

Minimal-Space Storage for Integer Data

Allan Glaser
Merck & Co., Inc.



Introduction

Modern programs and systems are required to handle large, and growing, amounts of data. The storage, manipulation, and movement of these data frequently present a burden, and it is incumbent upon the analyst and programmer to implement efficient techniques. Data compression has long been used to help alleviate the problem, and various methods are in common use.

If storage space is important, and there are a significant number of variables that can have only integer values, and each of those variables has a small domain of allowable values, then SAS bit level functions can be used to store data in a very small space.

As with any programming technique, there are tradeoffs that must be considered.

2-Valued Variables

Consider any data that can only have two values. Although the real data may not be [0, 1], it can readily be mapped to the [0, 1] domain.

We are accustomed to working with bytes, but [0, 1] data is really the content of a single bit. The values of 32 two-valued variables can be stored in a single num4 variable.

Storing data at the bit level

- may be useful when there are a large number of 2-valued integer variables,
- may require mapping data to the [0, 1] domain,
- can achieve a compression ratio of up to 32:1, and
- requires familiarity with bit-level operations.

If the data are binary, then a single num4 variable can hold 32 different data points. Assuming that the data originally used num4 variables, then the overall storage requirement will drop from $32 * 4 = 128$ bytes to just 4 bytes. The storage requirement is decreased by almost 97%.

To set (give a value of 1) the Nth bit of variable X, the BOR function is used:

$$X = \text{BOR} (X, 2^{**} (N - 1));$$

Exploring this in detail, assume that the 6th bit of variable X should be set. The value $2^{**} (N - 1)$ is a string of 0's with a single 1 in the 6th place from the right. Remember that as with the decimal system, the binary system also places least significant digits to the right. The logical-or operation between the original value of X and this other value will preserve all bits in X except it will force the 6th bit to 1. This can be illustrated pictorially, using an arbitrary value of X. Looking at the eight rightmost bits:

$$\begin{array}{rcl} X & = & \dots 01010101 \\ 2^{**}(6-1) = 32 & = & \dots 00100000 \\ \text{result of BOR} & = & \dots 01110101 \end{array}$$

The BAND function clears (assign a value of 0) the Nth bit of variable X:

$$X = \text{BAND} (X, 2^{**} 32 - 1 - 2^{**} (N - 1));$$

Retrieving the value of the Nth bit of variable X is slightly more complex:

$$Y = \text{BAND} (1, \text{BRSHIFT} (X, N - 1));$$

This approach

- uses a single bit to store the value of each binary variable,
- allows storing up to 32 distinct values in a single 4-byte variable, and
- does not introduce any rounding or truncation errors.

It is important to note that the concept of a missing value disappears. A variable that had two distinct values such as [0, 1] and also allowed missing values would really be a 3-valued variable and would no longer fit into this approach.

3+ Valued Variables

A variable may have n distinct values, which can be mapped to the [0, 1, 2, ..., n-1] domain. This domain requires $\lceil \log_2(n) \rceil$ bits, where $\lceil \rceil$ denotes the CEIL function. For example, a 6-valued variable can be represented in the [0, 1, 2, 3, 4, 5] domain, and those distinct 6 values can be represented by 3 bits.

To load a value of variable X into a specific set of bits within variable Y, the target bits of Y are first cleared, then X is shifted into the proper position and logically-or'ed with Y. To illustrate, suppose X requires n bits, and should be placed into Y starting with the Ath bit position:

$$\text{DO POSITION} = A \text{ TO } (A + N - 1);$$

$$Y = \text{BAND} (Y, 2^{**} 32 - 1 - 2^{**} (\text{POSITION} - 1));$$

END;

$$Y = \text{BOR} (Y, \text{BLSHIFT} (X, A - N));$$

Retrieving the N-bit value from variable Y, starting with the Ath bit position, uses this approach:

$$X = \text{BAND} (2^{**} N - 1, \text{BRSHIFT} (Y, A - N));$$

Note specifically that $(2^{**} N - 1)$ creates a right-aligned string of N 1's.

Domain Size vs. Effectiveness

What happens to the effectiveness of this approach with larger domains?

Within SAS, this approach is constrained to working with 32-bit numeric variables. If all of those 32 bits are used fully, then maximum effectiveness has been achieved. 32 binary variables fully use 32 bits, as do 16 4-valued variables, 8 16-valued variables, etc. Note that in each of these cases, $N * \log_2 B = 32$, where N is the number of constituent variables and B is the number of bits that each requires.

Effectiveness decreases if $N * \log_2 B = 32$ is not satisfied. The worst-case is constituent variables that require 17 bits, and in fact, there is no storage advantage with this approach for such data.

Bit-Sharing

If multiple variables do not fully use the domains that the number of bits allow, then bits can be effectively shared, thus increasing the compression.

To illustrate, assume that we have two 5-valued variables, i.e. each maps to the [0, 1, 2, 3, 4] domain. The previous approach requires a total of 6 bits to store the data; each variable would use 3 bits. However, there are $5^2 = 25$ distinct combinations of values, and only 5 bits can contain $2^5 = 32$ distinct values. The judicious use of exponents and logarithms can allow the original values to be mapped to the [0, 1, 2, ..., 23, 24] domain, and that can be mapped to 5 bits using the techniques above. Storage space is optimized but programming complexity is increased.

Discussion and Summary

Why 32 bits? The SAS functions that manipulate bits work on a 32-bit numeric value. It is possible to use character variables, and lengths other than 4, but those approaches introduce significant complications and difficulties.

No single technique will provide a universal optimal approach for compressing data. Depending upon the attributes and volume of data, bit-level manipulation may be a viable technique to reduce storage space and associated read / write time. This approach requires application of bit-level functions, and works best when there are a large number of limited-domain integer variables. There must be a thoughtful analysis of the space vs. simplicity tradeoffs.

The techniques explored here are lossless, i.e. they do not introduce any loss of information due to rounding, truncation, or similar operations.