Paper 4838-2020

Slinging Hash: The HASHING functions available in SAS®

Rick Langston

ABSTRACT

This paper provides an overview of the HASHING* functions that have been introduced in 9.4m6 of the SAS System. These functions can perform hashing using MD5, SHA1, SHA256, SHA384, SHA512, and even old-school CRC methods, along with HMAC computations.

INTRODUCTION

A hashing algorithm is one that accepts an input a stream of data (called "the message") and generates a mathematically reduced stream of data of a fixed size (called "the digest"). The purpose of the algorithm is to provide a unique digest for any given message, regardless of the length of the message. This may seem counterintuitive, but hashing algorithms take into account many attributes of messages in order to produce a hopefully unique digest.

This paper describes a recently introduced set of functions available in the SAS System so that the SAS programmer can choose the preferred hashing algorithm and the manner of applying it.

HASHING ALGORITHMS

An early hashing algorithm familiar to many readers of a certain age is CRC (cyclical redundancy check). The digest was 4 bytes, represented by 8 hex digits.

Over time, more complex algorithms have been introduced to improve on the uniqueness of digests, and to make the predictability of digests more difficult. The algorithms available in the SAS System, along with their digest sizes in bytes, are: CRC32 (4), MD5 (16), SHA1 (20), SHA256 (32), SHA384 (48), and SHA512 (64). It is beyond the scope of this paper to explain the inner workings of these algorithms. Suffice it to say that you will choose the algorithm to use based on specifications provided to you.

One should not confuse hashing with encryption. Most typically, encryption converts a plaintext stream into an encrypted stream to be later decrypted by someone who has the decryption key. A hashing algorithm is one-way, in that one cannot convert back to the original message. Instead, a message is converted to a digest, and later on the user provides the same message and expects to obtain the same digest when using the same algorithm.

An example of hashing vs. encryption would be with password storage. Your password can be hashed and its digest stored. All subsequent attempts to enter the password results in hashing and comparing against the stored digest.

THE HASHING* FUNCTIONS

The new functions have been introduced as of 9.4m6 of the SAS System. They all have HASHING as their prefix. And there are legacy functions being carried forward.

Function Name	Purpose
HASHING	Hash a message
HASHING_INIT	Initialize a piecemeal hash
HASHING_PART	Provide part of a piecemeal hash
HASHING_TERM	Terminate a piecemeal hash
HASHING_HMAC	Hash a message using HMAC
HASHING_HMAC_INIT	Initiate a piecemeal hash using HMAC
HASHING_FILE	Hash an entire file at once
HASHING_HMAC_FILE	Hash an entire file at once using HMAC
MD5	Legacy MD5 hashing function
SHA256	Legacy SHA256 hashing function

HASHING is the primary function to use. Its arguments are:

```
digest = hashing(algorithm,text<,flags>);
```

where algorithm is a character string or variable containing the value CRC32, MD5, SHA1, SHA256, SHA384, or SHA512. The text argument is a character string or variable. The entire contents of this string or variable, including any leading and trailing blanks, is the message provided to the hashing algorithm to compute the digest. The optional flags argument value of 1 indicates that the message value is in hex. The return value is always a hex representation of the digest.

Here is an example of invoking the HASHING function with several of the hashing methods.

```
data _null_;
     length method $6 digest $128;
     do method='CRC32','MD5','SHA1','SHA256';
        digest = hashing(method,'this is my message');
        put method ': ' digest;
        end;
     run;
```

```
CRC32 :  F8C1A4BB
MD5 :    36678968064D487BB169300312CBCF2B
SHA1 :   589205D6D61D7C2F0C8D2F4B29D6DB9ACA3A91D7
SHA256 : FEA2EA1F88FF2AB557A8EC16CB7C977886DC2A819430A9615038A9FA922806E3
```

When using the HASHING* functions, the resulting digest is always returns as a string of hex characters. If you need the binary version of the digest, you can use the $HEX informat with the INPUT function, as in:

```
bindata = input(digest,$hex.);
```

HASHING LARGE MESSAGES

Suppose you need to hash a message that is over 32,767 characters. That could really be a problem, being that SAS character variables have a limit of 32,767 characters. However, this is not necessarily a limitation for with the HASHING* functions. These functions support the new type of SAS character variable called a VARCHAR. And a VARCHAR can contain over 2 billion characters!

