Chapter 5
Data Acquisition and Integration

5.1 Introduction

This chapter first provides a brief review of data sources and types of variables from the point of view of data mining. Then it presents the most common procedures of data rollup and aggregation, sampling, and partitioning.

5.2 Sources of Data

In most organizations today data is stored in relational databases. The quality and utility of the data as well as the amount of effort needed to transform the data to a form suitable for data mining depends on the types of the applications the databases serve. These relational databases serve as data repositories for the following applications.

5.2.1 Operational Systems

Operational systems process the transactions that make an organization work. The data from these systems is, by nature, transient and keeps accumulating in the repository. A typical example of these systems is any banking transaction processing system that keeps records of opened and closed accounts, deposits, withdrawals, balances, and all other values related to the money moving among accounts, clients, and the outside world. Data extracted from such operational systems is the most *raw* form of data, in the sense that it has not been transformed, cleansed, or changed. It may contain errors due to data entry procedures or applications and usually has many missing values. It is also usually scattered over several tables and files. However, it is the most honest representation of the status of any business.

5.2.2 Data Warehouses and Data Marts

Data warehouses and data marts were conceived as a means to facilitate the compilation of regular reports on the status of the business by continuously collecting, cleaning, and summarizing the core data of the organization. Data warehouses provide a clean and organized source of data for data mining. In most cases, however, data warehouses were not created to prepare data for data modelers; they were rather created with a certain set of reporting functions in mind. Therefore, data residing in them might have been augmented or processed in a special way to facilitate those functions. Ideally, a specialized data mart should be created to house the data needed for data mining modeling and scoring processes.

5.2.3 OLAP Applications

OLAP, which stands for On Line Analytical Processing, and similar software are often given the name *Business Intelligence* tools. These applications reside in the data warehouse, or have their own data warehouse, and provide a graphical interface to navigate, explore, and "slice and dice" the data. The data structures that OLAP applications operate on are called *cubes.* They also provide comprehensive reporting capabilities. OLAP systems could be a source of data for data mining because of the interactive exploration capabilities that they offer the user. Therefore, the user would find the interesting data elements related to the problem through OLAP applications and then apply data mining modeling for prediction.

Alternatively, data mining can offer the identification of the significant variables that govern the behavior of some business measure (such as profit), and then OLAP can use these variables (as dimensions) to navigate and get qualitative insight into existing relationships. Data extracted from OLAP cubes may not be granular enough for data mining. This is because continuous variables are usually *binned* before they can be used as dimensions in OLAP cubes. This binning process results in the loss of some information, which may have a significant impact on the performance of data mining algorithms.

5.2.4 Surveys

Surveys are perhaps the most expensive source of data because they require direct interaction with customers. Surveys collect data through different communication channels with customers, such as mail, email, interviews, and forms on websites. There are many anecdotes about the accuracy and validity of the data collected from the different forms of surveys. However, they all share the following two common features:

- The number of customers who participate in the survey is usually limited because of the cost and the number of customers willing to participate.

- The questions asked in the survey can be designed to directly address the objective of the planned model. For example, if the objective is to market new products, the survey would ask customers about their preferences in these products, whether they would buy them, and what price would they pay for them.

These two points highlight the fact that, if well designed and executed, surveys are indeed the most accurate representation of possible customer behavior. However, they usually generate a limited amount of data because of the cost involved.

5.2.5 Household and Demographic Databases

In most countries, there are commercially available databases that contain detailed information on consumers of different products and services. The most common type is demographic databases based on a national census, where the general demographic profile of each residential area is surveyed and summarized. Data obtained from such database providers is usually clean and information-rich. Their only limitation is that data is not provided on the individual customer or record level, but rather averaged over a group of customers, for example, on the level of a postal (zip) code. Such limitations are usually set by privacy laws aimed at protecting individuals from abuse of such data.

The use of averaged data in models could lead to *diluting* the model's ability to accurately define a target group. For example, extensive use of census-like variables in a customer segmentation model would eventually lead to a model that clusters the population on the basis of the used census demographics and not in relation to the originally envisaged rate of usage or buying habits of the planned products or services.

It is not uncommon that analysts collect data from more than one source to form the initial mining view and for the scoring of mining models.

