Paper 1755-2018

Key-Independent Uniform Segmentation of Arbitrary Input Using a Hash Function

Paul Dorfman, Independent Consultant
Don Henderson, Henderson Consulting Services, LLC

ABSTRACT

Aggregating or combining large data volumes can challenge computing resources. For example, the process may be hindered by the system limits on utility space or memory and, as a result, either fail or run too long to be useful. It is a natural inclination to try solving the problem by segregating the input records into a number of smaller segments, process them independently, and combine the results. However, in order for such a divide-and-conquer tactic to work, two seemingly contradictory criteria must be met: First, to aggregate or combine the data correctly, no segment can share its key-values with the rest; and second, the segments must be more or less equal in size. In this paper, we show how a hash function can be used to achieve it for arbitrary input with no prior knowledge of the distribution of the key-values among its records. Effectively, the method renders any task of aggregating or combining data of any size doable by splitting its input into a large enough number of segments. The trade-off is the need to partially re-read the data. However, it is a rather small price to pay to enable a failing or endlessly running task so that it can finish on a timely basis.

INTRODUCTION

SAS programmers dealing with sizeable data are well familiar with hitting the brick wall of hardware inadequate to cope with the data volume to be processed. The dinosaurs among us remember having to maneuver between the Scylla of delivering data on time and Charybdis of sharing mainframe resources. Though the modern machines aim to allay such worries, the growing data volumes and processing demands have been keeping pace with the advances in hardware. As a result, we still run into tasks that cannot be reasonably solved by merely throwing more hardware at them; moreover, the latter is not always possible. Thus, nowadays, just as in the days of yore, programmers often seek their way around hardware limitations via a divide-and-conquer approach.

One of the oldest and well-known tricks of this nature is sorting a file (for example, too large to be sorted outright) by splitting it *arbitrarily* into a number of nearly equal segments, sort them independently, and interleave the results. Other data processing tasks, such as aggregation and merging, can benefit from split processing, too. However, in this case, the segmentation cannot be done arbitrarily, for in order to get the correct final result, no key-value present in one segment must exist in any other. In this paper, we aim to show how a one-way hash function (such as MD5 or SHA256), can be used to both (a) ensure that this condition is met and (b) segment the input reasonably uniformly.

THE PLAN

We will proceed according to the following plan:

- Just to demonstrate the divide-and-conquer principle, consider an example of *sorting* a file by *arbitrarily* splitting it into several segments, sorting them independently, and interleaving the results.
- Show that using a similar approach in order to *aggregate* a file by key is possible if it is segmented, not arbitrarily, but into *key-independent* segments - in other words, in such a way that no two segments share the same key-value.
- Discuss conditions under which the input can be split into key-independent segments *uniformly*, i.e. with more or less the same number of unique key-values in each segment.
- Demonstrate that if such conditions are met, an otherwise arbitrary input can be split into both key-independent and uniform segments by using a one-way hash function with no prior knowledge about the distribution of input key-values.

- Show how to concatenate the components of a composite key for using it in the hash function properly to avoid collisions - i.e., to ensure that different composite key-values map to distinct hash function signatures.

SIMPLE SEGMENTED PROCESSING EXAMPLE: SPLIT SORTING

Let us consider a disordered data set TRANS below:

```
data Trans ;
  input ID $ KEY VAR ;
  cards ;
B  2  1
B  2  2
B  3  2
A  1  3
A  2  1
A  1  3
B  2  3
B  1  3
A  3  2
B  2  2
B  3  1
A  2  3
B  3  2
A  3  2
A  1  3
;
run ;
```

While the file is too miniscule to take advantage of input segmentation, it can be used to demonstrate its principle. Imagine that it represents a severely scaled-down model of a really large file that cannot be sorted head-on due to hardware inadequacies (such as insufficient sort work space, for example). One long in the tooth way around the problem is to arbitrarily divide the file into *N* more or less equal segments and proceed as follows:

1. Sort the first segment into a file forming the "base" of the future sorted file.
2. Sort the next segment and interleave the result with the "base" file.
3. Repeat #2 until the last segment has been processed.

For example, to sort our sample file *Trans* in this manner by splitting it into 3 equal segments, one could code:

```
proc sort data=Trans(firstobs= 1 obs= 5) out=Srt; by ID Key; run;
proc sort data=Trans(firstobs= 6 obs=10) out=Seg; by ID Key; run;
data Srt; set Srt Seg; by ID Key; run;
proc sort data=Trans(firstobs=11 obs=15) out=Seg; by ID Key; run;
data Srt; set Srt Seg; by ID Key; run ;
```

This way, we need only enough sort work space to sort 1/*N* of the input file and store 1/*N* of it (file *Seg*) in the WORK library. Of course, the method of *N*-splitting can be different, and the process can be automated in a variety of ways by generating the needed SAS code (e.g. using a macro). And, of course, such approaches can increase the use of numerous computer resources to compensate for the restrictions based on a specific limited resource (in this example, work space).

