# 1

# Using SAS in a Data Mart, Data Lake, or Data Warehouse

The purpose of this chapter is to showcase how SAS has been used in data warehousing over its lifetime, and how that history impacts SAS data warehousing today. It provides an opportunity to see how slight changes in coding in SAS **data steps** can greatly impact data **input/output (I/O)**. It also covers how SAS data is managed, and how **Base SAS**, the analytic component, interacts with stored data.

As SAS developed, there became a need to set **indexes** on variables, and to use **SQL** coding in SAS. How PROC SQL in SAS compares with data steps and other SQL programming will be reviewed in this chapter. I will also explain strategies to deal with memory issues in SAS, and how it has evolved to now be used with data in the cloud.

In this chapter, we are going to cover the following main topics:

- How early versions of SAS handled data
- Different ways to access data in SAS
- Considerations in improving I/O in SAS
- Dealing with storage and memory issues in SAS
- Using SAS in modern data warehousing

> **Note:**
> The links to all the white papers and other sources mentioned in the chapter are provided in the *Further reading* section toward the end of the chapter.

# Technical requirements

The dataset used as a demonstration in this chapter, in `*.csv` format, can be found online on GitHub: `https://github.com/PacktPublishing/Mastering-SAS-Programming-for-Data-Warehousing/blob/master/Chapter%201/Datasets/Chap%201_1_Infile.csv`.

The code bundle for this chapter is available on GitHub here: `https://github.com/PacktPublishing/Mastering-SAS-Programming-for-Data-Warehousing/tree/master/Chapter%201`.

# Using original versions of SAS

Initially, SAS data had to be input through code into memory whenever analysis code was to be run on the data. This section covers the following:

- How to enter data into **SAS datasets** using SAS
- The early **PROCs** developed, such as `PROC PRINT` and `PROC MEANS`
- Improvements to data handling made in **Base SAS**

In this section, you will learn how SAS's data management processes were initially developed. The processes impact how SAS runs today.

## Initial SAS data handling

As described on SAS's website (`https://www.sas.com/en_us/company-information/profile.html`), SAS was invented in 1966 as the **Statistical Analysis System**, developed under a grant from the **United States** (**US**) **National Institutes of Health** (**NIH**) to eight universities. The immediate need was to develop a computerized system that could analyze the large amount of agriculture data being collected through the **US Department of Agriculture** (**USDA**).

According to the SAS history listed on the Barr Systems website (`http://www.barrsystems.com/about_us/the_company/professional_history.asp`), Anthony J. Barr was in the Statistics Department of North Carolina State University and was recruited to help program SAS. He was responsible for developing the first **analysis of variance** (**ANOVA**) and regression programs in SAS and created the software for inputting and transforming data.

Even today, it is relevant here to reflect on Barr's early development of what would later be called **data step** language in SAS. This is because current data import processes in SAS continue to use roughly the same approach, which presents both opportunities and limitations in data warehouse management.

In the early data step code, data was entered as part of the code, which still can be done today. Let's consider a modern example of using data step code to enter data in SAS by referring to the **2018 BRFSS Codebook** listed in the technical requirements for this chapter. Each year, the **United States Centers for Disease Control and Prevention** (**CDC**) organizes an annual anonymous phone survey of approximately 450,000 residents asking about health conditions and risk factors. This survey is called the **Behavioral Risk Factor Surveillance System (BRFSS)**. The 2018 BRFSS Codebook describes the 2018 version of a SAS dataset from a survey in the US that is conducted by phone every year.

The *codebook* describes specifications about each variable in the dataset, including the following:

- Variable name
- Allowable values
- Frequencies in the dataset for each value

The BRFSS Codebook is quite extensive and can be confusing for an analyst without a background in the dataset to understand it. In *Chapter 3*, *Helpful PROCs for Managing Data,* we will look closely at an example from the BRFSS Codebook. For now, let's review a codebook that is easier for the beginner to interpret. Here is an example of a codebook entry from the online codebook for the **US National Health and Nutrition Examination Survey** (**NHANES**):

BMXWT - Weight (kg)

**Variable name:** BMXWT
**SAS Label:** Weight (kg)
**English Text:** Weight (kg)
**Target:** Both males and females
0 YEARS - 160 YEARS

| Code or Value | Value Description | Count | Cumulative |
|---|---|---|---|
| 3.6 to 198.9 | Range of Values | 9445 | 9445 |
| . | Missing | 99 | 9544 |

Figure 1.1 – Example of a codebook entry from the US NHANES

The following table represents how three of the variables in the BRFSS Codebook – `_STATE`, `SEX1`, and `_AGE80` – could be represented in three lines of data:

| _STATE | SEX1 | _AGE80 |
|---|---|---|
| 12 | 1 | 72 |
| 25 | 2 | 25 |
| 27 | 2 | 54 |

Table 1.1 – Example of three variable values for three respondents in the 2018 BRFSS dataset

Here, the state of residence of the respondent is recorded under `X_STATE` according to its corresponding numerical **Federal Information Processing System** (**FIPS**) number, and `SEX1` is coded as `1` for **male** and `2` for **female**, `7` for **don't know/not sure**, and `9` for **refused** (to decode state FIPS numbers, please see the link for the FIPS state codes list in the *Further reading* section). The `_AGE80 variable` refers to the age of the respondent imputed from other data (with ages over 80 collapsed). Using the codebook to decode the preceding data, we see the three rows represent a 72-year-old man from Florida (FL), a 25-year-old woman from Massachusetts (MA), and a 54-year-old woman from Minnesota (MN).

Let's look at an example of using data step code to create this table in SAS:

```
DATA THREEROWS;
     INFILE CARDS;
     INPUT _STATE SEX1 _AGE80;
     CARDS;
12 1 72
25 2 25
27 2 54
;
RUN;
```

Let's go through the code:

1. The `THREEROWS` dataset is created in the **WORK directory** of SAS. The `WORK` directory, simply called `WORK`, is the working directory for the SAS session, which means when the session is over, the data in `WORK` will be erased.
2. As is typical in SAS programming, each of the programming lines ends with a semi-colon, except each of the data lines.
3. The next line, `INFILE CARDS;`, indicates to SAS that data will now be entered from cards (although it is possible to replace this with the more modern version of the command, `datalines`).
4. When Barr designed this process, the next step would be for SAS to input punch cards that held the data. The next line, `INPUT _STATE SEX1 _AGE80;`, designates that the data that will be input from the cards has these headers: `_STATE`, `SEX1`, and `_AGE80`.
5. The next line, `CARDS;`, indicates that it is time for the cards to start to be read. What follows in the code is our modern representation of entering the data represented in the table into SAS using `CARDS`.

6. By the time SAS processes the `CARDS` statement, it already knows from the `INPUT` statement to expect three columns – `_STATE`, `SEX1`, and `_AGE80` – so even without formatting the lines in three rows, SAS would read the values sequentially and assemble the dataset with three columns, ending when it hits the semi-colon at the end.
7. These three variables are numeric by default unless `'$'` is included in the `INPUT` statement.

> **Note:**
> It is not a good idea to store actual data values in SAS code today. They can easily be lost, and if the data is private, it can create privacy issues around the code. Further more, many of the datasets used today are extremely large, and it would not be practical to store them as actual data values in SAS; instead, they might be stored in a database system such as Oracle, or in an Excel file.

## Early SAS data handling

In September 1966, the conceptual ideas behind SAS were presented by Barr and others to the Committee on Statistical Software of the **University Statisticians of Southeast Experiment Station** (**USSERS**) at their meeting held in Athens, GA. Barr began working with others, including the current SAS CEO, James Goodnight, on developing the first worldwide release of SAS.

> **Note:**
> The first worldwide release of SAS in 1972 consisted of 37,000 lines of code, 65% written by Barr, and 32% written by Goodnight.