5.3 Variable Types

Designers of applications that use databases and different file systems attempt to optimize their applications in terms of the space required to keep the data and the speed of processing and accessing the data. Because of these considerations, the data extracted from databases is very often not in optimal form from the point of view of data mining algorithms. In order to appreciate this issue, we provide the following discussion of the types of variables that most data mining algorithms deal with.

5.3.1 Nominal Variables

Nominal, or *categorical*, variables describe values that lack the properties of order, scale, or distance between them. For example, the variables representing the type of a housing unit can take the categories House, Apartment, Shared Accommodation.

One cannot enforce any meaning of order or scale on these values. Other examples include Gender (Male, Female), Account Type (Savings, Checking), and type of Credit Card (VISA, MasterCard, American Express, Diners Club, EuroCard, Discover, . . .).

From the point of view of data mining algorithms, it is important to retain the *lack* of order or scale in categorical variables. Therefore, it is not desirable that a category be represented in the data by a series of integers. For example, if the type of a house variable is represented by the integers 1–4 (1 = Detached, 2 = Semi-detached, 3 = Townhome, 4 = Bungalow), a numerical algorithm may inadvertently add the numbers 1 and 2, resulting implicitly in the erroneous and meaningless statement of "Detached + Semi-detached = Townhome"! Other erroneous, and equally meaningless, implications that "Bungalow > Detached" or "Bungalow − Semi-detached = Townhome − Detached." The most convenient method of storing categorical variables in software applications is to use strings. This should force the application to interpret them as nominal variables.

5.3.2 Ordinal Variables

Ordinal, or *rank* or *ordered scalar*, variables are categorical variables with the notion of order added to them. For example, we may define the risk levels of defaulting on a credit card payment into three levels (Low, Medium, High). We can assert the order relationships High $\geq$ Medium $\geq$ Low. However, we cannot establish the notion of scale. In other words, we cannot accurately say that the difference between *High* and *Medium* is the same as the difference between *Medium* and *Low* levels of risk.

Based on the definition of ordinal variables, we can realize the problem that would arise when such variables are represented by a series of integers. For example, in the case of the risk level variable, representing these levels with numbers from 1 to 3 such that (Low = 1, Medium = 2, High = 3) would result in the imposition of an invalid notion of distance between the different values. In addition, this definition would impose the definition of scale on the values by implying that Medium risk is double the risk of Low, and High risk is three times the risk of Low.

Some ordinal variables come with the scale and distance notions added to them. These are best represented by a series of positive integers. They usually measure the frequency of occurrence of an event. Examples of such ordinal measures are number of local telephone calls within a month, number of people using a credit card in a week, and number of cars purchased by a prospective customer in her or his lifetime.

A typical problem, especially in data warehouses, exists in the representation of ordinal measures. Some ordinal measures are often subjected to "binning" to reduce the values we need to store and deal with. For example, a data warehouse may bin the number of times a customer uses a credit card per month to the representation 0–5 $\rightarrow$ 1, 6–10 $\rightarrow$ 2, 11–20 $\rightarrow$ 3, more than 20 $\rightarrow$ 4. Although this leads to a more compact representation of the variables, it may be detrimental to data mining algorithms for two reasons: (1) It reduces the granularity level of the data, which may result in a reduction in the predictive model accuracy, and (2) it distorts the ordinal nature of the original quantity being measured.

5.3.3 Real Measures

Real measures, or *continuous variables*, are the easiest to use and interpret. Continuous variables have all the desirable properties of variables: order, scale, and distance. They also have the meanings of zero and negative values defined. There could be some constraints imposed on the definition of continuous variables. For example, the age of a person cannot be negative and the monthly bill of a telephone line cannot be less than the subscription fees. Real measures are represented by real numbers, with any reasonably required precision.

The use of ratios in constrained continuous variables is sometimes troublesome. For example, if we allow the balance of a customer to be negative or positive, then the ratio between $ −10,000.00 and $ −5,000.00 is the same as that between $ +10,000.00 and $ +5,000.00. Therefore, some analysts like to distinguish between the so-called interval and ratio variables. We do not make that distinction here because in most cases the context of the implementation is clear. For example, if we wished to use the ratio of balances, we would restrict the balances to positive values only; if negative values occurred, we would devise another measure to signify that fact.