Note that for this particular task the segments can be chosen *arbitrarily* (and not necessarily the way it was done above) because the correctness of the final result is ensured by the interleaving. However, arbitrary segmentation will not work for data processing tasks whose partial results cannot be combined by interleaving them. One of such tasks is data aggregation.

SEGMENTED AGGREGATION

Now imagine that the sample file *Trans* above represents a severely downscaled model a large disordered file that needs to be *aggregated*. Suppose, for example, that we need to obtain the sum and count of distinct values of VAR for each unique combination of the key-values (*ID, KEY*). So, we expect the aggregated output to look like the result of the following query:

```
proc sql ;
  create table aggregate as
  select ID, Key, sum(Var) as SUM, count(distinct Var) as UCOUNT
  from   Trans
  group  ID, Key
  ;
quit ;
```

or any other equivalent aggregation technique, such as sort/control-break, aggregating by using the SAS hash object, etc. In other words, we expect the content of the aggregated file to look as follows:

ID	KEY	SUM	UCOUNT
A	1	9	1
A	2	4	2
A	3	4	1
B	1	3	1
B	2	8	3
B	3	5	2

Now suppose that the real file is so large that aggregating it head-on is impossible and we need to find a workaround. Inspired by the input segmentation concept, we may ask if it is possible to apply it for data aggregation just as we did for sorting. To wit, can we split the input file in more or less equal chunks, aggregate them independently and combine the results?

Though *in principle* the answer is "yes", splitting the file *arbitrarily* will not work for data aggregation. It is obvious from a cursory look at file *Trans* as segmented above for sorting:

Obs	ID	KEY	VAR	Segment
1	B	2	1	1
2	B	2	2	
3	B	3	2	
4	A	1	3	
5	A	2	1	
6	A	1	3	2
7	B	2	3	
8	B	1	3	
9	A	3	2	
10	B	2	2	
11	B	3	1	3
12	A	2	3	
13	B	3	2	
14	A	3	2	
15	A	1	3	

For example, taking the records with the key-value (*ID,Key*)=(B,2), we see that they are present in all three segments. Hence, if we aggregate the segments independently and stack the results, we will have three records with (*ID,Key*)=(B,2) in the output, whereas we need just one. Moreover, aggregating the result again to get the keys collapsed to their unique values will work only for the additive statistic SUM; but it will not work in the end for the non-additive statistic UCOUNT.

Apparently, for the split aggregation to work properly, we need to segment the file in such a way that a *given key-value is present in one, and only one, segment*. The way to do it is to base the segmentation, not on arbitrary grouping of the records, but on the key-values themselves. For example, we may notice that in our sample file, the partial key-values *Key*=(1,2,3) are spread more or less uniformly throughout the file. So, we may think of segmenting it as follows:

Obs	ID	KEY	VAR	Segment
4	A	1	3	1
6	A	1	3	
8	B	1	3	
15	A	1	3	
1	B	2	1	2
2	B	2	2	
5	A	2	1	
7	B	2	3	
10	B	2	2	
12	A	2	3	
3	B	3	2	3
9	A	3	2	
11	B	3	1	
13	B	3	2	
14	A	3	2	

Since with such a split any value of partial key *Key* belongs to one, and only one, segment, no composite key-value (*ID,Key*) found in one segment is found in any other segment. Therefore, we can now aggregate each segment independently and simply stack the partial results together. Below, it is done against the sample file *Trans* by aggregating one segment at a time via SQL and appending the output to the combined aggregate:

```
proc sql;
  create table seg as
  select ID, Key, sum(Var) as SUM, count (distinct Var) as UCOUNT
  from   trans where key=1 group ID, Key;
quit;
proc append base=agg data=seg; run;
proc sql;
  create table seg as
  select ID, Key, sum(Var) as SUM, count (distinct Var) as UCOUNT
  from   trans where key=2 group ID, Key;
quit;
proc append base=agg data=seg; run;
proc sql;
  create table seg as
  select ID, Key, sum(Var) as SUM, count (distinct Var) as UCOUNT
  from   trans where key=3 group ID, Key;
quit;
proc append base=agg data=seg; run;
```

Again, this repetitive process can be easily automated for any selected number of segments *N* using a macro or another code-generating SAS tool.

