Efficient Storage and Processing of Sequential Indicators Using SAS® Bitwise Functions

Gerhardt Pohl, Wenyu Ye, Eli Lilly and Company, Indianapolis, IN

ABSTRACT
Sequential flags (i.e. sequences of true / false indicators) are a commonly encountered data structure in health care information records. For example, in looking at insurance claims data, a sequence of flags may indicate whether a patient is enrolled in a plan during each calendar month. Sequential flags can also be useful in intermediate analysis data sets for marking months in which patients fill prescriptions for a particular drug. Storing these indicators as individual bits rather than characters can dramatically reduce the required disk space. Furthermore, certain operations can be done more efficiently using SAS bitwise functions (BAND, BOR and BXOR) than is possible using character string processing. For example, one can determine the months in which patients filled prescriptions for two different medications by simply “AND’ing” together two bit sequences, an operation that can take place in as little as one cpu operation. Alternative methods for accomplishing a variety of essential tasks such as counting the number of true flags and comparing two sequences to one another are presented. Surprisingly, for some tasks, processing the flags as characters proves to be faster than using bitwise manipulation. Therefore, an optimal solution appears to be storing the information as bits but processing either as bits or as expanded-on-the-fly character strings as appropriate.

INTRODUCTION
When dealing with large data sets, details of data encoding become important for both storage and run time efficiency. We looked at one commonly encountered data structure, sequences of true / false indicators in an effort to optimize efficiency from a balanced storage, processing time and programming effort perspective.

CHARACTERS VS. BITS
Logical flags are typically stored as characters with ‘1’ denoting a true state and ‘0’ denoting a false state. This is not optimal from a storage perspective since on the machine level, each character requires one byte of capacity (i.e., 8 bits). If the bits themselves are used to encode the information, storage requirements are reduced by 87.5% to 1/8 the size of the character representation.

TEST METHODS

IMPLEMENTATION DETAILS
Note that SAS bitwise functions require numerical variables as arguments. This severely limits the length of the bit string that can be directly manipulated with the bitwise functions. To overcome this barrier, we coded bit-level flag sequences as characters, reformating smaller pieces of the character strings to numerical variables as needed before applying bitwise functions. To simplify coding we made the direction of processing across each byte consistent with the direction of processing bytes across a character string, i.e., bits are addressed from left to right within individual characters and left to right across the characters in the string. Storing the bit strings as characters allows practically unlimited length.

TEST DATA AND SYSTEM HARDWARE
Sample data files had 16,777,216 records and varied between 6 to 96 bytes per record. File sizes were, therefore, 0.1 to 1.5 GB. Examples use sequences of 48 flags. This yields a character string of 48 bytes and bit string of 6 bytes.

Detailed hardware configuration was as follows:
SAS 8.2 operating under SunOS 5.8
Sun Microsystems Sun Fire 3800
CPU speed is 750Mhz,
Memory usage set to default values
Disk storage on Clariion CX700. 225 GB per device. RAID 5. Front-end directors are 2 Gbit with 4 GB cache.
Data and SAS workspace set to same directory.

Test runs were done at times of low system load. Results are reported as real time, i.e., the actual time elapsed for the job. Run times are the average of 3 separate runs. Little variability was seen in run times between replications.
TASKS AND STRATEGIES
Three basic strategies were investigated: storing as bits/processing as bits, storing as bits/processing as chars (i.e., generate chars on-the-fly) and storing as chars/processing as chars. These strategies were each applied to two different tasks, counting the number of "on" flags and "AND'ing" two sequences. The strategies were compared in terms of speed, storage space required and programming effort.

COUNT FLAGS OPERATION
The goal of this task is simply to count the number of flags that are true. This is foundational to almost any analysis using sequential flags.

STORE AS BITS/PROCESS AS BITS
As noted in the comment block in the code below, the variable bitflags in the input dataset was created as a 6-character string. This allows for storage of 48 flags in the 48 bits of the variable. To directly test the value of each flag at the bit level, we employ a bit testing constant. In a bit testing constant the mask on the right-hand side of the conditional statement shows the position and bit value that will yield a true result in the comparison. In the first IF-THEN, for example, the result is true if the first bit is 1. Periods in a bitmask are neutral place-holders which yield a true result no matter what the value of the corresponding bit. Therefore, the second IF-THEN yields a true result if the second bit is "1". Each successive bit is tested with the running total being stored in the variable count.

```sas
data _null_; /* format bitflags $6; */
set savedata.shortstring;
count=0;
if bitflags='1'b then count=count+1;
if bitflags='.1'b then count=count+1;
if bitflags='..1'b then count=count+1;
if bitflags='...1'b then count=count+1;
etc.
run;
```

STORE AS BITS/PROCESS AS CHARS
In this strategy, the data is again stored as bits in the 6 character string bitflags as noted in the comment block. Rather than testing the bits directly, the bits are unpacked into characters and tested in a more typical character comparison. The 48-character string charflags holds the unpacked flags. The conversion from bits to chars is easily accomplished with the built-in $BINARY# format. The code simply loops over the individual characters in charflags testing each of the individual character level flags.

```sas
data _null_; /* format bitflags $6; */
set savedata.shortstring;
count=0;
/* format charflags $48; */
charflags=put(bitflags,$binary48.);
do i=1 to 48;
if substr(charflags,i,1)='1' then count=count+1;
end;
run;
```