With the three types of variable from the mining point of view, the first task the analyst should consider, when acquiring the data, is to decide on the type of data to be used for each variable depending on its meaning. Of special interest are variables that represent *dates* and *times*. With the exception of time series analysis, dates and times are not useful in their raw form. One of the most effective methods of dealing with date and time values is to convert them to a *period* measure, that is, calculate the *difference* between the values and a fixed *reference* value. For example, instead of dealing with the date of opening an account, we deal with total tenure as the difference between today's date and the date of opening the account. In fact, we use this method every day by referring to the age of a person instead of her or his birth date. In this way, we convert dates and times to real measures, with some constraint if necessary, as in the case of a person's age. (Negative age is not well defined!)

5.4 Data Rollup

The simplest definition of data rollup is that we convert categories to variables. Let us consider an illustrative example.

Table 5.1 shows some records from the transaction table of a bank where deposits are denoted by positive amounts and withdrawals are shown as negative amounts.

We further assume that we are building the mining view as a *customer view*. Since the first requirement is to have one, and only one, row per customer, we create a new view such that each unique Customer ID appears in one and only one row. To *roll up* the multiple records on the customer level, we create a set of new variables to represent the combination of the account type and the month of the transaction. This is illustrated in Figure 5.1. The result of the rollup is shown in Table 5.2.

Table 5.1 A sample of banking transactions.

Customer ID	*Date*	*Amount*	*Account type*
1100-55555	11Jun2003	114.56	Savings
1100-55555	21Jun2003	−56.78	Checking
1100-55555	07Jul2003	359.31	Savings
1100-55555	19Jul2003	89.56	Checking
1100-55555	03Aug2003	1000.00	Savings
1100-55555	17Aug2003	−1200.00	Checking
1100-88888	14June2003	122.51	Savings
1100-88888	27June2003	42.07	Checking
1100-88888	09July2003	−146.30	Savings
1100-88888	09July2003	−1254.48	Checking
1100-88888	10Aug2003	400.00	Savings
1100-88888	11Aug 2003	500.00	Checking
. . .			

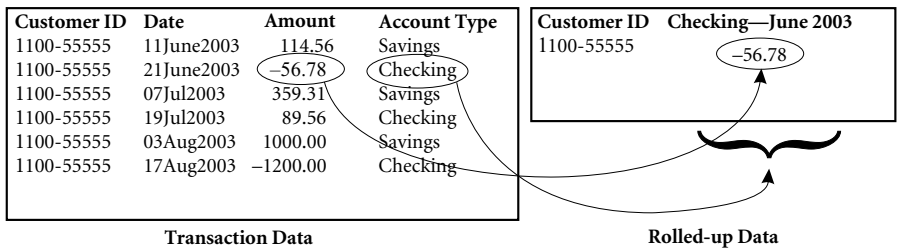

Figure 5.1 Data rollup.

Table 5.1 shows that we managed to aggregate the values of the transactions in the different accounts and months into new variables. The only issue is what to do when we have more than one transaction per account per month. In this case, which is the more realistic one, we have to summarize the data in some form. For example, we can calculate the sum of the transactions values, or their average, or even create a new set of variables giving the count of such transactions for each month–account type combination.

It is obvious that this process will lead to the generation of possibly hundreds, if not thousands, of variables in any data-rich business applications. Dealing with such

Table 5.2 Result of rolling up the data of Table 5.1.

Cust. ID	*C-6*	*C-7*	*C-8*	*S-6*	*S-7*	*S-8*
1100-55555	−56.78	89.56	−1200.00	114.56	359.30	1000.00
1100-88888	42.07	−1254.00	500.00	122.51	−146.30	400.00

a large number of fields could present a challenge for the data preparation and data mining software tools. It is therefore required that we keep the number of these new fields to a minimum while keeping as much information about the nature of the data as possible. Unfortunately, there is no magic recipe to achieve this balance. However, a closer look at the preceding data reveals that the key to controlling the number of new variables is to decide on the level of granularity required to perform the rollup. For example, is it necessary to roll up the transactions of each month, or is it enough to roll up the data per quarter? Similarly, in our simplified case, we had only two categories for the account type, but typically, there would be many more categories. Then comes the question of which categories we can group together, or even ignore, to reduce the number of new variables.