Improvements implemented in the 1972 worldwide release of SAS focused on procedures known as **PROCs**. Procedures are applied to SAS datasets. Some PROCs are for data editing and handling, but most are focused on data visualization and analysis, with the data handling typically done using data steps. Barr developed some basic PROCs still used today to assist in data handling, including `PROC SORT` and `PROC PRINT`.

Let's look at an example of `PROC SORT` and `PROC PRINT`. Coming back to the data we entered earlier, the rows were sorted according to the value in the `_STATE` variable:

- If we wanted to sort the dataset in order of the respondent's age, or `_AGE80`, we could use `PROC SORT` with the `_AGE80 command`.
- Following that with a `PROC PRINT` would then print the resulting dataset to the screen.

This is shown in the following code:

```
PROC SORT data=THREEROWS;
by _AGE80;
PROC PRINT;
RUN;
```

While Barr also developed analysis PROCs such as `PROC ANOVA`, which conducts an ANOVA, Goodnight developed PROCs aimed principally at analysis, such as `PROC CORR` for correlations and `PROC MEANS` for calculating means. The most ideal situation at the time for data handling was to have the data already stored on the cards, essentially in the format it needed to be in for analysis. However, data step code was available for editing the data.

At this time, NIH discontinued funding the project, but the consortium of universities that had worked on the project agreed to provide funding support, allowing the programmers to continue building SAS. Barr, Goodnight, and others continued to develop the software, adding mainly statistical functions rather than data management functions, and released a 1976 version. For the 1976 version, Barr rewrote the *internals* of SAS, including the following:

- Data management functions
- Report-writing functions
- The compiler

This was the first big rewrite of SAS's processing functions.

In 1976, SAS Institute, Inc. was incorporated, with ownership split between Barr, Goodnight, and two others:

Over 100 organizations including pharmaceutical companies, insurance companies, banks, governmental entities, and members of the academic community were using SAS.

More than 300 people attended the first SAS users conference in 1976.

Reflecting on this short history, it is understandable that even today, SAS maintains the reputation of being the only statistical software that can comprehensively handle **big data**. While in some regards this statement remains true, it is also necessary to revisit a more subtle point, which is that SAS was initially developed for data analysis – not for data storage. Even with improvements, SAS data handling is still limited by some of the features originally developed in this early era.

## SAS data handling improvements

Both the SAS programs and the data SAS analyzed were initially stored on punch cards. These were physical cards with hole punches in them to indicate instructions to the computer. The following photograph shows a real punch card that was used for an IBM 1130 program:

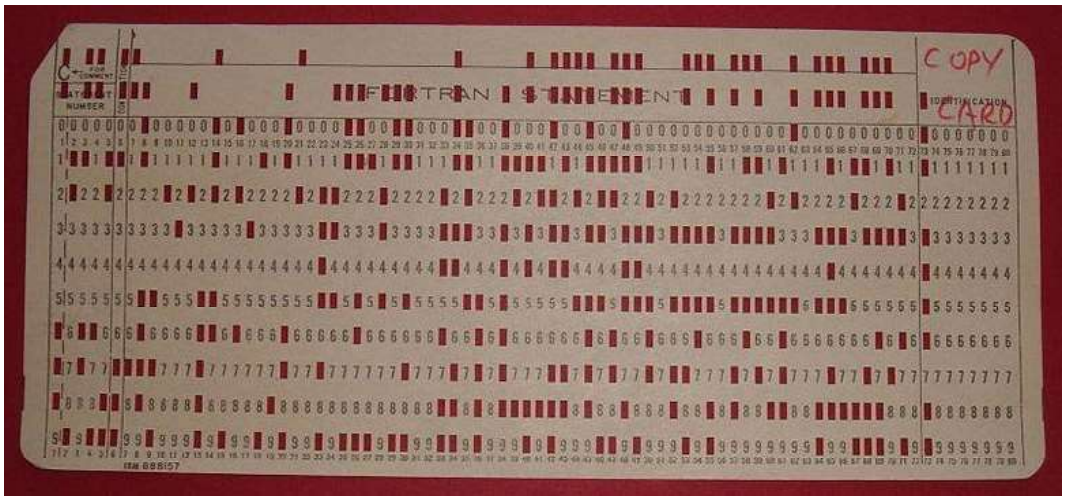


Figure 1.2 – This card contains a self-loading w:IBM 1130 program that copies the deck of cards placed after it in the input hopper. Photograph by Arnold Reinhold, CC BY-SA 2.5 (`https://creativecommons.org/licenses/by-sa/2.5/deed.en`)

In his 2005 report titled *Programming with Punch Cards*, Dale Fisk explains how creating a set of punch cards to run a computer program was a multi-step, labor- and time-intensive process:

1. First, cards had to be punched by hand.
2. Next, the program punched into the cards had to be compiled through a computer, which would produce a printed list of errors if there were any.
3. If the program compiled, the computer would print a set of cards with the compiled program.
4. This new set of cards would be the ones used to launch the program.

SAS was the first application that could feasibly handle running programs to analyze large datasets using punch cards. The positive result of the development of early SAS programs was the ability to use these punch cards to run complex regressions that could never have been attempted before.

But punch cards also created challenges. The foundation SAS component, called **Base SAS**, was about 300,000 lines of code. This program was stored in 150 boxes of cards that would stand 40 feet high. These boxes were separate from boxes of cards of data that SAS would be used to analyze.

This meant that card storage was an issue. A lot of space was required already for the computers themselves, which could take up entire rooms. In addition, the cards were unwieldy and required their own set of error handling procedures, including the `LOST CARD` error that can still be displayed in SAS today under particular circumstances when there is an error reading in data.

Nevertheless, SAS continued to reach out and recruit new customers. From the beginning, SAS has always prided itself on its customer service. As of 1978, there were 600 SAS customer sites, but only 21 employees. The climate was that of everyone pitching in to help fill customer needs, and Goodnight was known for recognizing the value of employees to the company.

# Accessing data in SAS

This section covers how accessing data in SAS changed over the years:

- First, SAS data storage moved from punch cards to **mainframes**.
- Next, the invention of **personal computers (PCs)** led to reconfiguring how SAS data was accessed.
- Consequently, reading data into SAS from external data files became more common.

In this section, we will discuss how to read data in SAS from an external file, as well as the opportunities and limitations of how SAS processes data.

## Upgrading to mainframes

In 1979, Databank of New Zealand adapted SAS to run under IBM's VM/CMS system using IBM's **disk operating system** (**DOS**), thus solving the punch card problem and establishing SAS as mainframe software that was remotely hosted. This represented essentially the second rewrite of SAS since its 1976 rewrite. This upgrade made SAS more easily accessible to more customers. It also facilitated the ability for SAS to include more sophisticated components to add onto Base SAS. At the same time, it created new challenges for efficient data **input and output (I/O)**.

In 1979, Barr resigned from SAS, and SAS moved into its current headquarters in Cary, North Carolina (NC). In 1980, SAS added the components SAS/GRAPH for the presentation of graphics, and SAS/ETS for econometric and time series analysis. Prior to SAS/GRAPH, plots were developed using text characters. SAS/GRAPH allowed the output to be displayed as graphics rather than text.

New PROCs added as part of the new SAS/GRAPH component included `PROC GCHART` and `PROC GPLOT`. In the current SAS University Edition, these PROCs are no longer available and have been replaced with updated versions. At the time, however, SAS/GRAPH was considered a great improvement in graphical display over what had been available previously.

> **Note:**
>
> For examples of `PROC GCHART` and `PROC GPLOT` output, read Mike Kalt and Cynthia Zender's white paper on SAS graphics for the non-statistician (included in the *Further reading* section).

During the 1980s, SAS as a company grew dramatically; its campus expanded to 18 buildings that included a training center, publications warehouse, and video studio. By the end of the 1980s, SAS had nearly 1,500 worldwide employees and had established new offices on four continents. But as the 1980s wore on, PCs were becoming popular, and customers were demanding a way of running SAS on PCs.