The takeaway from this simple example is that we can take advantage of segmented processing for data aggregation if we can use some part of the processing key in order to:

- Segment the input into key-independent groups of records to guarantee the correctness of the final result. In other words, no key-values in any segment must be present in any other segment.
- Make the segments more or less equally sized to balance the processing workload for each segment approximately evenly.
- For the specific example presented above, it was possible to visually inspect the data and manually assign a value for the segment variable. That is clearly not a practical approach for large and complete data files. Consider, for example, Point of Sale data for a large retailer with a rewards program. Keeping track of total purchases as well as unique visits is one such use case.

SEGMENTATION BASED ON KNOWN KEY PROPERTIES

In our simple aggregation example, we were able to satisfy both segmentation criteria because we *knew ahead of time* how the values of *Key* are distributed across the input file. Sometimes similar favorable situations happen in the real data processing world as well. Namely, we may already know that the data set to be aggregated contains a partial key - or part of it - whose values have been purposely spread approximately evenly across the data collection when it was created. For example:

- A specific position in a retail customer ID number contains a digit whose values from 0 to 9 are known to be distributed evenly across the customer population.
- The values of some key have been created randomly.
- The frequency of key values distribution are available from an already done analysis.

In all such instances, we can merely rely on the known key properties to achieve the key-independent and even segmentation, just as it was done in the aggregation example above.

SEGMENTATION WHEN KEY PROPERTIES ARE UNKNOWN

Most of the time, we do not know which keys and how can be used to achieve key-independent and even segmentation. In principle, an attempt could be made to discover it by performing data analysis on the input keys. However, it takes time and guarantees no success, since we do not know a priori which partial keys (or their combinations thereof to look for). This is exacerbated by the fact that real-world large files likely to benefit from segmented aggregation contain multi-component, high-cardinality composite keys.

On the other hand, this very fact also virtually guarantees that there exists some condition, perhaps involving multiple partial keys, that *could* be used to segment the file as required *if we knew* what it is. The problem is that we do not.

It raises the question: Assuming that the input can be segmented key-independently and evenly, can it be done simply and inexpensively in a "blind" manner, i.e. without any prior knowledge of the key distribution properties? Fortunately, the answer to this question is "yes".

The ability to do this in a “blind” manner addresses a corollary to the known/unknown cases. Suppose we know what the key properties are, but the relative sizes change on a regular basis. Consider the example referenced above where we could assume that a single digit/position in the key was distributed uniformly, but the sizes were still too large even once reduces by a factor of 10. Suppose we need to use 3 adjacent digits/positions but doing that involved uneven group sizes. We need to group these together in approximately equally size chunks.

THE CONCEPT

The idea behind the "blind" segmentation is based on *random deterministic* mapping. To wit:

- Map each key-value of the processing key to alternate, highly random, variable (let us call it *HKey*, say). In the case of our sample file *Trans*, it would mean mapping (*ID,Key*)->*Hkey*.
- Ensure that each unique key-value of the processing key corresponds to one, and only one unique value of the mapping variable. That is, the distinct key-values of (*ID,Key*) must have one-to-one relationship with the respective distinct values of *HKey*.
- Use the values of some part of *Hkey* (e.g., of one of its bytes) grouped into N more or less equal sets or ranges to split the input into *N* segments.

In order to implement this plan, we need a reasonably fast mapping function that can satisfy the requirements listed above. Luckily, SAS has two such functions in its arsenal: MD5 and SHA256.

ONE-WAY HASH FUNCTION TO THE RESCUE

In this paper, we will concentrate on using the MD5 function for input segmentation. (The SHA256 function can be used in the same exact manner with certain distinctions we will briefly discuss near the end.) Let us use an example to see how the MD5 function will map the distinct values of (*ID, Key*) from our sample file *Trans* to its mapping variable *Hkey*:

```
proc sql ;
  create table Map as
  select distinct ID
       , Key
       , MD5 (catx (":", ID, Key)) as Hkey length=16 format=$hex32.
  from   Trans
  order  ID, Key ;
quit ;
```

The expression for *HKey*, above, does the following:

- Concatenates the component keys ID and Key, separating the values by a colon (more on it later).
- Feeds the result into the function MD5. The function generates a character string called "hash signature". Its first 16 bytes are non-blank, and the rest are blank. We are interested only in the non-blank part, hence length=16. (The $hex32. format is used to visualize its characters, some of which may be unprintable.)

