col); select.append(col)
col.p_name = 'd2'; order.append(col); select.append(col)
```

Here you have added two columns (**k1** and **k2**) to the *KeyColumn* object, two columns (**d1** and **d2**) to the *OrderColumn* object and two columns (**d1** and **d2**) to the *SelectColumn* object. The *SelectColumn* object contains the same columns as the *OrderColumn* object. You now have three collections each having two *SasColumn* objects. The columns originate from the same *SasColumn* object but are unique in all three collections.

6 Class ClsSasTable

The *SasTable* object defines a table with its key columns, order columns (optional) and select columns. All what you do is assigning previously defined objects to this object.

Constructor	**pName**	The table name.
	pKeys	A *KeyColumn* object. It is a pointer variable. But that is transparent. Fastest method to add the *KeyColumn* object to this object.
	POrder	A *OrderColumn* object. It is a pointer variable. But that is transparent. Fastest method to add the *OrderColumn* object to this object.
	pSelect	A *SelectColumn* object. It is a pointer variable. But that is transparent. Fastest method to add the *SelectColumn* object to this object.
	(The constructor parameters are optional.)	
Properties	**p_error_msg**	Returns the error message generated by this class.
	p_keys	Set or returns the *KeyColumn* object. It is a pointer variable.
	p_order	Set or returns the *OrderColumn* object. It is a pointer variable.
	p_select	Set or returns the *SelectColumn* object. It is a pointer variable.
	(Using the properties you can also change the values.)	
Methods		This class has no methods.
Events	**Event_handler**	This event is fired when you enter wrong data types in the method parameters or property values. It is a sort of debugging feature. Once fired it executes the call back function specified here.

Example (with events):

```
def consume_class_event(pCls):
  """ The callback function """
  if pCls.p_error_msg != None:
    print(pCls.p_error_msg)

col = ClsSasColumn()
e.sas_col_event += consume_class_event(col)
col.event_handler(consume_class_event)

keys = ClsKeyColumns()
e.sas_key_event += consume_class_event(keys)
keys.event_handler(consume_class_event)

order = ClsOrderColumns()
e.sas_order_event += consume_class_event(order)
order.event_handler(consume_class_event)

select = ClsSelectColumns()
e.sas_select_event += consume_class_event(select)
select.event_handler(consume_class_event)

table = ClsSasTable()
e.sas_table_event += consume_class_event(table)
table.event_handler(consume_class_event)

merge = ClsSasMerge()
e.sas_merge_event += consume_class_event(merge)
merge.event_handler(consume_class_event)

col.p_name = "k1"; keys.append(col)
col.p_name = "k2"; keys.append(col)
col.p_name = 'd1'; order.append(col); select.append(col)
col.p_name = 'd2'; order.append(col); select.append(col)
table.p_name = "Ta"
table.p_keys = keys
table.p_order = order
table.p_select = select
merge.append(table)
```

Here you have specified one table, ready to get assigned to the *SasMerge* object.

7 Class ClsSasMerge

The *SasMerge* object defines all tables involved in the merge. This class returns the SQL statement. The SQL statement can be generated in a sorted and non-sorted variant. The sorted variant might be handy when you want to iterate through all the rows. But when you want to do additional **`where`** filtering you have to turn the sort order off.

Default the SQL MERGE statement always output the key parameters, using the names as listed by the first table, the individual sort pointers of the table, and the overall sort pointer of the SQL MERGE statement.

`Constructor`	**`pTable`**	A *SasTable* object. It is a pointer variable. But that is transparent. Fastest method to add one table to this object.
	(The constructor parameters are optional.)	
`Properties`	**`p_error_msg`**	Returns the error message generated by this class.
	`p_tables`	Returns the collection (a list) of *SasTable* objects.
	`p_index`	Returns the index of the *SasTable* object in the collection.
	`p_count`	Returns the column count; the number of *SasTable* objects in this object.
	`P_orderby`	Set or returns the flag that tells if the query must be returned sorted or not. It adds **`ORDER BY Tm.p`** to the query. **`Tm.p`** is the alias name for the overall sorting pointer of the table.
	`p_slq_merge`	Return the SQL MERGE statement. First the method *get_sql_merge()* must have been invoked.
`Methods`	**`Append(Obj)`**	Append a *SasTable* object to this collection. The append order is important.
	`Remove(Obj)`	Removes the *SasTable* object from this collection.
	`get_sql_merge()`	return the SQL MERGE string.
`Events`	**`Event_handler`**	This event is fired when you enter wrong data types in the method parameters. It is a sort of debugging feature. Once fired it executes the call back function specified here.

A complete example (with events) that gives three different methods of defining a table.