Therefore, SAS had to iterate again in order to keep up with the pace of technological innovation in the background. Even though SAS was now running on mainframes, SAS's effort in innovation had been concentrated on the PROCs, and less attention was paid to optimizing the data management functions provided by the data steps. The response by both the company and SAS was to find ways to optimize the functioning of the SAS system, rather than rebuilding PROCs or data steps.

## Transitioning to personal computers

To accommodate PC users, in the 1980s, SAS was rewritten in the C language, which was popular at the time for PC applications. At the same time, SAS developed a new software architecture to run across multiple platforms, which SAS is still known for today. At the time, this PC functionality was introduced as a **micro-to-mainframe** link, allowing customers to store a dataset on a mainframe while running programs from their PCs.

Having an application running on PCs afforded SAS the opportunity to improve the user experience. SAS developed a **graphical user interface (GUI)** that resembled the Macintosh and Windows environments that were popular at the time and continued to move away from a numbers-centric format for data display and toward enhanced graphics and visualizations.

> **Note:**
> PC SAS does not run on Macintosh computers. But during the 1980s, SAS developed **JMP*** (pronounced *jump*), which is a statistical program that can perform many of the same tasks as SAS on macOS.

SAS still had the limitations that its data steps were sequential; so it still read data line by line, just as it had done with punch cards. Data steps were the main functions used to export data out of SAS, and therefore, SAS exported data line by line. This created a lack of flexibility in the format of output files. So, to get around this, the **Output Delivery System (ODS)** was created. This system allows the user to format output in a variety of formats and is still used currently in SAS today, such as Excel, `*.pdf`, or `*.rtf` files, with a specific component for delivering graphics called **ODS Graphics**.

While this period of SAS's evolution brought many innovations, they were mostly in the area of improving the user experience, rather than focusing on data handling. In terms of the micro-to-mainframe link, the development was mostly focused on the *micro* rather than the *mainframe* component. This focus on user experience seemed consistent with SAS's values of putting customers and employees first. During the 1980s, the company was recognized by its customers as helping them make sense out of their vast amount of data and helping them have the results of their data analysis guide their decisions. It also innovated in the area of employee wellness, opening an on-site childcare center in 1981, followed by establishing an on-site fitness center, health care center, and café.

## Reading external files

With the movement to PCs, customers wanted to import external data files into local copies of SAS, rather than using `CARDS` or `DATALINES` to input data, or connecting to a mainframe data source. This was accomplished through revisions to the `INFILE` statement, the use of external file references, and the setting of options. Today, SAS has created an automated way to read files using a **graphical user interface (GUI)** which launches a wizard that creates `PROC IMPORT` code.

If you are using the University Edition of SAS, you can place the data file for this chapter named `Chap_1_1_Infile.csv` into your `myfolders` folder, and if you run the following `PROC IMPORT` code, the file should be imported into `WORK`:

```
%web_drop_table(WORK.IMPORT);
FILENAME REFFILE '/folders/myfolders/Chap 1_1_Infile.csv';
PROC IMPORT DATAFILE=REFFILE
     DBMS=CSV
     OUT=WORK.IMPORT;
     GETNAMES=YES;
RUN;
PROC CONTENTS DATA=WORK.IMPORT;
RUN;
%web_open_table(WORK.IMPORT);
```

When SAS code is run, it produces a **log file**. In current SAS applications, the log file opens in a separate window. The log file repeats the code that has been run and includes messages providing feedback, including error messages. It is important to always review the log file to make sure errors and key warnings do not exist, as well as to confirm any assumptions by SAS. Using the point-and-click GUI in SAS University Edition, SAS will create the preceding code and then run it to import the `Chap 1_1_Infile.csv file`. This is evident because this is the code that is displayed in the log file.

Notice that the code refers to the following items:

- An external reference file using the `REFFILE` command
- A connection between the data file being created and the reference file through `DATAFILE = REFFILE`
- The specification that the input file is a **comma-separated values (CSV)** file through `DBMS = CSV`
- The `OUT` specification to make SAS output the resulting dataset named `IMPORT` into `WORK` through `OUT = WORK.IMPORT`
- The automatic placement of a `PROC CONTENTS` command to display the contents of the dataset

`PROC IMPORT` code like the preceding code does not provide the opportunity for a programmer to specify details about how they want the resulting dataset formatted, column by column. This type of specification can be achieved using a series of commands.

Let's consider the source file, which is `Chap 1_1_Infile.csv`. This source file has the rows from the BRFSS 2018 dataset for FL, MA, and MN (`_STATE` equals `12`, `25`, or `27`). It also has these columns: `_STATE`, `SEX1`, `_AGE80`, `and _BMI5`, which is the respondent's body mass index (BMI) stored as a four-position integer that should have a decimal placed between the second and third integer. The following table displays three records from the source data:

| _STATE | SEX1 | _AGE80 | _BMI5 |
|---|---|---|---|
| 12 | 2 | 76 | 3175 |
| 25 | 1 | 67 | 2525 |
| 27 | 1 | 43 | 2999 |

Table 1.2 – Example of four variables from three records of BRFSS source data

Using the following code, we can read in the `*.csv` file and specify details about the column formats:

```
data Chap_1_1_Infile;
%let _EFIERR_ = 0;
infile '/folders/myfolders/Chap 1_1_Infile.csv' delimiter = ','
firstobs=2
     MISSOVER DSD lrecl=32767;
     informat    _STATE 2.;
     informat    SEX1 2.;
     informat    _AGE80 2.;
     informat    _BMI5 2.2;
     format      _STATE 2.;
     format      SEX1 2.;
     format      _AGE80 2.;
     format      _BMI5 4.1;
input
     _STATE
     SEX1
     _AGE80
     _BMI5
;
if _ERROR_ then call symputx('_EFIERR_',1);
RUN;
```

The preceding code provides an opportunity to look at various aspects of the way SAS reads in data using the `INFILE` statement:

- The code opens with a data step specifying the output file to be named `Chap_1_1_Infile` and placed in `WORK`.
- The `%let _EFIERR_ = 0;` and `if _ERROR_ then call symputx('_EFIERR_',1);` commands are used for error handling.
- The `INFILE` command has many options that can be used. The preceding code uses `delimiter = ','` to indicate that the source file is comma-delimited, and `firstobs=2` to indicate that the values in the first observation are on row `2` (as the column names are in row `1`).
- The `INFORMAT` command provides the opportunity to specify the format of the source data being read in. This is important to make sure data is read correctly without losing any information. Notice how `_BMI5` is specified at `2.2`, meaning two numbers before the decimal, and two numbers after it.
- The `FORMAT` command allows the ability to specify the format of the data output into the `Chap_1_1_Infile file`. Note that formats do not change how the data is stored, but only control how they are displayed. This can be confusing to new SAS programmers. Notice how `_BMI5` is specified at `4.1`, so it should result in a variable with four numbers before the decimal and one number after it.
- As with when we used `CARDS` and `DATALINES`, the `INPUT` statement signals the point where SAS should start reading in the data, and names each column in order.

The resulting dataset in `WORK`, named `Chap_1_1_Infile`, looks like this:

| _STATE | SEX1 | _AGE80 | _BMI5 |
|---|---|---|---|
| 12 | 2 | 76 | 31.8 |
| 25 | 1 | 67 | 25.2 |
| 27 | 1 | 43 | 30.0 |

Table 1.3 – Example of the same source data formatted using the FORMAT command

The ability to specify details about importing data was necessary for SAS users to be able to read **flat files** that were exported out of another system. The `INFILE` approach with `FORMAT` and `INFORMAT` allowed the necessary flexibility in programming to allow conditionals to be placed in code to facilitate SAS reading only parts of the files, and the ability to direct SAS to specific coordinates on raw datafiles and direct it to read those values a certain way. But while using `INFILE` and related commands increased the flexibility behind the use of big data in SAS (because the data step functioning was still based on the sequential read approach used with the punch cards), there were limited opportunities for the programmer to improve I/O.