In the end, even with careful selection of the categories and resolution of combining the different categories to form new variables, we usually end up with a relatively large number of variables, which most implementations of data mining algorithms cannot handle adequately. However, we should not worry too much about this problem for the moment because data reduction is a basic step in our planned approach. In later chapters, we will investigate techniques to reduce the number of variables.

In the last example demonstrating the rollup process, we performed the rollup on the level of two variables: the account type and the transaction month. This is usually called *multilevel rollup*. On the other hand, if we had had only one type of account, say only savings, then we could have performed a simpler rollup using only the transaction month as the summation variable. This type of rollup is called *simple rollup*. In fact, multilevel rollup is only an aggregation of several simple rollups on the row level, which is the customer ID in our example. Therefore, data preparation procedures, in either SAS or SQL, can utilize this property to simplify the implementation by performing several simple rollups for each combination of the summarization variables and combining them. This is the approach we will adopt in developing our macro to demonstrate the rollup of our sample dataset.

Now let us describe how to perform the rollup operation using SAS. We will do this using our simple example first and then generalize the code using macros to facilitate its use with other datasets. We stress again that in writing the code we preferred to keep the code simple and readable at the occasional price of efficiency of execution and the use of memory resources. You are welcome to modify the code to make it more efficient or general as required.

We use Table 5.1 to create the dataset as follows.

```
Data Transaction;
Informat CustID $10.;
Informat TransDate date9.;
format TransDate Date9.;
input CustID $  TransDate Amount AccountType$;
Cards;
55555   11Jun2003     114.56    Savings
55555   12Jun2003     119.56    Savings
55555   21Jun2003     -56.78    Checking
55555   07Jul2003     359.31    Savings
55555   19Jul2003      89.56    Checking
55555   03Aug2003    1000.00    Savings
66666   22Feb2003     549.56    Checking
77777   03Dec2003     645.21    Savings
55555   17Aug2003   -1200.00    Checking
88888   14Jun2003     122.51    Savings
88888   27Jun2003      42.07    Checking
88888   09Jul2003    -146.30    Savings
88888   09Jul2003   -1254.48    Checking
88888   10Aug2003     400.00    Savings
88888   11Aug2003     500.00    Checking
;
run;
```

The next step is to create the month field using the SAS `month` function.

```
data Trans;
 set Transaction;
  Month = month(TransDate);
run;
```

Then we accumulate the transactions into a new field to represent the balance in each account.

```
proc sort data=Trans;
  by CustID month AccountType;
run;

/* Create cumulative balances for each of the accounts */
data Trans2;
 retain Balance 0;
 set Trans;
  by CustID month AccountType;
  if first.AccountType then Balance=0;
   Balance = Balance + Amount;
   if last.AccountType then output;
   drop amount;
run;
```

Finally, we use PROC TRANSPOSE to roll up the data in each account type and merge the two resulting datasets into the final file.

```
/* Prepare for the transpose */
proc sort data=trans2;
 by CustID accounttype;
 run;

proc transpose data =Trans2 out=rolled_C prefix=C_;
 by CustID accounttype;
ID month ;
var balance ;
where AccountType='Checking';
run;

proc transpose data =Trans2 out=rolled_S prefix=S_;
by CustID accounttype;
ID month ;
var balance ;
where AccountType='Savings';
run;

data Rollup;
 merge Rolled_S Rolled_C;
 by CustID;
 drop AccountType _Name_;
run;
```

To pack this procedure in a general macro using the combination of two variables, one for transaction categories and one for time, we simply replace the Month variable with a TimeVar, the customer ID with IDVar, and the AccountType with TypeVar. We also specify the number of characters to be used from the category variable to prefix the time values. Finally, we replace the two repeated TRANSPOSE code segments with a %do loop that iterates over the categories of the TypeVar (which requires extracting these categories and counting them). The following is the resulting macro.

Step 1

Sort the transaction file using the ID, Time, and Type variables.

```
proc sort data=&TDS;
by &IDVar &TimeVar &TypeVar;
run;
```

Step 2

Accumulate the values over time to a temporary _Tot variable in the temporary table Temp1 (see Table 5.3). Then sort Temp1 using the ID and the Type variables.

Table 5.3 Parameters of macro `TBRollup()`.