The output looks as follows (note that in the table below, the hex digits of *Hkey* are spaced apart artificially for better visual discernability):

ID	KEY	Hkey
A	1	1A A8 1A 75 62 B7 05 FB 67 79 65 5B 8E 40 7E E3
A	2	D6 B3 D7 E5 13 1F 54 1D DE F6 81 D8 AC C1 17 13
A	3	8E 1A 7B 2F 99 09 E6 3C B6 BC D2 2E 7D E8 AB 21
B	1	0E C9 E6 87 5E 4C 6E 67 02 E1 B8 18 13 A0 B7 0D
B	2	B3 0B E9 97 C4 A0 4C 08 09 C2 5D B6 D0 A0 D3 DC
B	3	0E 04 B1 C7 15 01 16 B3 35 E8 56 60 17 29 78 63

A cursory look at the table reveals that there is no regular pattern in the distribution of the values of any particular byte of *HKey*: Because the signature of the hash function MD5 is random, its bytes are random as well. Hence, if we use any of them (or their combination thereof) to separate the key-values of (*ID,Key*) into a number of segments, we can expect the segments to be similar in size, even though no a priori knowledge about the actual key-values of (*ID, Key*) is used to achieve it.

Segmenting the input evenly is one reason to use a hash function such as MD5. Another, even more important, reason is that it provides for splitting the key-values of (*ID,Key*) into *key-independent* segments, i.e. segments mutually exclusive of one another with respect to the key-values they contain. It is made possible by the one-to-one relationship between the distinct values of (*ID,Key*) and those of the hash signature *HKey*. To understand how, suppose that for any *fixed part* of *HKey* (such as a specific byte; let us name it *HKeyPart*, say), we have picked two different values. These values of *HKeyPart* correspond to (a) two different sets of key-values of (*ID,Key*) and (b) different values of *HKey* as a whole. But because of the one-to-one relationship between *HKey* and (*ID,Key*), different values of *HKey* cannot be related to the same key-value of (*ID,Key*). Therefore, a key-value present in one set cannot be present in the other. Q.E.D.

As a corollary, the key-values in any two segments will be also mutually exclusive if we base one segment on *a number* of *HKeyPart* values and base the other on a number of different *HKeyPart* values.

SEGMENTATION NUTS AND BOLTS

Since the values of every byte of the hash signature *HKey* is equally random, the simplest way is to select a byte from *HKey* and work with its values; let us call it just *Hbyte*. A obvious head-on approach is to break the characters in the collating sequence into *N_segments* (non-overlapping) ranges by their hexadecimal values and then select the segment depending on the range into which the hex value of *Hbyte* happens to fall. For example, for *N_segments*=3, we can segment based on the following range logic (note that “00”x thru “FF”x represents 256 distinct values and “00”x-“55”x is 86 of of them; “56”x-“AA”x is 85 of them, and the balance is 256-85-86=85):

```
   proc format; invalue seg "00"x-"55"x=1 "56"x-"AA"x=2 other=3; run ;
```

and then define the expression for the segment to be used in the WHERE clause thus:

```
input (Hbyte, seg.)
```

This way, the three ranges will have (as mentioned above) 86, 85, and 85 values, respectively. However, this approach requires hard coding that needs to be changed depending on the number of segments. Though it can be remedied programmatically, it is much more convenient to base the segregation process, not on the actual character values of *Hbyte*, but on their respective numeric positions in the collating sequence, i.e. on their *ranks*.

This can be done in a variety of ways. However, all of them are based on the fact that a value of any byte in *HKey* is a single character represented by its rank in the collating sequence, i.e. by a specific integer from 0 to 255. In other words, any character value is in effect nothing more than an equivalent of a 256-radix number. In SAS, this rank is returned by either the RANK function or by the PIB*w* informat, so that for any single character *Hbyte*, its rank is determined by either of the expressions:

```
 rank = rank (Hbyte) ;

 rank = input (HByte, pib1.) ;
```

(As we will see later, the functionality of the PIB*w* informat is somewhat richer than that of the RANK function.) Let us apply one of the formulae above to one of the bytes of *HKey* - for example, the 10th:

```
   data Map_rank10 ;
     set Map ;
     Rank = input (char (HKey, 10), pib1.) ;
   run ;
```

As a result, we will get the following picture:

ID	KEY	Hkey	Rank
A	1	1A A8 1A 75 62 B7 05 FB 67 79 65 5B 8E 40 7E E3	121
A	2	D6 B3 D7 E5 13 1F 54 1D DE F6 81 D8 AC C1 17 13	246
A	3	8E 1A 7B 2F 99 09 E6 3C B6 BC D2 2E 7D E8 AB 21	188
B	1	0E C9 E6 87 5E 4C 6E 67 02 E1 B8 18 13 A0 B7 0D	225
B	2	B3 0B E9 97 C4 A0 4C 08 09 C2 5D B6 D0 A0 D3 DC	194
B	3	0E 04 B1 C7 15 01 16 B3 35 E8 56 60 17 29 78 63	232