# Improving I/O

Although SAS has created many features to improve data warehousing, it is still necessary to improve I/O through the strategic use of SAS code. This section will cover the following:

- Features for warehousing that have been developed by SAS
- The importance of using the `WHERE` rather than the `IF` clause in data processing
- How sorting and indexing can be done to improve I/O in SAS

## Developing warehouse environments

The 1990s saw people working with SAS and big data to find creative solutions to improve data I/O. In his 1997 SAS white paper (available under *Further reading*), Ian Robertson describes the benefits of his case study migrating the **Wisconsin Department of Transportation Traffic Safety and Record-keeping System (TSRS)** from a mainframe SAS setup to one where data was served up to analysts through a **local area network (LAN)**.

By this time, SAS had been reconfigured to run on their LAN's operating system, OS/2, so his team was able to save processing costs by moving the SAS analysis and reporting functions away from the mainframe and onto the PCs of the analysts. One of the innovations that enabled this was **SAS/Connect**, a component that allowed local PCs to connect to a mainframe storing data elsewhere. Using SAS/Connect, Robertson's team downloaded the data onto their LAN, making a local copy for analysis. Over a 2-day period, they were able to transfer 1.6 GB of TSRS data from the years 1988 through 1995 and 30 MB of source code from the mainframe storage system to the LAN.

In SAS's history of data warehousing mentioned on their website, the differences between a **data warehouse, data mart**, and **data lake** are explained:

- A **data warehouse** stores a large amount of enterprise data covering many topics.
- A **data mart** stores a smaller amount of data focused on one topic, usually sourced in a data warehouse.
- **Data warehouses** and **data marts** consist of raw datasets that have been restructured for the purposes of analysis and reporting.
- By contrast, a **data lake** stores a large amount of raw data that has not been processed.

Robertson's local version of the TSRS could be seen as a data warehouse, in that all the data from the source system had been moved to the LAN. By moving the data from the mainframe to the LAN, the data was now in the same physical place as the application accessing it, and this reduced not only I/O time but CPU cost. However, the group encountered a few difficulties after moving the data that required revising their source code.

## Using the WHERE clause

Robertson's team used a few different strategies to improve the efficiency of their source code as an approach to **performance tuning**. Robertson's team reviewed their SAS processing code and found that they were using a lot of `WHERE` clauses to subset data into regions. They began looking into ways to improve the efficiency of their coding with respect to the use of `WHERE` clauses.

The purpose of the `WHERE` clause is to subset datasets. The `WHERE` clause becomes important in data warehousing in two major areas: data management and data reporting. For this reason, `WHERE` is used in both data steps and PROCs.

Consider our example dataset, `Chap_1_1_Infile`, and imagine we wanted to create a dataset of just the Massachusetts records (`_STATE = 25`). Even though the source dataset, `Chap_1_1_Infile`, has 38,901 observations, by declaring a `WHERE` clause in our data step, we can avoid reading all those records and only process the ones where `_STATE = 25` comes into the data step:

```
DATA Massachusetts;
     set Chap_1_1_Infile;
     WHERE _STATE = 25;
RUN;
```

The first line of the log file says this:

```
NOTE: There were 6669 observations read from the data set WORK.
CHAP_1_1_INFILE.
       WHERE _STATE=25;
```

This indicates that as SAS was processing the file, it skipped over reading the rows where it saw anything other than `25` in `_STATE`. On my computer, the log file said this operation only took 0.01 seconds of CPU time. However, at the time Robertson was writing, the difference between using and not using a `WHERE` clause could really impact CPU time, which drove up data warehousing costs.

`WHERE` can also speed up the reporting of large datasets. Consider the use of `PROC FREQ` on our example dataset, `Chap_1_1_Infile`, to get the frequency of the gender variable, `SEX1`, in Massachusetts (`_STATE=25`):

```
PROC FREQ data=Chap_1_1_Infile;
     Where _STATE = 25;
     Tables(SEX1);
RUN;
```

Looking at the log file, again, we see that this frequency calculation only considered the 6,669 records from Massachusetts in its processing, thus saving processing time by not considering the whole file. On my computer, the log file says that the CPU time used was 0.10 seconds – still a very small number, but 10 times the number seen in the data step processing. This demonstrates the extra processing power needed for the frequency calculations produced by `PROC FREQ`.

## Using IF compared to WHERE

In 2003, Nancy Croonen and Henri Theuwissen published a white paper providing tips on reducing CPU time through more efficient data step and PROC programming (available under *Further reading*), as many SAS users were still operating in a mainframe environment. In addition to advocating the use of the `WHERE` clause, the authors did studies to compare the CPU time for different ways of accomplishing the same tasks in SAS. They discussed trade-offs between using the `WHERE` and `IF` clauses.

Earlier, we used `WHERE` in a data step to subset the `Chap_1_1_Infile` dataset to just the records from Massachusetts and named the dataset `Massachusetts`. Let's do the same thing again, but this time, we'll use `IF` instead of `WHERE`:

```
DATA Massachusetts;
     set Chap_1_1_Infile;
     IF _STATE = 25;
RUN;
```

On my computer, I notice no difference in processing time – it is still 0.01 seconds. However, the first line of the log file is different:

```
NOTE: There were 38901 observations read from the data set
WORK.CHAP_1_1_INFILE.
```

This indicates that SAS read in all 38,901 records before processing the rest of the code, which was instructions to only keep records in the `Massachusetts` dataset `IF _STATE = 25`. From this demonstration, it seems like `WHERE` is superior to `IF` when trying to make efficient code.

However, this is not always true. There is only a meaningful reduction in processing time by using `WHERE` instead of `IF` in case the variable in the statement has many different values, otherwise known as **high cardinality**. Croonen and Theuwissen show an example comparing the use of `WHERE` and `IF` to subset a large dataset by a nominal variable with seven levels, which would have **low cardinality**. In their example, there was only a small reduction in processing time using `WHERE` compared to `IF`.

> **Note:**
>
> When the `WHERE` clause is used, SAS creates a simple index on that variable and searches the index. When the `IF` clause is used, SAS does a line-by-line sequential search of the dataset. Note that `IF` must be used when applied to temporary variables not in the dataset. For a deeper discussion on the use of the `WHERE` and the `IF` clauses, please see the SAS white paper by Sunil Gupta (under *Further reading*).

Robertson's team managing the TSRS data warehouse had already optimized their code using `WHERE` for subsetting in their data processing and reporting. However, they found that after they moved their data to a local LAN, they needed to further improve the efficiency of their code, so they started looking into approaches to **indexing**.

## Sorting in SAS

An **index** is a separate file from a dataset that is like an address book for SAS to use when looking up records in a large dataset. Unlike **structured query language (SQL)**, SAS does not create indexes automatically in processing. There are ways the programmer can index variables in a dataset as well. Given certain data processing code, placing indexes on particular variables can speed up processing.

As discussed earlier, it is helpful to identify what variables are used in `WHERE` clauses that could benefit from indexing, and also, whether they are high or low cardinality. If a certain high-cardinality variable is used repeatedly in a `WHERE` clause, it is a good candidate for an index.

A **simple index** is an index made of one variable (such as _STATE). A **composite index** is one made of two or more variables (such as _STATE plus SEX1). Because SAS processes records sequentially, the easiest way a programmer can simulate an index on a SAS dataset is by sorting the dataset by that variable. It is not unusual for SAS datasets in a data lake to be stored sorted by a particular high-cardinality variable (such as a unique identification number of the row).