```
col = ClsSasColumn()
e.sas_col_event += consume_class_event(col)
col.event_handler(consume_class_event)

keys = ClsKeyColumns()
e.sas_key_event += consume_class_event(keys)
keys.event_handler(consume_class_event)

order = ClsOrderColumns()
e.sas_order_event += consume_class_event(order)
order.event_handler(consume_class_event)

select = ClsSelectColumns()
e.sas_select_event += consume_class_event(select)
select.event_handler(consume_class_event)

table = ClsSasTable()
e.sas_table_event += consume_class_event(table)
table.event_handler(consume_class_event)

merge = ClsSasMerge()
e.sas_merge_event += consume_class_event(merge)
merge.event_handler(consume_class_event)

# First table
col.p_name = "k1"; keys.append(col)
col.p_name = "k2"; keys.append(col)
col.p_name = 'd1'; order.append(col); select.append(col)
table.p_name = "Ta"
table.p_keys = keys
table.p_order = order
table.p_select = select
```

```
merge.append(table)

# Second table
table.p_name = "Tb"
table.p_keys[0].p_name = "k3"
table.p_keys[1].p_name = "k4"
table.p_order[0].p_name = "d2"
table.p_select[0].p_name = "d2"
merge.append(table)

# Third table
table  = ClsSasTable()
keys   = ClsKeyColumns()
order  = ClsOrderColumns()
select = ClsSelectColumns()
col = ClsSasColumn(pName="k5")
keys.append(col)
col = ClsSasColumn(pName="k6")
keys.append(col)
col = ClsSasColumn(pName="d3")
order.append(col)
col = ClsSasColumn(pName="d3")
select.append(col)
table.p_name = "Tc"
table.p_keys = keys
table.p_order = order
table.p_select = select
merge.append(table)

sql = merge.get_sql_merge()          # make the SQL statement
print(sql)
```

8 Appendix A (Additional processing)

Assume you merge four tables, as listed by the schema below:

Table	Ta	Tb	Tc	Tc
Key	k1, k2, k3	ka, kb, kc	x, y, z	col1, col2, col3
Order	d3	d_8, d_9	d1	d5
Select	d1, d2, d3	d1, d2, d3	d1, d2, d3, d4	d1
Table alias	T0	T1	T2	T3

The **SELECT** clause is then as follows:

```
SELECT Tm.p, Tm.k1, Tm.k2, Tm.k3,
       Tm.p_0, Tm.p_1, Tm.p_2, Tm.p_3,
       T0.d1, T0.d2, T0.d3,
       T1.d1, T1.d2, T1.d3,
       T2.d1, T2.d2, T2.d3, T2.d4,
       T3.d1
```

Aliasing of the involved tables is done automatically. Aliasing of the select columns must be defined in the model. The order columns play an important role inside the query, but are not outputted as long as you do not select them.

If you want only output rows coming from **Ta** and **Tb** you have to add the following **where** clause to the statement:

```
where Tm.p_0 is null
  and Tm.p_1 is not null
  and Tm.p_2 is not null
  and Tm.p_3 is null
```

When you want do additional processing in the body you have to use the query as an inner query. But as listed in the introduction, more flexible is to do this:

```
WITH Tx as (
  <sas_merge_query>
)
SELECT Tx.<whatever>
FROM Tx
WHERE <whatever>
```

The typical SAS behaviour that you can output the same row multiple times with variations in the data can be done as follows:

```
WITH Tx as (
  <sas_merge_query>
)
SELECT Tx.<whatever>
FROM Tx
WHERE <whatever>
UNION ALL
SELECT Tx.<whatever>
FROM Tx
WHERE <whatever>
UNION ALL
SELECT Tx.<whatever>
FROM Tx
WHERE <whatever>
```

9 Appendix B (Code)

The code is packed in ZIP file SasMergePython.zip and contains three modules. The Python code is written in Visual Studio 2017 and tested on Windows 10.

Modulename	Description	Dependencies
`SasMergeModule.py`	Contains the six classes as described here.	`None`
`ClsPg.py`	PostgreSQL connection class. I use version 9.4 In my case I have stored the login credentials in Windows Environment variables.	`psycopg2` `events`
`main.py`	The test code. The startup file in Visual Studio.	`events`

9.1 SAS test data set