Now we need some method of turning the ranks into the value ranges. One simple way of doing so is to obtain the remainder of the division of *Rank* by the required number of *N_segments*. For *N_segments*=3:

```
data Map_rank10_segment ;
  set Map ;
  Rank = input (char (HKey, 10), pib1.) ;
  Segment = 1 + mod (Rank, 3) ;
run ;
```

The unity is added to the response of the MOD function merely to get the values of *Segment* in the range of 1-3 instead of 0-2. Now the step generates the following output:

ID	KEY	Hkey	Rank	Segment
A	1	1A A8 1A 75 62 B7 05 FB 67 79 65 5B 8E 40 7E E3	121	2
A	2	D6 B3 D7 E5 13 1F 54 1D DE F6 81 D8 AC C1 17 13	246	1
A	3	8E 1A 7B 2F 99 09 E6 3C B6 BC D2 2E 7D E8 AB 21	188	3
B	1	0E C9 E6 87 5E 4C 6E 67 02 E1 B8 18 13 A0 B7 0D	225	1
B	2	B3 0B E9 97 C4 A0 4C 08 09 C2 5D B6 D0 A0 D3 DC	194	3
B	3	0E 04 B1 C7 15 01 16 B3 35 E8 56 60 17 29 78 63	232	2

Now we can nest it all together and code segmented aggregation thus:

```
%let N_segments = 3 ;
%let X = 1+mod(input(char(MD5(catx(":",ID,Key)),10),pib1.),&N_segments) ;
proc sql;
  create table seg as
  select ID, Key, sum(Var) as SUM, count (distinct Var) as UCOUNT
  from   trans where &X=1 group ID, Key;
quit;
proc append base=agg data=seg; run;
proc sql;
  create table seg as
  select ID, Key, sum(Var) as SUM, count (distinct Var) as UCOUNT
  from   trans where &X=2 group ID, Key;
quit;
proc append base=agg data=seg; run;
proc sql;
  create table seg as
  select ID, Key, sum(Var) as SUM, count (distinct Var) as UCOUNT
  from   trans where &X=3 group ID, Key;
quit;
proc append base=agg data=seg; run;
```

As we have noted above, this type of repetitive code can be automated based on *N_segments* as a parameter - for example, by using a simple macro (which can be further parameterized if desired):

```
 %macro agg (in=, out=, N_segments=) ;
   %local X seg  ;
   %let X=1+mod(input(char(MD5(catx(":",ID,Key)),10),pib1.),&N_segments) ;
   %do seg = 1 %to &N_segments ;
     proc sql;
       create table seg as
       select ID, Key, sum(Var) as SUM, count (distinct Var) as UCOUNT
       from   &in where &X = &seg group ID, Key;
     quit;
     proc append base=&out data=seg; run;
   %end ;
%mend ;
```

```
  %agg (in=Trans, out=Agg, N_segments=3)
```

Depending on the method chosen for aggregation and programmer's personal preferences, other code-assembling methods can be used as well.

Note that we do not have to write out any interim files containing the computed values of *HKey*, *Rank*, or *Segment* - above, they have been used just for the sake of illustrating the concept. Rather, we simply calculate the nested expression *X* to obtain the values of *Segment* entirely on the fly.

Also note that we did not have to select the 10th byte of *HKey* to do the segregation. The only reason we have chosen it above is that with our small sample file *Trans*, it provides for better illustration of the concept, splitting the key-values into the 3 segments most uniformly. With so few distinct input keys to work with, If a different byte were picked, the distribution could miss a segment or get more skewed. However, this is a mere side effect of the small number of input keys: we just do not have enough for the randomness of the hash function to fully manifest itself in every byte of *HKey*. As we will see in the next section, in the situations more closely resembling real life, choosing a particular *Hkey* byte on which to base the segmentation is irrelevant.

SIZE MATTERS

Resorting to the input segmentation techniques described in this paper makes sense under the circumstances where a frontal, single-pass attack fails due to the input data volume. In turn, it means dealing with large input data and multi-component, high-cardinality keys involved in the process. Let us see what happens with the distribution of the *HKey* and *Segment* values when the distinct input data keys are sufficiently numerous and diverse. To do so, we will:

Generate a moderately sized set of composite keys (*ID,Key*) with a random number of distinct *ID* values for each value of *Key*.

- Apply the segmentation process described above to each key (*ID,Key*).
- Observe how many distinct (*ID,Key*) key-values fall into each segment.