According to the BRFSS 2018 Codebook, in the source dataset, 53 regions are represented under the _STATE variable. If this variable was used in warehouse processing, then it would be logical to store the dataset sorted by _STATE, as shown in the following code:

```
PROC SORT data=Chap_1_1_Infile;
     by _STATE;
RUN;
```

Sorting itself takes some time; on my computer, it took 0.05 seconds of CPU time. Compared to the 0.01 seconds it took to read in the dataset of about 36,000 rows and the 0.10 seconds it took to do a PROC FREQ, this shows that if there is to be a policy in the data warehouse that datasets are to be stored sorted by a particular variable, sorting them will take some time to execute.

While sorting is a simple way of placing an index on a SAS variable, it may not be adequate. Using our example dataset, imagine we wanted to know the mean age (_AGE80) of respondents by gender (SEX1). We could use PROC MEANS to do this using the following code:

```
PROC MEANS data=Chap_1_1_Infile;
     by SEX1;
RUN;
```

With our dataset, Chap_1_1_Infile, in the state it is in at the beginning of the PROC, meaning sorted by the _STATE variable and not the SEX1 variable, the preceding PROC MEANS code will not run. The code produces the following error:

```
ERROR: Data set WORK.CHAP_1_1_INFILE is not sorted in ascending
sequence. The current BY group has SEX1 = 9 and the next BY
group has SEX1 = 1.
```

As described in the error wording, SAS is expecting the dataset to already be sorted by the `BY` variable, which is `SEX1` in the `PROC MEANS` code. Solving this problem can easily be accomplished by resorting the dataset on the `SEX1` variable prior to running the preceding code. But this will cause the dataset to no longer be sorted by `_STATE`, and we will lose the benefit of being able to efficiently use `_STATE` in a `WHERE` clause. It is in these more complex situations that we cannot rely on using sorting for a simple index, and should consider placing an index on certain variables.

## Setting indexes on variables

Robertson's team ran into a similar problem, where different variables were used in `WHERE` clauses throughout their programming. Therefore, they could not simply sort their datasets by one variable and rely on that indexed variable to speed up their processing.

One of the ways they dealt with this was to deliberately place **indexes** on certain SAS variables using a method other than sorting. As described earlier with sorting, indexes can help improve SAS's performance when extracting a small subset of data from a larger dataset. According to Michael Raithel, who wrote a SAS white paper about indexing, if the subset comprises up to 20% of the dataset, then an index should improve program performance (white paper available under *Further reading*). But if it is larger, it may not impact or even worsen performance. We saw this situation earlier when comparing the efficiency of processing between using a `WHERE` versus an `IF` clause.

SAS continued to release new enterprise versions with upgraded PROCs and data steps and new functionality. Starting in the 1980s, main upgrade versions were released and stated as an integer (for example, version 6), but in reality, these versions were upgraded regularly, with each upgrade designated by two digits after the decimal, (for example, version 6.01). SAS had tried to build indexing features into its version 6 but found that there were performance problems, according to Diane Olson's white paper on the topic (available under *Further reading*). The version 7 releases, otherwise known as the Nashville releases and first available in 1998, fixed these problems.

Let's create an index on the `_STATE` variable using our example dataset, `Chap_1_1_Infile`. One way we could have created an index on `_STATE` was in the original data step we used to read in the data. Notice the same code we used before follows, but with the addition of the `index` command in the first line of the data step:

```
data Chap_1_1_Infile (index=(_STATE));
```

This is often the most efficient way to place an index, but datasets that have already been read into WORK can have indexes set on a variable using various approaches. One way is to use PROC DATASETS, which is demonstrated here:

```
PROC DATASETS nolist;
     modify Chap_1_1_Infile;
           index create _STATE;
RUN;
```

The nolist option suppresses the printout of the dataset, and the modify statement is used to tell SAS to modify the Chap_1_1_Infile dataset to create an index on the _STATE variable. In both of these examples, a simple index was created. Imagine we wanted to create a composite index including both _STATE and SEX1. We could do that using a data step, or we could do it using PROC DATASETS.

Using a data step, we could set the composite index by replacing the first line of our data step code shown earlier with this code:

```
data Chap_1_1_Infile (index=(STATE_SEX = (_STATE SEX1)));
```

Notice the differences between when we set a simple index on _STATE in the data step and the preceding code:

- Because we are setting a composite index, we have to actually name the index a name that is different than the variables in the dataset. We are using the name STATE_SEX for the index.
- Then, in parentheses, we specify – in order – the two variables in the composite index, which are _STATE and SEX1.

> **Note:**
> Some analysts prefer to use the _IDX suffix when naming indexes to indicate they are indexes.

To create the same index using PROC DATASETS, we would use the same code as we did for our simple index on _STATE, only replacing the index create line with this code:

```
index create STATE_SEX = (_STATE SEX1);
```

In the development of the TSRS data warehouse, Robertson's team leveraged indexes in their performance tuning. First, since indexes are used in `WHERE` clauses and not `IF` clauses, they rewrote their code to strategically switch `IF` clauses with `WHERE` clauses to improve performance. Then, they set indexes on the variables that were used in `WHERE` clauses, and only saw a 6% storage overhead.

# Dealing with storage and memory issues

This section will cover issues with storage and memory when using SAS for big data. It will cover the following:

- How SAS dealt with competition from **structured query language (SQL)** for data storage
- How `PROC SQL` works and can be used in data warehouse processing
- Considerations about memory and storage that need to be made when using SAS in a data warehouse in modern times
- How SAS can work in the cloud

## Avoiding memory issues

Even as SAS got more powerful, datasets kept getting bigger, and there were always challenges with running out of memory during processing. For example, using `WHERE` instead of `IF` when reading in data would not only reduce CPU usage and the time it took for code to run, it would also prevent unnecessary usage of memory. Even today, tuning SAS code may be necessary to avoid memory issues.

In a data warehouse, mart, or lake, datasets that were transformed in SAS may be stored outside of SAS in SAS format. This makes them easy to read into SAS. However, this format can be very large, so the option to `COMPRESS` the dataset was created. Curtis Smith reported on his test compressing SAS files in his white paper (available under *Further reading*), and found that depending upon the dataset, compressing datasets could make them take up half the space.

Smith recommended not only compressing datasets but also deleting unneeded variables to make datasets smaller. In a data warehouse, mart, or lake, source datasets contain **native variables**. In a data lake, these datasets may remain relatively unprocessed. However, in a data warehouse or data mart, decisions need to be made about what variables to keep available for analysis in the warehouse. Further more, **transformed variables** may be added during processing to serve the needs of the users of the warehouse.

The team running the data warehouse should ask the following for each raw dataset:

- If native variables should be available for analysis, which ones should be kept?
- If transformed variables should be available for analysis, which ones should be provided?

By carefully answering these questions, only the columns needed from each dataset can be retained in analysis files, thus reducing processing time for warehouse developers and users.

## Accommodating Structured Query Language

SQL was developed and deployed by various companies in the 1990s and early 2000s. SQL was aimed at data maintenance and storage using **relational** tables rather than flat files. SQL approaches only became possible in the 1990s due to upgrades in technology that allowed faster processing of data.

SQL languages accomplish the same data editing tasks that data steps do in SAS, but they use a different approach. Unlike SAS, which is a **procedural language**, SQL is a **declarative language**:

- **In SAS**, the programmer must program a data step to do the *procedures* in the most efficient way to optimize data handling.
- **In SQL**, the programmer *declares* what query output they desire using easy-to-understand, simple English statements, and an optimization program (or **optimizer**) running in the background figures out the most efficient way to execute the query.

While using efficient code in SQL can still improve performance, the efficiency of SQL is less dependent upon the programmer's code and more dependent on the function of its optimizer. Hence, maintaining data in a database became easier using SQL rather than SAS data steps. In SQL, programmers had to learn a few basic commands that could perform a variety of tasks when used together. But with SAS data steps, programmers needed to study a broad set of commands, and they also had to learn the most efficient way to assemble those commands together in order to achieve optimal SAS performance.