STORE AS CHARS/PROCESS AS CHARS
In this strategy, the data is converted to characters outside the tested code, i.e., the data is stored as 48-character strings in a disk file. This is very similar to the store as bits/ process as chars scenario except that there is now no requirement to convert the format from numeric to character with the PUT function. There is, however, some potential extra overhead in reading in the larger data file for processing.

```sas
data _null_; /* format charflags $48; */
set savedata.longstring;
count=0;
do i=1 to 48;
if substr(charflags,i,1)='1' then count=count+1;
end;
```
RESULTS FOR COUNT FLAGS OPERATION
Run times are summarized in the table below. Note that storing as bits and operating directly on the bits is considerably slower than the other scenarios. Storing as chars and processing as chars is the fastest scenario. The added overhead of manipulating the eight-times larger file is apparently less than the overhead of converting the bit strings to characters on the fly as in the store as bits/ process as chars scenario. From a balanced storage/speed perspective, however, we would argue that the middle scenario is optimal. It is only slightly slower than the fastest scenario, but still affords a significant savings in storage space. The coding burden of all the scenarios is light. The only added sophistication needed is some familiarity with SAS formats and the PUT function.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Real Time (Minutes:Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store Bits/Process Bits</td>
<td>9:49</td>
</tr>
<tr>
<td>Store Bits/Process Chars</td>
<td>5:21</td>
</tr>
<tr>
<td>Store Chars/Process Chars</td>
<td>4:37</td>
</tr>
</tbody>
</table>

* Optimal Strategy *

“AND” OPERATION
In this task the focus is on manipulating multiple flag sequences. In particular, we compare two strings to determine which positions are '1' in both.

STORE AS BITS/PROCESS AS BITS
In this scenario, we take advantage of the SAS bitwise function BAND. This function takes a numeric argument, requiring us to convert our bit-string character variable to a numeric variable. Note that the maximum length of a numeric variable is limited by system and software architecture. In our case, for example, the largest unsigned integer is limited to 32 bits. For flexibility in applying to other datasets and architectures, we split the character variables one byte pieces. The format each of these pieces is changed using the INPUT function with the built-in informat BITS#.#. The individual numeric bytes are then intersected using the BAND function. Finally, the results are converted back from individual numeric bytes to one-byte characters using the BYTE function; and the characters are reassembled into a continuous character bit string.

```sas
data _null_; format bitband $6.; set savedata.shortstring2; do i=1 to 6;
  * Convert character bit strings to numeric variables;
  num_byte1=input(substr(bitflags1,i,1),bits8.0);
  num_byte2=input(substr(bitflags2,i,1),bits8.0);
  * AND the numeric bits together;
  num_band=band(num_byte1,num_byte2);
  * Load AND result back into bit string;
  substr(bitband,i,1)=byte(num_band);
end;
run;
```

STORE AS BITS/PROCESS AS CHAR
In this scenario, we forgo using the bitwise function, opting instead, to convert the compact bit string character variable into individual character bytes. These are checked one flag at a time using IF-THEN_ELSE logic and converted back to a packed bit-string character variable.

```sas
data _null_;
  format bitband $6.;
  format charband $48.;
  set savedata.shortstring2;
  charflags1=put(bitflags1,$binary48.);
  charflags2=put(bitflags2,$binary48.);
  do i=1 to 48;
    if substr(charflags1,i,1)='1' and substr(charflags2,i,1)='1'
      then substr(charband,i,1)='1';
    else substr(charband,i,1)='0';
  end;
  * Load AND result back into bit string;
  bitband=input(charband,$binary48.);
run;`
```

**STORE AS CHARS/PROCESS AS CHARS**

In this scenario, we eliminate the overhead of format transformations by storing the flags as characters and processing as characters. The code is quite straightforward.

```sas
data _null_;
  format charband $48.;
  set savedata.longstring2;
  do i=1 to 48;
    substr(charflags1,i,1)='1' and substr(charflags2,i,1)='1'
      then substr(charband,i,1)='1';
    else substr(charband,i,1)='0';
  end;
run;`
```

**RESULTS FOR “AND” OPERATION**

Results for this task are decidedly different than from the count bits task. Even with the overhead of multiple format conversions, processing the flags using a bitwise function is much faster than processing as characters with IF-THEN comparisons. The increased speed is most likely the result of the bitwise function operating on eight flags at once rather than having to loop over each flag individually. At the machine level, looping over each character flag requires multiple fetches with the values being loaded to the registers and compared. In contrast AND’ing two bit-level values is a basic machine operation which can be executed in a minimal amount of time. Therefore, for this task one can have the best of both worlds, smaller storage requirements with shorter execution times. The coding is somewhat more complex, but the only added sophistication is using formats and informats to change between variable types. The technique is well illustrated in the examples and easily adaptable, perhaps, even as a short macro.
CONCLUSION
Bit level encoding yields substantial savings in storage – 87.5% reduction compared to character encoding. The optimal processing technique for sequential flags varies, however, with the precise application. Counting flags by inputting bits and generating characters on the fly increases run times 16% compared to inputting characters directly and manipulating the characters. However, storing as bits and generating characters on the fly does preserve storage efficiency and is, therefore, an optimal balanced solution. “AND’ing” two strings at bit level reduces run time by 63% compared to character storage and processing; therefore, one achieves both small storage requirements and faster run times from using bit-level storage and processing. These techniques do not present any major programming barrier. Coding is easily implemented by adapting the code illustrated here.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the authors at:

  Gerhardt Pohl
  Eli Lilly and Company
  Drop Code 4123
  Indianapolis, IN 46285
  pohl_gerhardt@lilly.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration. Other brand and product names are trademarks of their respective companies.