First, let us divide the keys into 3 segments, as we did above, and select the first (i.e. leftmost) byte of *HKey* as the segmentation base:

```
%let N_segments = 3 ; * Number of segments ;
%let N_bytes    = 1 ; * Number of leftmost HKey bytes ;
data ID_Key ;
  do ID = "A","B","C","D" ;
    do Key = 1 to ceil (ranuni(1) * 1000) ;
      format HKey $hex32. ;
      HKey = md5 (catx (":", ID, Key)) ;
      Rank = input (HKey, pib&N_bytes..) ;
      Segment = 1 + mod (Rank, &N_segments) ;
      output ;
    end ;
  end ;
run ;
proc freq data = ID_key noprint ;
  tables Segment / out = Freq_&N_segments&N_bytes ;
run ;
```

The FREQ procedure generates the following output:

Segment	COUNT	PERCENT
1	606	33.37
2	610	33.59
3	600	33.04

As we see, if the keys are sufficiently numerous and diverse, basing the segments just on the first byte of the hash function signature spreads the key-values among the segments extremely evenly. Moreover, we do not have to rummage around the actual key-values of (*ID,Key*) looking for some condition to provide for a uniform distribution. Note that the split is very even in spite of the fact that the number of distinct keys in the file is negligibly small compared to real-life, big-file cases. Furthermore, running a frequency on the *Rank* variable reveals that even with such a limited number of input keys (1816, to be exact), the ranks of the first byte of *HKey* more or less evenly hit *every single* collating sequence position from 0 to 255, i.e. every single character from "00"x to "FF"x.

The key point to be made here is that this works regardless of the nature (or industry) of the input data. It works for:

- Point of Sale retail data
- Financial Transactions
- Insurance Claim Data
- Social Security Payments

And so on; and so forth.

HOW MANY BYTES?

An alert reader could ask why then bother with any other byte of *HKey* or involve more than this single byte into the segmentation process by specifying *N_bytes* greater than 1. It is a very good question; and the answer is that most of the time, it is indeed unnecessary. However, sometimes, depending on the input, including more than the first leftmost byte of *HKey* in the process may result in still more even distribution. For example, if for the same set of keys we specified *N_bytes*=3 and reran the program, we would get a slightly better distribution:

Segment	COUNT	PERCENT
1	606	33.37
2	606	33.37
3	604	33.26

Practically speaking, of course, it would not have any significant impact on aggregation performance. But just in case the reader would like to experiment with different combinations of *N_segments* and *N_bytes*, it should be noted that:

- The PIB*w* informat (where the width *w* above is dictated by the value of *N_bytes*) automatically selects the first *w* bytes of *HKey*, so there is no need to use the SUBSTR function before the informat is applied.
- *N_bytes*, i.e. the PIB width *w*, should not exceed 6, for otherwise the response of the INPUT function can exceed the integer precision of the SAS numeric expression. This is because the largest integer returned by PIB*w* equals to (256***w*)-1. Besides, the shorter the informat width *w*, the faster the INPUT function executes.

HOW MANY SEGMENTS?

Depending on the enormity of your input, you may of course elect to split it into more than 3 segments and thus trade more than 2 extra passes through the input data for the ability to get the job ultimately done. In principle, even using just the first byte of *HKey* can provide for splitting the input into up to 256 key-independent segments. Using two bytes of *Hkey* can provide for 256**2-way splitting, and so forth.

However, it is hard to imagine that splitting the input in so many segments could be practically necessary, no matter how huge the input could be. Though splitting it into a greater number of smaller segments reduces the resources needed to process each segment roughly proportionally to *N_segments*, each extra segment also means another pass through the input file. Thus, the number of segments needs to be chosen judiciously to (a) make each segment small enough for the system resources to be able to handle its processing and (b) avoid overtaxing the system with unnecessary extra I/O.

For example, if the aggregation is done using the hash object, the most significant resource hurdle to overcome is its memory footprint dictated by the need to ultimately store in memory all unique values of (*ID,Key*), which is necessary to calculate the "count distinct" statistic. If it required, say, 256G of RAM, while we had only 64G available, choosing *N_segments*=8 could be a reasonable compromise because it would reduce the memory footprint needed to process one segment to 32G at the expense of 8 passes through the data. In addition, the cost of the increased I/O is somewhat mitigated by using the WHERE clause to subset the input data for each particular segment.