What SQL cannot do is analyze data the way SAS can. Therefore, over the latter half of the 1990s and early 2000s, while many databases began to be stored and maintained in SQL, SAS could still be used on them for analysis through the **SAS/Access** feature.

A 1995 edition of the periodical Computerworld described current options for SAS users in an article titled *SAS Institute's customers keep the faith* by Rosemary Cafasso (available under *Further reading*). There were two ways to conceive of data storage in SAS at that time:

- Using SAS only for analysis, and connecting to a non-SAS data storage system to do this
- Using SAS for both data storage and analysis

For the first option, SAS/Access features could be used. For the second option, **SAS/Application Facility (SAS/AF)** was used to create a client/server environment to support both data storage and analysis in SAS. Another term for this setup is **server SAS** (as opposed to **PC SAS**, which is an application that runs entirely on a PC without a client/server relationship). The advantage of using SAS/AF is that a comprehensive SAS solution could be used that optimized the client/server relationship (through, for example, partitioning the application so it ran on different processors).

Also, in 1995, SQL optimizers had not been improved to the point where they outperformed SAS data steps, so at that time SAS/AF was a better approach than SAS/Access to connect to a SQL database. As noted in the Computerworld article, this led programmers to gravitate toward working either entirely in a SAS environment, or entirely outside of one.

With the visualization tools included in SAS/AF, SAS was now competing with visualization applications as well as data management applications. SAS's users continued to rate it highly, and were very loyal, as moving away from the SAS/AF platform would be very difficult given its dissimilarity to other applications.

## Using PROC SQL

SAS's trajectory in general through its release of version 8 in 1999 and later version 9 (the current one) in 2002 has been to build extra functions into its core analysis products, and to also design supporting products to support its functionality. Unlike in the early years, SAS has not revisited data step functioning, nor considered redeveloping its data step language as declarative rather than procedural.

Through the late 1990s and early 2000s, SQL became more predominant, and therefore more programmers were trained in SQL. These SQL programmers had a lot of trouble transferring their skills to use in SAS data steps, so SAS developed a SQL language within SAS called `PROC SQL`.

PROC SQL has the following features:

- It is a language within SAS, in that PROC SQL code starts with a PROC SQL statement and ends with a quit statement.
- It uses SQL commands, such as CREATE TABLE and SELECT with GROUP BY and WHERE.
- It allows the user to control its use of processors during execution through the THREADED option.
- It includes a WHERE clause and other clauses that use indexes if they are available.
- Unlike other SQLs, it does not have an independent optimizer program, so creating optimized code is important.

Like SQL, PROC SQL is much easier to use than data step language for a few common tasks. One particularly useful task that is much easier in PROC SQL is creating a VIEW of the data, which allows the user to look at a particular section of the dataset.

Imagine we wanted to view the data in our example dataset, Chap_1_1_Infile, but we only wanted to look at the data for women (SEX1 = 2) who live in Massachusetts (_STATE = 25). We could use this PROC SQL code:

```
PROC SQL;
Select * from Chap_1_1_Infile
     where SEX1 = 2 and _STATE = 25;
quit;
```

This code produces output in the following structure (with just the first three rows provided):

| _STATE | SEX1 | _AGE80 | _BMI5 |
|---|---|---|---|
| 25 | 2 | 43 | 36.9 |
| 25 | 2 | 53 | 38.1 |
| 25 | 2 | 66 | 41.6 |

Table 1.4 – Output from PROC SQL

To get similar output using SAS commands, the following PROC PRINT code could be used. Note that all variables in the order stored in the dataset are displayed since the VAR statement is excluded:

```
PROC PRINT DATA=Chap_1_1_Infile;
where SEX1 = 2 and _STATE = 25;
RUN;
```

But imagine we did not want to return all the variables – assume we only wanted to return age (_AGE80) and BMI (_BMI5). We could easily replace the asterisk in our PROC SQL code to specify only those two columns:

```
PROC SQL;
Select _AGE80, _BMI5 from Chap_1_1_Infile
     where SEX1 = 2 and _STATE = 25;
quit;
```

In PROC PRINT, to achieve the same output, we would add a VAR statement to our previous code:

```
PROC PRINT DATA=Chap_1_1_Infile;
where SEX1 = 2 and _STATE = 25;
var _AGE80 _BMI5;
RUN;
```

Even in this short example, it is easy to see how SAS PROCs and data steps are more complicated than SQL commands because SQL has fewer, more modular commands. By contrast, SAS has an extensive toolset of commands and options that, when understood and used wisely, can achieve just about any result with big data.

## Using SAS today in a warehouse environment

While PROC SQL appears to be a workaround from learning complicated data step language, this is not the case in data warehousing. Because of the lack of optimization of PROC SQL, in many environments, it is very slow and can only be feasibly used with smaller datasets. Even today, when transforming big data in SAS, in most environments, it is necessary to use data step language, and this affords the programmer an opportunity to develop optimized code, as efficiency is always necessary when dealing with data in SAS.

However, when interfacing with another **database management system (DBMS)** where native data are stored in SQL, SAS `PROC SQL` might be more useful. In his recent white paper on working with big data in SAS, Mark Jordan describes various modern approaches to improving the processing efficiency of both `PROC SQL` and SAS data steps in both server SAS environments, as well as environments where SAS is used as the analysis engine and connects to a non-SAS DBMS through SAS/Access.

Jordan describes two scenarios for big data storage and SAS:

- **Using a modern server SAS set up**: Server SAS comes with its own OS, and Base SAS version 9.4 includes its own DS2 programming language. These can be used together to create threaded processing that can optimize data retrieval.
- **Using SAS for analysis connected to non-SAS data storage**: In this setup, SAS/Access is used to connect to a non-SAS DBMS and pull data for analysis into the SAS application. This can create a lag, but if SAS and the DBMS are co-located together and the DBMS can use parallel processing, speed can be achieved.

Ultimately, the main bottleneck in SAS processing has to do with I/O, so the easier it is for the SAS analytic engine to interact with the stored data, the faster processing will go. But even in this modern era, limitations surrounding data I/O continue to force SAS users to develop efficient code.

Jordan provides the following tips for thinking about coding for a SAS data warehouse:

- Use `WHERE` instead of `IF` wherever possible (due to its increased processing efficiency).
- As stated earlier, reduce columns retained to just the native and transformed variables needed in the warehouse.
- Using the options `SASTRACE` and `SASTRACELOC` will echo all the SQL generated to the SAS log file, which can be useful for performance tuning.
- Use `PROC SQL` and data steps to do the same tasks, and then compare their processing time using information from the SAS log to choose the most efficient code.
- It is especially helpful to compare `PROC SQL` code performance on summary tasks, such as developing a report of order summaries, because `PROC SQL` may perform better than PROCs or data steps.

- If using a server SAS setup with DS2 and data steps and if the log from your data steps shows a CPU time close to the program runtime, then your data steps are **CPU-bound**. In those cases, rewriting the data step process in DS2 could be helpful because it could take advantage of threaded processing.
- DS2 has another advantage as it is able to develop results at a higher precision level than data steps.
- DS2 code uses different commands than data step code but can achieve the same results.
- On **massively parallel processing (MPP)** DBMS platforms such as Teradata and Hadoop, DS2 can run as an in-database process using the **SAS In-Database Code Accelerator**. Using this code accelerator can significantly improve the efficiency of data throughput in these environments.

> **Note:**
> In his white paper, Mark Jordan compared `PROC SQL` processing using the `SCAN` command compared to the `LIKE` command for retrieving a record with criteria set on a high-cardinality variable and found the `LIKE` command to be more efficient.

## Using SAS in the cloud

In his white paper, Jordan also describes how SAS now has a new **Viya architecture** that offers **cloud analytic services (CAS)**. A CAS library allows the following capabilities:

- Fast-loading data into memory
- Conducting distributed processing across multiple nodes
- Retaining data in memory for use by other processes until deliberately saved to disk

A CAS library has **application programming interfaces (APIs)** that allow actions to be executed from a variety of languages, including **Java**, **Python**, and **R**, and of course, the SAS Version 9.4 client application.