Here is an example where we hash a message of 1 million concatenated 'x' characters using a VARCHAR.

```
data _null_;
     length x1m varchar(*);
     length status $32;
     x1m='x';
     x1m=repeat(x1m,999999);
     sha256=hashing('sha256',x1m);
     status = ifc
(sha256='1B977E9F84F1B26B6ED7F68B0498FAEE2385EA4125BD29ADCE4A7D9106BA3134',
                  'matched','not matched');
     put status=;
     run;
```

The x1m variable is defined as a VARCHAR in a LENGTH statement, using the VARCHAR(*) keyword. That value in parentheses can be a specific number if you know the maximum length. We first set x1m to a single 'x' character, and then we can use the REPEAT function to create a string of 1 million 'x' characters. Note that the REPEAT function will only produce VARCHAR output if the first argument is a VARCHAR, so that is why we must set x1m first and pass x1m as the first argument. Once we have a VARCHAR value with the 1 million 'x' characters, we can call the HASHING function, which will recognize that the second argument is a VARCHAR. We already know the digest value for this hash, so we can compare it and set our status variable accordingly. And the result seen is

```
status=matched
```

If, for whatever reason, you do not wish to use a VARCHAR, you can use the HASHING* functions to produce a digest in "piecemeal" fashion. Here is an example using HASHING_INIT, HASHING_PART, and HASHING_TERM:

```
data _null_;
     length status $32;
     handle = hashing_init('sha256');
     do i=1 to 1000000 by 10000;
        rc = hashing_part(handle,repeat('x',9999));
        end;
     sha256 = hashing_term(handle);
     status = ifc
(sha256='1B977E9F84F1B26B6ED7F68B0498FAEE2385EA4125BD29ADCE4A7D9106BA3134',
                  'matched','not matched');

     put sha256=;
     run;
```

And as in the previous example, we will see this result:

```
status=matched
```

In this example, we initialize a hash stream with HASHING_INIT, indicating what hashing method is used. A numeric "handle" is returned, which is used in all subsequent calls for hashing. We pass 10,000 'x' characters at a time to the HASHING_PART function, providing the handle. After we've passed all 1 million 'x' characters, we are ready to produce the digest, so we call the HASHING_TERM function with the handle.

It is important to note that this approach can only be used within a single DATA step. You cannot pass the handle from one DATA step to the next, nor save it for later use.

Now suppose that your stream of 1 million 'x' characters is not generated, but instead resides in a file called x1m.txt. You could read in the file and process it via the HASHING* functions already described, but more conveniently you can use the HASHING_FILE function to do this in "one fell swoop" instead.

```
data _null_;
     length status $32;
     sha256 = hashing_file('sha256','x1m.txt');
     status = ifc
(sha256='1B977E9F84F1B26B6ED7F68B0498FAEE2385EA4125BD29ADCE4A7D9106BA3134',
                    'matched','not matched');

     put sha256=;
     run;
```

And as in the previous examples, we will see this result:

```
status=matched
```

HASHING USING HMAC

You may need to follow specifications that involve HMAC (Hash-based Message Authentication Code). HMAC is a public algorithm that uses a "secret key" in addition to hashing. Your specifications will either include this secret key or will describe how to derive it. The HASHING* functions include support for HMAC, as this simple example demonstrates:

```
data _null_;
     text = 'my message';
     secret_key = 'secret';
     sha256=hashing_hmac('sha256',secret_key,text);
     put sha256=;
     handle = hashing_hmac_init('sha256',secret_key);
     do i=1 to length(text);
        rc = hashing_part(handle,substr(text,i,1));
        end;
```

```
      sha256 = hashing_term(handle);
      put sha256=;
      run;
```

We will see these results:

```
sha256=8C4516FBC5E61E76CFC8BBD6CC42F8BEB7776DF08ED57044146B8F2BE308B883
sha256=8C4516FBC5E61E76CFC8BBD6CC42F8BEB7776DF08ED57044146B8F2BE308B883
```

There is also the HASHING_HMAC_FILE function that works the same as HASHING_FILE, shown in this example with the same message and secret key from above:

```
filename myfile temp recfm=f lrecl=1;
data _null_; file myfile;
     put 'my message';
     run;
data _null_;
     sha256 = hashing_hmac_file('sha256','secret','myfile',4);
     put sha256=;
     sha256 = hashing_hmac_file('sha256','secret',pathname('myfile'));
     put sha256=;
     run;
```

We see the same results as above.

Note that HASHING_HMAC_FILE (as well as the previously described HASHING_FILE function) accept a physical file name or a fileref. In the first instance above, the fileref myfile is passed as a character value. The fourth argument is given as 4, which indicates that the third argument is a fileref. The records will be read from the file as per the specifications given in the FILENAME statement. If the fourth argument is omitted, then it is assumed that the third argument is a path name.