SEGMENTATION PROCESSING: RECAP

If you need to aggregate data so large that it does not yield to a frontal attack, it can be divided-and-conquered by splitting the input into a number of nearly equal key-independent segments without any a priori knowledge about the key-values involved in aggregation in the following manner:

1. Select the number of segments, *N_segments*, in such a way that the system resources can comfortably deal with processing 1/*N_segments* of the input. The trade-off is *N_segments* passes through the data using the WHERE clause to subset it for each specific segment.
2. Concatenate the processing key components and pass the result to the MD5 function to obtain its signature, *HKey*.
3. Select up to 6 leftmost bytes of *Hkey*. In most cases, the first byte alone will more than suffice. Due to the randomness of *HKey*, it does not matter which *HKey* bytes to use, so using the leftmost ones is just simpler (and faster).
4. Calculate the numeric equivalent (denoted above as *Rank*) of the combined character value of the selected bytes. Regardless of the number of bytes chosen, the PIB*w* informat (with the width *w* equal to the number of selected bytes) will return the value of *Rank* in one fell swoop.
5. Create a numeric expression returning a value of *Segment* from 1 to *N_segments* depending on *Rank* and *N_segments*. It can be done in a number of ways, including an informat, the MOD functions, etc.
6. Compose an expression (let us call it *X*) combining all of the above.
7. Process the first segment as you would process the unsegmented input by letting WHERE *X*=1 and append the result to a file you want to hold the final aggregated result. Then process the second segment in the same manner by letting WHERE *X*=2, and so forth until all segments have been processed.

Note that once the hash signature has been obtained, there are many ways to segregate the input limited only by the creativity of the programmer. For example, if you have decided to use a single byte of HKey (say, the leftmost), you may:

Elect to eschew the calculation of *Rank* altogether and act directly upon the character values of the byte.

Use the $HEX*1* informat to extract the first half of the byte, which is wide enough for a 16-way split.

Use the BITS*w* informat instead of the PIB*1* informat to obtain the numeric equivalent (the *Rank* variable, above) of the desired number of *w* leftmost bits. Thus, one can choose *w*=1 (good for up to a 2-way split), *w*=2 (4-way), *w*=3 (8-way), and so on till *w*=8 (256-way). It may come in handy if one should decide to split the input exactly into 2, 4, 8, etc. segments, because in this case, *Rank* can be used directly as *Segment* without the need to compute it in the item #5 above.

Let your imagination fly.

DOES THE AGGREGATION METHOD MATTER?

Not as far as the input segmentation concept is concerned. Using it to make the available system resources sufficient for processing each separate segment works regardless of the technique chosen to aggregate the data. Although above, the concept was illustrated by using the SQL procedure, it would work just the same if the data aggregation were done using a different technique, such as the sort/control-break, the SAS hash object, the MEANS procedure (except that the latter cannot calculate the "count distinct" statistic), etc.

For instance, if we wanted to aggregate using the sort/control-break method instead of SQL, we would only have to replace the SQL step in macro SEGMENT above with the following, leaving everything else intact:

```
proc sort data = &in (where=(&X = &i)) out=seg ;
  by ID Key Var ;
run ;
data seg (drop=Var) ;
  do until (last.Key) ;
    set seg ;
    by ID Key Var ;
    SUM = sum (SUM, Var) ;
    UCOUNT = sum (UCOUNT, first.Var) ;
  end ;
run ;
```

Now it is only natural to ask whether data aggregation is the only data processing task, to which this divide-and-conquer paradigm is applicable. The answer is: But of course not; any task where processing is done by a key can benefit from key-independent uniform segmentation if its input is too large for taking on it head-on. Merging (aka joining) data is another typical example.

SEGMENTED MERGE/JOIN

At this point, we already have in our arsenal all the tools do perform any merge/join using input segmentation in the same manner as we have done it with data aggregation. To illustrate how it can be done, let us create another small sample file in addition to file *Trans*:

```
data Extra ;
  input ID:$1. Key Extra:$2. ;
  cards ;
B 2 E6
A 0 E0
B 1 E2
A 1 E3
B 3 E4
B 7 E7
A 2 E1
A 3 E5
;
run ;
```

Suppose that we want to equi-join file *Trans* with file *Extra* by (*ID,Key*). Using a head-on SQL approach, we could code:

```
proc sql ;
  create table Join as
  select trans.ID, trans.Key, trans.Var, extra.Extra
  from   Trans, Extra
  where  trans.ID = extra.ID and trans.Key = extra.Key ;
quit ;
```

Expectedly, the output, where the non-matching rows are eliminated (equi-join), will look as follows:

ID	KEY	VAR	EXTRA
B	2	1	E6
B	2	2	E6
B	3	2	E4

ID	KEY	VAR	EXTRA
A	1	3	E3
A	2	1	E1
A	1	3	E3
B	2	3	E6
B	1	3	E2
A	3	2	E5
B	2	2	E6
B	3	1	E4
A	2	3	E1
B	3	2	E4
A	3	2	E5
A	1	3	E3

Let us assume, as we have done before, that files *Trans* and *Extra* are merely severely scaled-down counterparts of two real files with many more observations and variables, too large to yield to the single-pass frontal attack approach above. Furthermore, suppose that if we split the processing in 2 independent segments (and hence 2 passes through each side of the join), we would have enough resources to attain the goal. The plan is simple:

1. Segregate Trans into 2 key-independent uniform segments using the hash function.
2. Use exactly the same algorithm to split Extra into 2 segments.
3. Join segment 1 of Trans to segment 1 of Extra. Add the result to the initially empty final joined file.
4. Join segment 2 of Trans to segment 2 of Extra. Add the result to the final joined file.