Today, not all warehouse data is stored in the cloud, and many datasets are still stored on traditional servers. Jordan recommended that if the user has an installation of the SAS 9.4M5 application and has access to SAS Viya CAS, and they want to decide whether or not to move to CAS from a traditional server, they should compare the processing time on a subset of data in both environments. Jordan was able to demonstrate cutting the processing time from over 1 minute to 2.35 seconds by moving his data from a traditional server to SAS Viya CAS.

# Using SAS in modern warehousing

Today, SAS data warehousing is more complicated than it was in the past because there are so many options. Learning about these options can help the user envision the possibilities, and design a SAS data warehousing system that is appropriate for their organization's needs. This section will cover the following:

- A modern case study that used SAS components for analyzing unstructured text in helpdesk tickets
- A case study of a data SAS warehouse that upgraded an old system to include a new API allowing users more visualization functionality through **SAS Visual Analytics**
- A case study of a legacy SAS shop that began to incorporate R into their system
- A review of how SAS connects with a new cloud storage system, Snowflake

## Warehousing unstructured text

In his white paper on warehousing unstructured text in SAS, Nick Evangelopoulos describes how the **IT Shared Services** (**ITSS**) division at the **University of North Texas** (**UNT**) used SAS to study their service tickets to try to improve services (link available under *Further reading*). Here are the steps they took:

- They chose to study a set of 9,691 tickets (representing approximately 18 months' worth of tickets) comprised mainly of unstructured text from the native application platform ServiceNow.
- Using the open source statistical application R, they conducted text cleaning. Mostly, this consisted of removing back-and-forth conversations by email that were recorded in the unstructured ticket text.
- Using the text mining component of SAS called **SAS Text Miner** (used within the SAS platform **SAS Enterprise Miner (SAS EM)**), they were able to use text extraction to help classify the tickets by topic.
- Next, the team used Base SAS and the analytics component **SAS STAT** to add indicator variables and other quantitative variables to the topics, thus creating a quantitative dataset that could be analyzed and visualized.

After doing this, the team wondered if SAS EM would classify the tickets under the same topic as the user entering the ticket would. To answer this question, the team analyzed 1,481 new tickets that were classified using SAS EM as well as being classified by the user. They found dramatic differences between how users and SAS EM classified the tickets, suggesting that this classification may need additional development in order to be useful.

## Using SAS components for warehousing

A white paper by Li-Hui Chen and Manuel Figallo describes a modern SAS data warehouse using SAS applications (available under *Further reading*). The **US Department of Health and Human Services** (**DHHS**) has a data warehouse of health indicators called the **Health Indicators Warehouse** (**HIW**). They described how they upgraded their SAS data warehouse system to improve performance and customer service using **SAS Visual Analytics (VA)** accessed through an API.

The HIW serves many users over the internet. Prior to the upgrade, SAS datasets were accessed from storage using SAS, and **extract-transform-load (ETL) processes** needed to take place manually on the data before it could be visualized. This made the data in the warehouse difficult to visualize.

With the upgrade, this is the new process:

1. Users obtain permission to access the API, which controls access to the underlying data as well as the VA capabilities.
2. Using the API, which contains a GUI, users indicate which health indicator they want to extract from the HIW, and how they want to visualize it.
3. The API extracts the necessary data from the HIW data store using automated processing.
4. The API performs necessary ETL processes to support visualization.
5. The API then visualizes the results using VA.

Here is a conceptual diagram of the old and new systems:

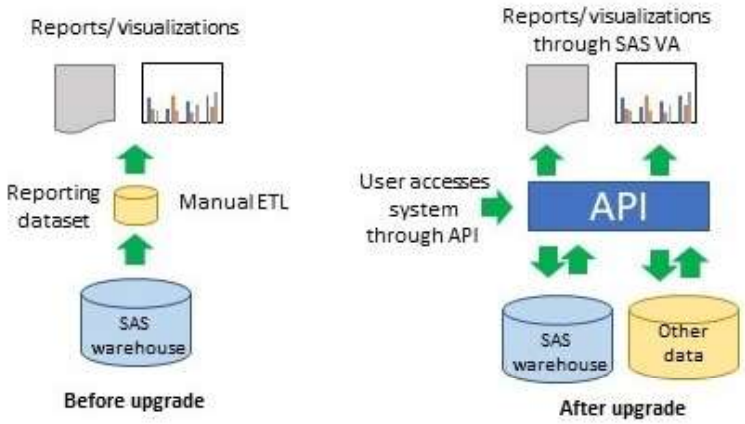


Figure 1.3 – SAS warehousing system before and after adding an API layer

Focusing on I/O, the authors pointed out that ETL in the API is achieved by running SAS **macros**, or code routines developed in the **SAS macro language** that can take user or system inputs and can be run automatically. They pointed out that they can run these macros either through a **stored process** (where the macro can be run on one dataset at a time) or a **batched process** (where the macro is run on several datasets at once). The authors found that they needed to use a batch process when transferring large amounts of HIW data through an API call.

## Using other applications with SAS

SAS has been around a long time and has typically been the first choice for warehousing big data. However, since the invention and rise of SQL, there has been competition between SAS and SQL for data storage functions. With the rise of R, open source statistical software known for visualization and an easy web interface, SAS has seen competition with respect to statistical analysis functions.

Over time, SAS responded to competition by building in extra functionality. SAS/Access, SAS VA, and SAS Viya are all examples of this. However, the reality is that SAS is best at analytics, so other applications tend to be superior at these other functions. This has created challenges for legacy SAS warehouses that are now rethinking how they use SAS in their system. Teams are approaching this challenge with a variety of responses.

Dr. Elizabeth Atkinson shared her team's story of moving from a 100% SAS shop to incorporating R for some functions. She leads a biostatistics service at the Mayo Clinic, a famous specialty clinic in the US, which has been a SAS shop since 1974, when punch cards were still being used, and now has a staff of 300 in 3 locations. The service supports data storage and analysis for studies, both large and small.

In 2014, Mayo went to negotiate their SAS license and found that the price had increased significantly. SAS has always been a distinctive product with a high price. According to the Computerworld article, in 1995, a full SAS application development package, when bundled for 10 users, cost $1,575 per seat; this is expensive even by today's standards. However, in 2014, the increase in cost was felt to be unsustainable, and the Mayo team started looking for other options.

They wanted to decrease their dependence on SAS by moving some of their functions to R, and also improving their customer service and satisfaction. They faced the following challenges:

- **SAS infrastructure was entrenched**: All of the training was based on SAS, SAS was integrated into every workflow, and automation used SAS macros. Many users only trusted SAS and did not trust numbers coming out of R. SAS users relied on their personal code repositories.

- **R infrastructure was dynamic**: Unlike SAS, R releases new versions often. R innovates quickly, so it is hard to keep up a stable R environment. **R packages**, which are external components of Base R that can be added, were also upgraded regularly, leading to code that would break without warning, and cause user confusion.
- **Time constraints**: Reworking some SAS functions to be done by R required a lot of effort of deconstructing SAS and constructing R. Both leaders and users had time constraints.
- **Different learning styles and levels of knowledge**: SAS users had spent years learning data steps. R data management is completely different. It was hard for SAS users to learn R, and R users to learn SAS.
- **R support needed**: SAS provides customer support, but that is not available with open source software like R. The organization needed to build its own R support desk. Compared to SAS, R's documentation is less standardized and comprehensive.