```
*% Unique keys (k1/k2/k3) in all three tables;
*% Take k1 only for non unique key;
data Ta;
  infile datalines;
  input k1 k2 k3 da1 $ da2 $ da3 $;
  datalines;
1 2 1 r1_a_1 r1_a_2 r1_a_3
1 2 2 r2_a_1 r2_a_2 r2_a_3
1 2 3 r3_a_1 r3_a_2 r3_a_3
2 3 4 r4_a_1 r4_a_2 r4_a_3
2 3 5 r5_a_1 r5_a_2 r5_a_3
;
run;

data Tb;
  infile datalines;
  input k1 k2 k3 db1 $ db2 $ db3 $;
  datalines;
1 2 1 r1_b_1 r1_b_2 r1_b_3
1 2 3 r2_b_1 r2_b_2 r2_b_3
2 3 4 r3_b_1 r3_b_2 r3_b_3
3 1 5 r4_b_1 r4_b_2 r4_b_3
4 2 6 r5_b_1 r5_b_2 r5_b_3
4 7 7 r6_b_1 r6_b_2 r6_b_3
;
run;

data Tc;
  infile datalines;
  input k1 k2 k3 dc1 $ dc2 $ dc3 $;
  datalines;
0 2 1 r0_c_1 r0_c_2 r0_c_3
1 2 1 r1_c_1 r1_c_2 r1_c_3
1 2 2 r2_c_1 r2_c_2 r2_c_3
4 7 7 r3_c_1 r3_c_2 r3_c_3
5 8 8 r4_c_1 r4_c_2 r4_c_3
6 9 9 r5_c_1 r5_c_2 r5_c_3
;
run;

proc sort data=Ta out=Ta;
  by k1 k2 k3 da1;
proc sort data=Tb out=Tb;
  by k1 k2 k3 db1;
proc sort data=Tc out=Tc;
  by k1 k2 k3 dc1;
run;

data Dmerge_1;
  merge Ta (in=p1) Tb (in=p2) Tc (in=p3);
  by k1 k2 k3;
run;

data Dmerge_1A;
  merge Ta (in=p1) Tb (in=p2) Tc (in=p3);
  by k1 k2 k3;
  if p1 = 1 and p2 = 1 and p3 = 1;
run;
```

```
data Dmerge_2;
  merge Ta (in=p1) Tb (in=p2) Tc (in=p3);
  by k1;
run;

data Dmerge_2A;
  merge Ta (in=p1) Tb (in=p2) Tc (in=p3);
  by k1;
  if p1 = 1 and p2 = 1 and p3 = 1;
run;

*% Another case with four tables;
data Ta;
  infile datalines;
  input k1 k2 k3 k4 da1 $;
  datalines;
1 2 3 1 r1_a_1
1 2 3 2 r2_a_1
1 2 3 3 r3_a_1
1 2 3 4 r4_a_1
1 2 3 5 r5_a_1
;
run;

data Tb;
  infile datalines;
  input k1 k2 k3 k4 db1 $;
  datalines;
1 2 1 0 r1_b_1
1 2 3 1 r2_b_1
1 2 4 2 r3_b_1
1 2 5 3 r4_b_1
1 2 6 4 r5_b_1
1 2 7 5 r6_b_1
;

data Tc;
  infile datalines;
  input k1 k2 k3 k4 dc1 $;
  datalines;
0 2 3 1 r0_c_1
0 2 3 2 r1_c_1
1 2 3 3 r2_c_1
1 2 3 4 r3_c_1
1 2 3 5 r4_c_1
1 2 3 6 r5_c_1
;

data Td;
  infile datalines;
  input k1 k2 k3 k4 dd1 $;
  datalines;
1 2 3 1 r1_d_1
1 2 3 2 r2_d_1
1 2 3 3 r3_d_1
1 2 3 4 r4_d_1
1 2 3 5 r5_d_1
1 2 3 6 r6_d_1
1 2 3 7 r7_d_1
1 2 3 8 r8_d_1
1 2 3 9 r9_d_1
1 2 3 10 r10_d_1
;

proc sort data=Ta out=Ta;
  by k1 k2 k3 k4 da1;
proc sort data=Tb out=Tb;
  by k1 k2 k3 k4 db1;
proc sort data=Tc out=Tc;
  by k1 k2 k3 k4 dc1;
proc sort data=Td out=Td;
  by k1 k2 k3 k4 dd1;
run;
```

```
data Dmerge_1;
  merge Ta (in=p1) Tb (in=p2) Tc (in=p3) Td (in=p4);
  by k1 k2 k3 k4;
run;

data Dmerge_2;
  merge Ta (in=p1) Tb (in=p2) Tc (in=p3) Td (in=p4);
  by k1;
run;
```

9.2 PostgreSQL test dataset