Header	`TBRollup(TDS, IDVar, TimeVar, TypeVar, Nchars, Value, RDS)`
Parameter	*Description*
`TDS`	Input transaction dataset
`IDVar`	ID variable
`TimeVar`	Time variable
`TypeVar`	Quantity being rolled up
`Nchars`	Number of characters to be used in rollup
`Value`	Values to be accumulated
`RDS`	The output rolled up dataset

```
data Temp1;
retain _TOT 0;
set &TDS;
by &IDVar &TimeVar &TypeVar;
if first.&TypeVar then _TOT=0;
_TOT = _TOT + &Value;
if last.&TypeVar  then output;
drop &Value;
   run;
proc sort data=Temp1;
by &IDVar &TypeVar;
run;
```

Step 3

Extract the categories of the `Type` variable, using `PROC FREQ`, and store them in macro variables.

```
proc freq data =Temp1 noprint;
tables &TypeVar /out=Types ;
run;

data _null_;
set Types nobs=Ncount;
if &typeVar ne " then
call symput('Cat_'||left(_n_), &TypeVar);
if _N_=Ncount  then call symput('N', Ncount);
run;
```

Step 4

Loop over these `N` categories and generate their rollup part.

```
%do i=1 %to &N;
proc transpose
data =Temp1
out=_R_&i
prefix=%substr(&&Cat_&i, 1, &Nchars)_;
by &IDVar &TypeVar;
ID &TimeVar ;
var _TOT ;
where &TypeVar="&&Cat_&i";
run;
%end;
```

Step 5
Finally, assemble the parts using the `ID` variable.

```
data &RDS;
merge %do i=1 %to &N; _R_&i %end ; ;
by &IDVar;
drop &TypeVar _Name_;
run;
```

Step 6
Clean the workspace and finish the macro.

```
proc datasets library=work nodetails;
delete Temp1 Types %do i=1 %to &N; _R_&i  %end; ;
run;
quit;

%mend;
```

We can now call this macro to roll up the previous example `Transaction` dataset using the following code:

```
data Trans;
set Transaction;
Month = month(TransDate);
drop transdate;
run;

  %let IDVar   = CustID;      /* The row ID variable */
  %let TimeVar = Month;       /* The time variable */
  %let TypeVar = AccountType; /* The Type variable */
  %let Value   = Amount;      /* The time measurement variable */
  %let NChars  = 1;           /* Number of letters in Prefix */
  %let TDS     = Trans;       /* The value variable */
  %let RDS     = Rollup;      /* the rollup file */
```

```
%TBRollup(&TDS, &IDVar, &TimeVar, &TypeVar, &Nchars,
          &Value, &RDS);
```

The result of this call is shown in Table 5.4.

Table 5.4 Result of rollup macro.

CustID	C_6	C_7	C_8	C_{12}	S_6	S_7	S_8	S_{12}
5555	−56.78	89.56	−1200	.	234.12	359.31	1000	.
6666	.	.	.	549.56	.	.	.	.
7777	.	.	.	.	.	.	.	645.21
8888	42.07	−1254.48	500	.	122.51	−146.3	400	.

5.5 Rollup with Sums, Averages, and Counts

In addition to finding the sum of a value variable during the rollup, it may also be more meaningful sometimes to calculate average value and/or the number of records that represent certain events—for example, number of deposits, number of withdrawals, or number of mailings a customer received responding to an offer.

In our rollup macro, these requirements would alter only the middle part of our code, where we calculated the cumulative value of the `Value` variable. The following code segment would modify the macro to calculate the average value and the number of transactions for each account type instead of the total.

Step 2

```
data _Temp1;
retain _TOT 0;
retain _NT 0;
set &TDS;
by &IDVar &TimeVar &TypeVar;
if first.&TypeVar then _TOT=0;
_TOT = _TOT + &Value;
if &Value ne . then _NT=_NT+1;
if last.&TypeVar  then
do;
_AVG=_TOT/_NT;
output;
_NT=0;
end;
drop &Value;
run;
```

Furthermore, the code inside the %do loop should also reflect our interest in transposing the values of the average variable, _AVG. Therefore, the code will be as follows

Step 4

```
%do i=1 %to &N;
Oproc transpose
data =_Temp1
out=_R_&i
prefix=%substr(&&Cat_&i, 1, &Nchars)_;
by &IDVar &TypeVar;
ID &TimeVar;
var _AVG;
where &TypeVar="&&Cat_&i";
run;
%end;
```

The complete code for the modified code to roll up the average value is included in the macro ABRollup().