MD5 AND SHA256

Although you can using the MD5 and SHA256 hashing methods with any of the HASHING* functions, there are two legacy hashing functions that you may see in older SAS code. The MD5 and SHA256 functions are passed just a message, and they return a binary version of the digest.

```
data _null_;
     message = 'test message';
     length md5bin $16 shabin $32 hexval $64;
     md5bin = md5(message);
     put md5bin=$hex32.;
     hexval = hashing('md5',message);
     put hexval=;
     shabin = sha256(message);
     put shabin=$hex64.;
     hexval = hashing('sha256',message);
     put hexval=;
```

```
run;
```

We see these results:

```
md5bin=C72B9698FA1927E1DD12D3CF26ED84B2
hexval=C72B9698FA1927E1DD12D3CF26ED84B2
shabin=3F0A377BA0A4A460ECB616F6507CE0D8CFA3E704025D4FDA3ED0C5CA05468728
hexval=3F0A377BA0A4A460ECB616F6507CE0D8CFA3E704025D4FDA3ED0C5CA05468728
```

Be careful if you use MD5 or SHA256, in case you have errors and the _ALL_ listing is produced in the log. The results are in binary and could cause unpredictable behavior when viewing these results. This is why I feel it is better to use the HASHING* functions because they always return hex characters for which there will no problem in displaying them.

ENCODINGS AND HASHING

You must be aware that hashing works on binary streams, even if those streams appear to be text characters. These characters can have different binary representations based on the encoding. Here is a simple example to illustrate potential problems.

Suppose you have found a web page for Beyoncé Knowles, and you cut-and-paste her first name (which ends in an acute-accented e) into this program to run in SAS University Edition, using a SAS Studio client with a browser:

```
data _null_;
      text='Beyoncé';
      hash = hashing('md5',text);
      put text=$hex. hash=;
      run;
```

When you run this code, you see the following results:

```
text=4265796F6E63C3A9 hash=DC7D3CAAA4FC4F835E58C233CC1C5EDA
```

But if instead you were to cut-and-paste the code into Display Manager and run it on Microsoft Windows, you'd likely see this instead:

```
text=4265796F6E63E9 hash=52AFE581268F48CB345655EF164ABEA6
```

The hash is not the same, but it appeared that we used the same input text! The reason is seen by examining the hex representation of the text value. All but the last letter has the same hex representation. That is, Beyonc is represented in hex as 4265796F6E63, but the acute accent e is represented in the first case as C3A9 and in the second case as E9. This is because the first case, from SAS Studio, uses UTF-8 encoding, while the second case, using Display Manager, is in Windows Latin1 encoding (at least for me, as a US user).

If you anticipate that you will be working with data that may be represented in multiple encodings, it would be wise to use the KCVT function to ensure that the data are transcoded to a known encoding. If our example changed to:

```
data _null_;
      text='Beyoncé';
      hash = hashing('md5',kcvt(text,'utf8'));
      put text=$hex. hash=;
      run;
```

then the result is the same regardless of whether we run in SAS Studio or Display Manager.

HASHING TEXT FILES ON UNIX™ AND **WINDOWS™**

You may need to be aware of a special circumstance that can cause problems when you are working with text files both on Windows and on Unix. As an example, suppose you have a file called myfile.txt and it resides on Windows, and it contains 2 records with these values:

```
record 1
record 2
```

You use the HASHING_FILE function with the MD5 method and find that the digest is 5E39268C8F6D5C29A1A1E4766178B149. But then you FTP this file over to a Unix environment using text mode, not binary mode. And then you find that the MD5 digest of that file on Unix is 07EE2CDA525EA6C430ACAF81B2A526DE. But if instead you FTP the file over using binary mode, then the MD5 digest is the same as it was on Windows. Why is this?

This happens because the file contains carriage return (hex 0d) and line feed (hex 0a) when created on Windows. But text files on Unix have only line feeds between records. So when you transfer a file in text mode, those hex 0d carriage returns are removed. Conversely, if you transfer text files from Unix to Windows, the 0d carriage returns are added in.

The binary stream is not the same when the carriage returns are removed, so the digest will be different.

USING IN DS2 AND IN A CAS SERVER ENVIRONMENT

You can use all of the HASHING* functions in DS2 as well as the DATA step. You can also use the functions in a DATA step that would run inside a CAS server. The one exception is the HASHING_FILE and HASHING_HMAC_FILE functions if they are used with a fileref argument (that is, the flags argument is 4). This is because the fileref concept does not apply to DS2 nor to a CAS server environment.

USEFUL WEBSITES

You may find it useful to check your work to ensure that you are writing your SAS hashing program correctly. A website I found to be very helpful in verifying results is www.hashemall.com . At this site, you can enter text to be processed using any hashing method that the HASHING* functions use. The text can be cut-and-pasted into a text box, or uploaded as a file. For HMAC testing, I have used https://www.freeformatter.com . In fact, I used both of these websites to check my work for this paper!

CONCLUSION

The HASHING* functions bring together all the most common hashing techniques into one set of functions to allow for immediate hashing, piecemeal hashing, HMAC hashing, and hashing from files.

REFERENCES

www.freeformatter.com

www.hashemall.com

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

RickLangston1955@gmail.com