```
drop table if exists Ta1;
drop table if exists Tb2;
drop table if exists Tc3;

create table Ta1 (k1 int, k2 int, k3 int, da1 varchar(8), da2 varchar(8), da3 varchar(8));
create table Tb2 (k4 int, k5 int, k6 int, db1 varchar(8), db2 varchar(8), db3 varchar(8));
create table Tc3 (k7 int, k8 int, k9 int, dc1 varchar(8), dc2 varchar(8), dc3 varchar(8));

insert into Ta1 values(1, 2, 1, 'r1_a_1', 'r1_a_2', 'r1_a_3');
insert into Ta1 values(1, 2, 2, 'r2_a_1', 'r2_a_2', 'r2_a_3');
insert into Ta1 values(1, 2, 3, 'r3_a_1', 'r3_a_2', 'r3_a_3');
insert into Ta1 values(2, 3, 4, 'r4_a_1', 'r4_a_2', 'r4_a_3');
insert into Ta1 values(2, 3, 5, 'r5_a_1', 'r5_a_2', 'r5_a_3');

insert into Tb2 values(1, 2, 1, 'r1_b_1', 'r1_b_2', 'r1_b_3');
insert into Tb2 values(1, 2, 3, 'r2_b_1', 'r2_b_2', 'r2_b_3');
insert into Tb2 values(2, 3, 4, 'r3_b_1', 'r3_b_2', 'r3_b_3');
insert into Tb2 values(3, 1, 5, 'r4_b_1', 'r4_b_2', 'r4_b_3');
insert into Tb2 values(4, 2, 6, 'r5_b_1', 'r5_b_2', 'r5_b_3');
insert into Tb2 values(4, 7, 7, 'r6_b_1', 'r6_b_2', 'r6_b_3');

insert into Tc3 values(0, 2, 1, 'r0_c_1', 'r0_c_2', 'r0_c_3');
insert into Tc3 values(1, 2, 1, 'r1_c_1', 'r1_c_2', 'r1_c_3');
insert into Tc3 values(1, 2, 2, 'r2_c_1', 'r2_c_2', 'r2_c_3');
insert into Tc3 values(4, 7, 7, 'r3_c_1', 'r3_c_2', 'r3_c_3');
insert into Tc3 values(5, 8, 8, 'r4_c_1', 'r4_c_2', 'r4_c_3');
insert into Tc3 values(6, 9, 9, 'r5_c_1', 'r5_c_2', 'r5_c_3');

drop table if exists Txa1;
drop table if exists Txb2;
drop table if exists Txc3;
drop table if exists Txd4;

create table Txa1 (k1 int, k2 int, k3 int, k4 int, da1 varchar(8));
create table Txb2 (k1 int, k2 int, k3 int, k4 int, db1 varchar(8));
create table Txc3 (k1 int, k2 int, k3 int, k4 int, dc1 varchar(8));
create table Txd4 (k1 int, k2 int, k3 int, k4 int, dd1 varchar(8));

insert into Txa1 values(1, 2, 3, 1, 'r1_a_1');
insert into Txa1 values(1, 2, 3, 2, 'r2_a_1');
insert into Txa1 values(1, 2, 3, 3, 'r3_a_1');
insert into Txa1 values(1, 2, 3, 4, 'r4_a_1');
insert into Txa1 values(1, 2, 3, 5, 'r5_a_1');

insert into Txb2 values(1, 2, 1, 0, 'r1_b_1');
insert into Txb2 values(1, 2, 3, 1, 'r2_b_1');
insert into Txb2 values(1, 2, 4, 2, 'r3_b_1');
insert into Txb2 values(1, 2, 5, 3, 'r4_b_1');
insert into Txb2 values(1, 2, 6, 4, 'r5_b_1');
insert into Txb2 values(1, 2, 7, 5, 'r6_b_1');

insert into Txc3 values(0, 2, 3, 1, 'r0_c_1');
insert into Txc3 values(0, 2, 3, 2, 'r1_c_1');
insert into Txc3 values(1, 2, 3, 3, 'r2_c_1');
insert into Txc3 values(1, 2, 3, 4, 'r3_c_1');
insert into Txc3 values(1, 2, 3, 5, 'r4_c_1');
insert into Txc3 values(1, 2, 3, 6, 'r5_c_1');

insert into Txd4 values(1, 2, 3, 1, 'r1_d_1');
insert into Txd4 values(1, 2, 3, 2, 'r2_d_1');
insert into Txd4 values(1, 2, 3, 3, 'r3_d_1');
insert into Txd4 values(1, 2, 3, 4, 'r4_d_1');
insert into Txd4 values(1, 2, 3, 5, 'r5_d_1');
insert into Txd4 values(1, 2, 3, 6, 'r6_d_1');
```

```
insert into Txd4 values(1, 2, 3, 7, 'r7_d_1');
insert into Txd4 values(1, 2, 3, 8, 'r8_d_1');
insert into Txd4 values(1, 2, 3, 9, 'r9_d_1');
insert into Txd4 values(1, 2, 3, 10, 'r10_d_1');
```