5.6 Calculation of the Mode

Another useful summary statistic is the mode, which is used in both the rollup stage and the EDA (see Chapter 7). The mode is the most common category of transaction. The mode for nominal variables is equivalent to the use of the average or the sum for the continuous case. For example, when customers use different payment methods, it may be beneficial to identify the payment method most frequently used by each customer.

The computation of the mode on the mining view entity level from a transaction dataset is a demanding task because we need to search for the frequencies of the different categories for *each* unique value of the entity variable. The macro shown in Table 5.5 is based on a *classic* SQL query for finding the mode on the entity level from a

Table 5.5 Parameters of macro VarMode().

Header	VarMode(TransDS, XVar, IDVar, OutDS)
Parameter	*Description*
TransDS	Input transaction dataset
XVar	Variable for which the mode is to be calculated
IDVar	ID variable
OutDS	The output dataset with the mode for unique IDs

transaction table. The variable being searched is XVar and the entity level is identified through the unique value of the variable IDVar.

```
%macro VarMode(TransDS, XVar, IDVar, OutDS);
/* A classic implementation of the mode of transactional
   data using SQL */
proc sql noprint;
create table &OutDS as
SELECT &IDVar , MIN(&XVar ) AS mode
FROM (
               SELECT &IDVar,  &XVar
               FROM &TransDS p1
               GROUP BY &IDVar, &XVar
               HAVING COUNT( * ) =
                     (SELECT MAX(CNT )
                      FROM (SELECT COUNT( * ) AS CNT
                            FROM &TransDS p2
                            WHERE p2.&IDVar= p1.&IDVar
                            GROUP BY p2.&XVar
                            ) AS p3
                      )
             ) AS p
       GROUP BY p.&IDVar;
quit;
%mend;
```

The query works by calculating a list holding the frequency of the `XVar` categories, identified as `CNT`, then using the maximum of these counts as the mode. The query then creates a new table containing `IDVar` and `XVar` where the `XVar` category frequency is equal to the maximum count, that is, the mode.

The preceding compound `SELECT` statement is computationally demanding because of the use of several layers of `GROUP BY` and `HAVING` clauses. Indexing should always be considered when dealing with large datasets. Sometimes it is even necessary to partition the transaction dataset into smaller datasets before applying such a query to overcome memory limitations.

5.7 Data Integration

The data necessary to compile the mining view usually comes from many different tables. The rollup and summarization operations described in the last two sections can be performed on the data coming from each of these data sources independently. Finally, we would be required to assemble all these segments in one mining view. The most used assembly operations are *merging* and *concatenation*. Merging is used to collect data for the same key variable (e.g., customer ID) from different sources. Concatenation is used to assemble different portions of the same data fields for different segments of the key variable. It is most useful when preparing the scoring view with a very large number of observations (many millions). In this case, it is more

efficient to partition the data into smaller segments, prepare each segment, and finally concatenate them together.

5.7.1 Merging

SAS provides several options for merging and concatenating tables together using DATA step commands. However, we could also use SQL queries, through PROC SQL, to perform the same operations. In general, SAS DATA step options are more efficient in merging datasets than PROC SQL is. However, DATA step merging may require sorting of the datasets before merging them, which could be a slow process for large datasets. On the other hand, the performance of SQL queries can be enhanced significantly by creating indexes on the key variables used in merging.

Because of the requirement that the mining view have a unique record per category of key variable, most merging operations required to integrate different pieces of the mining view are of the type called *match-merge with non-matched observations*. We demonstrate this type of merging with a simple example.

Example 5.1 We start with two datasets, Left and Right, as shown in Table 5.6.

Table 5.6 Two sample tables: Left and Right.