To integrate R into their shop, they took the following steps:

- **Committed funding**: Divisional funding was committed to the project.
- **Identified R champions**: This was a group of R users with expertise in R and SAS.
- **Set up an R server**: Having an R server available increased enthusiasm and interest in R.
- **Rebuilt popular local SAS macros in R**: These are the ones that were deconstructed and rebuilt in R. Many of these were for reporting. They took the opportunity to improve reporting when rebuilding these macros.
- **Developed integrated SAS and R training**: Because they are now a combined shop, their training shows how to do the same tasks in SAS and R. They also hold events demonstrating R and providing online examples.
- **Set up an R helpdesk**: This provides on-call, in-house R support. They maintain a distribution list and send out R tips.

Even after offering R as an alternative, many users chose to stay with SAS. The reasons the shop could not completely convert from R to SAS include the following:

- **Time and cost constraints**: It was not possible to move all the small projects already in SAS over to R.
- **Data retrieval and ETL**: R cannot handle big data like SAS. The SAS data steps provide the ability to control procedural data processing in SAS, and this is not possible in R.

- **Analysis limitations**: Certain tasks are much clumsier in R than in SAS. At the Mayo Clinic, they found that mixed effect models were much more challenging in R than in SAS.

One of the overall benefits of this effort was that it opened the larger conversation behind what skills will be needed among analysts in the division in the future. These considerations run parallel to the consideration as to what SAS and non-SAS components will be used in the data system in the near future, what roles they will play, how they will be supported, and how they will work together to improve the user experience.

## Connecting to Snowflake

As data gets bigger and bigger, new solutions have been developed to store data in the cloud. Microsoft Azure and **Amazon Web Services (AWS)** are cloud services to help move business operations to the cloud. **Snowflake** (`https://www.snowflake.com/`) is a relatively new cloud data platform that runs on Microsoft Azure and AWS and may run on other cloud services in the future.

Snowflake enables a programmer to make a virtual data warehouse with little cost, thus solving a data storage problem. However, data still needs to be accessed to be analyzed. Therefore, SAS upgraded its SAS/Access component to now be able to connect directly to Snowflake.

SAS documentation about connecting to Snowflake indicates that Snowflake uses SQL as its query language. Both `PROC SQL` and regular SAS functions can be passed to Snowflake, but there are cases where SAS and Snowflake function names conflict. Further more, careful settings of options and code tuning are needed to improve I/O from SAS to Snowflake.

Although products like Snowflake can solve the big data storage problem, the issue with SAS will always be I/O. Using the newest and most appropriate technology along with the most efficient coding approaches will always be the best strategy for dealing with the data warehousing of big data in SAS.

# Summary

This chapter provided a short history of SAS, focusing on how it has been used for data storage and analysis over the years. Initially, SAS data was stored on punch cards. Once data became electronic, the main challenge to SAS users working with big data was I/O. As SAS environments evolved from being on mainframes to being accessible by PCs, SAS developed new products and services to complement its core analytics and data management functions.

SAS data steps are procedural, and allow the programmer opportunities to greatly improve I/O through the use of certain commands, features, and approaches to programming. When SQL became popular, `PROC SQL` was invented. This allowed SAS users to choose between using data steps or SQL commands when managing data in SAS.

Today, SAS is still used in data warehousing, but there are new challenges with accessing data in the cloud. SAS data warehouses today can include predominantly SAS components, such as SAS VA and CAS. Or, SAS can be part of a warehouse system that includes other components and applications, such as cloud storage in Snowflake, and supplemental analytic functions provided by R.

Modern SAS data warehousing still seeks to improve I/O and to better serve warehouse users through the development of an efficient system that meets customer needs. Creativity is required in the design of modern SAS data warehouses so that the system can leverage the best SAS has to offer while avoiding its pitfalls.

Although this chapter covers the entire history of SAS for data storage, it is important for the new data scientist to understand this information because the way SAS runs today can often be explained by certain events in its history. Particular terminology and features that are unique to SAS arise from how it has evolved over time, and it is helpful to know this background when communicating with today's SAS data warehouse developers and data scientists.

The next chapter takes a sharp focus on the act of reading data into SAS and will close with strategies that can be used when importing difficult data into SAS.

# Questions

1. What is the difference between SAS and SQL with respect to data handling?
2. What is the difference between subsetting datasets using `WHERE` compared to the `IF` clause?
3. What is the component of SAS that allows it to connect to non-SAS databases?
4. Under what circumstance should you place an index on a variable in a large dataset?
5. Should you use SAS to enter a small dataset through data steps? State the reason for your answer.
6. What is the main advantage of using all SAS components in your warehouse?
7. What is a good way to decide whether to use a data step or `PROC SQL` for a particular data editing task?

# Further reading

- BRFSS reference: Centers for Disease Control and Prevention (CDC). Behavioral Risk Factor Surveillance System Survey Data. Atlanta, Georgia: U.S. Department of Health and Human Services, Centers for Disease Control and Prevention, 2020.
- *Introduction to ODS Graphics for the Non-statistician*, SAS white paper by Mike Kalt and Cynthia Zender – available at `https://support.sas.com/resources/papers/proceedings11/294-2011.pdf`
- *Programming with Punch Cards* by Dale Fisk – report available at `http://www.columbia.edu/cu/computinghistory/fisk.pdf`
- *How to save $30,000 in 4 Hours": Migrating SAS® systems from the mainframe to the PC* SAS white paper by Ian Robertson – available at `https://support.sas.com/resources/papers/proceedings/proceedings/sugi22/SYSARCH/PAPER304.PDF`
- US National Health and Nutrition Examination Survey (NHANES) codebook – available at `https://wwwn.cdc.gov/Nchs/Nhanes/2015-2016/BMX_I.htm#BMXWT`
- FIPS state codes list – available at `https://transition.fcc.gov/oet/info/maps/census/fips/fips.txt`
- *Reducing the CPU Time of Your SAS® Jobs by More than 80%: Dream or Reality?* SAS white paper by Nancy Croonen and Henri Theuwissen – available at `https://support.sas.com/resources/papers/proceedings/proceedings/sugi28/002-28.pdf`
- *Creating and Exploiting SAS® Indexes* SAS white paper by Michael Raithel – available at `https://support.sas.com/resources/papers/proceedings/proceedings/sugi29/123-29.pdf`
- *Power Indexing: A Guide to Using Indexes Effectively in Nashville Releases* SAS white paper by Diane Olson – available at `https://support.sas.com/resources/papers/proceedings/proceedings/sugi25/25/dw/25p124.pdf`
- *Programming Tricks For Reducing Storage And Work Space* SAS white paper by Curtis Smith – available at `https://support.sas.com/resources/papers/proceedings/proceedings/sugi27/p023-27.pdf`
- *WHERE vs. IF Statements: Knowing the Difference in How and When to Apply* SAS white paper by Sunil Gupta – available at `https://support.sas.com/resources/papers/proceedings/proceedings/forum2007/213-2007.pdf`

- 1995 edition of ComputerWorld – available at `https://books.google.com/books?id=OcZZwenlL_0C&q=SAS+Institute#v=snippet&q=SAS%20Institute&f=false`
- *Working with Big Data in SAS®* SAS white paper by Mark Jordan – available at `https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/2160-2018.pdf`
- *From Unstructured Text to the Data Warehouse: Customer Support at the University of North Texas* SAS white paper by Nick Evangelopoulos – available at `https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1900-2018.pdf`
- *Bridging the Gap: Importing Health Indicators Warehouse Data into SAS® Visual Analytics Using SAS® Stored Processes and APIs* SAS white paper by Li-Hui Chen and Manuel Figallo – available at `https://support.sas.com/resources/papers/proceedings16/10540-2016.pdf`
- Video of Dr. Elizabeth Atkinson – available at `https://resources.rstudio.com/rstudio-conf-2018/a-sas-to-r-success-story-elizabeth-j-atkinson`
- SAS documentation about connecting to Snowflake – available at `https://documentation.sas.com/?docsetId=acreldb&docsetTarget=n1d5j8d7wegfezn1irjj3hcrne1n.htm&docsetVersion=9.4&locale=en`