Table: Left			*Table:* Right		
ID	*Age*	*Status*	*ID*	*Balance*	*Status*
1	30	Gold	2	3000	Gold
2	20	.	4	4000	Silver
4	40	Gold			
5	50	Silver			

The two tables can be joined using the MERGE–BY commands within a DATA step operation as follows:

```
DATA Left;
 INPUT ID Age Status $;
 datalines;
 1  30  Gold
 2  20  .
 4  40  Gold
 5  50  Silver
 ;
RUN;
```

```
DATA Right;
 INPUT ID Balance Status $;
 datalines;
 2  3000  Gold
 4  4000  Silver
 ;
RUN;

DATA Both;
 MERGE Left Right;
 BY ID;
RUN;

PROC PRINT DATA=Both;
RUN;
```

The result of the merging is the dataset `Both` given in Table 5.7, which shows that the `MERGE-BY` commands did merge the two datasets as needed using `ID` as the key variable. We also notice that the common file `Status` was overwritten by values from the `Right` dataset. Therefore, we have to be careful about this possible side effect. In most practical cases, common fields should have identical values. In our case, where the variable represented some customer designation status (`Gold` or `Silver`), the customer should have had the same status in different datasets. Therefore, checking these status values should be one of the data integrity tests to be performed before performing the merging.

Table 5.7 Result of merging: dataset `Both`.

Obs	*ID*	*Age*	*Status*	*Balance*
1	1	30	Gold	.
2	2	20	Gold	3000
3	4	40	Silver	4000
4	5	50	Silver	.

Merging datasets using this technique is very efficient. It can be used with more than two datasets as long as all the datasets in the `MERGE` statement have the common variable used in the `BY` statement. The only possible difficulty is that SAS requires that *all* the datasets be sorted by the `BY` variable. Sorting very large datasets can sometimes be slow.

◆

You have probably realized by now that writing a general macro to merge a *list* of datasets using an `ID` variable is a simple task. Assuming that all the datasets have been sorted using `ID` prior to attempting to merge them, the macro would simply be given as follows:

```
%macro MergeDS(List, IDVar, ALL);
DATA &ALL;
```

```
     MERGE &List;
     by &IDVar;
run;
%mend;
```

Finally, calling this macro to merge the two datasets in Table 5.6 would simply be as follows:

```
%let List=Left Right;
%let IDVar=ID;
%let ALL = Both;
%MergeDS(&List, &IDVar, &ALL);
```

5.7.2 Concatenation

Concatenation is used to attach the contents of one dataset to the end of another dataset without duplicating the common fields. Fields unique to one of the two files would be filled with missing values. Concatenating datasets in this fashion does not check on the uniqueness of the ID variable. However, if the data acquisition and rollup procedures were correctly performed, such a problem should not exist.

Performing concatenation in SAS is straightforward. We list the datasets to be concatenated in a SET statement within the destination dataset. This is illustrated in the following example.

Example 5.2 Start with two datasets TOP and BOTTOM, as shown in Tables 5.8 and 5.9.

Table 5.8 Table: TOP.

Obs	*ID*	*Age*	*Status*
1	1	30	Gold
2	2	20	.
3	3	30	Silver
4	4	40	Gold
5	5	50	Silver

Table 5.9 Table: BOTTOM.

Obs	*ID*	*Balance*	*Status*
1	6	6000	Gold
2	7	7000	Silver

We then use the following code to implement the concatenation of the two datasets into a new dataset.

```
DATA TOP;
 input ID Age Status $;
 datalines;
 1  30  Gold
 2  20  .
 3  30  Silver
 4  40  Gold
 5  50  Silver
 ;
run;
DATA BOTTOM;
input ID Balance Status $;
 datalines;
 6  6000  Gold
 7  7000  Silver
 ;
run;

DATA BOTH;
 SET TOP BOTTOM;
run;
```

The resulting dataset is shown in Table 5.10.

Table 5.10 Table: BOTH.

Obs	*ID*	*Age*	*Status*	*Balance*
1	1	30	Gold	.
2	2	20	.	.
3	3	30	Silver	.
4	4	40	Gold	.
5	5	50	Silver	.
6	6	.	Gold	6000
7	7	.	Silver	7000

As in the case of merging datasets, we may include a list of several datasets in the SET statement to concatenate. The resulting dataset will contain all the records of the contributing datasets in the same order in which they appear in the SET statement.

◆

The preceding process can be packed into the following macro.

```
%macro ConcatDS(List, ALL);
DATA &ALL;
 SET &List;
run;
%mend;
```

To use this macro to achieve the same result as in the previous example, we use the following calling code.

```
%let List=TOP BOTTOM;
%let ALL = BOTH;
%ConcatDS(&List, &ALL);
```

This Page Intentionally Left Blank