Since we already have all the nuts and bolts of the algorithm from the aggregation section, we can proceed directly to macro coding:

```
 %macro join (in=, out=, N_segments=) ;
     %local X seg  ;
     %let X=1+mod(input(MD5(catx(':',ID,Key)),pib1.),&N_segments) ;
     %do seg = 1 %to &N_segments ;
       proc sql ;
         create table seg as
         select trans.ID, trans.Key, trans.Var, extra.Extra
         from   Trans (where=(&X = &seg))
              , Extra (where=(&X = &seg))
         where  trans.ID = extra.ID and trans.Key = extra.Key ;
       quit ;
       proc append base=&out data=seg; run;
     %end ;
   %mend ;
   %join (in=Trans, out=Join, N_segments=2)
```

Again, as in the case of data aggregation, the segmentation process is independent from the particular method used to execute the join. For example, if we decided to use the SAS hash object to join the files, we could merely replace the entire SQL step with the following DATA step:

```
   data seg ;
         if _n_ = 1 then do ;
           if 0 then set Extra (keep=Extra) ;
           dcl hash h (dataset:"Extra(where=(&X=&seg))", multidata:"Y") ;
           h.defineKey ("ID", "Key") ;
           h.defineData ("Extra") ;
           h.defineDone () ;
         end ;
```

```
          set Trans (where=(&X=&seg)) ;
          if h.find() = 0 ;
        run ;
```

Regardless of the joining method, the segregation process remains the same: Join every segment of *Trans* to the corresponding segment of *Extra*. Note that because the segmentation is key-independent and done identically in both files, the algorithm is valid since only the key-values in the *corresponding* segments of *Trans* and *Extra* can possibly match.

A sample use case here might be processing a large fact table in a Star Schema data warehouse: partition the fact table so it can be joined with the required dimension tables as part of the data aggregation process.

ENSURING ONE-TO-ONE KEYS-TO-SIGNATURE MAPPING

As explained above, the fundamental premise on which the key independence of the segmentation process rests is the strict one-to-one correspondence between the key-values of the original processing key, such as (*ID,Key*), and the values of the hash function signature *HKey*. In order to achieve it, two conditions must be met:

1. The hash function itself must pair any value of its argument to a distinct value of its signature related to this, and only this, argument value.
2. The response of the function used to concatenate the components of a composite processing key, such as CATX, must correspond to one, and only one, key-value of any given composite key.

Let us consider these two conditions separately.

HASH FUNCTION MAPPING

A big fuss has been made of the fact that under some very esoteric circumstances and with a big deal of computing effort, the MD5 function can be in principle forced to return the same signature for two distinct values of its argument. Such an event is termed a *collision*. Based on this fact, the function has been declared "unsafe" by puritans of cryptographic rigor (particularly those who believe in the inviolability of Murphy's law). However, let us take a step back and examine what it means from the standpoint of real-world data processing we, the SAS programmers, have to deal with.

In the worst case scenario, the approximate number of distinct arguments that need to be hashed to get a 50 percent chance of an MD5 collision is about 2**64≃2E+19. It means that to encounter just 1 collision, the MD5 function has to be executed against 200 quintillion distinct arguments, i.e. approximately 1 trillion times per second for 100 years. To put it into common sense perspective, the lifetime odds of being struck by a meteorite is 1 in 1.6E+6 (one in 1.6 million), i.e. about 2 trillion times greater. Practically speaking, it means that in the data processing world such as that we are dealing with here, an MD5 collision will never occur.

If after this logical excursion you are still worried about getting a MD5 collision and a resulting data processing inaccuracy, the SHA256 hash function can be used instead. This function is known to be free of collisions even theoretically. For our purposes, it work exactly the same as MD5, except that its hash signature non-blank length is $32, and so the length of *HKey* should be sized accordingly; for example:

```
      format HKey $hex64. ;
      HKey = sha256 (catx (":", ID, Key)) ;
```

So, if SHA256 is bulletproof, why not do away with MD5 and use SHA256 instead at all times? The answer is that SHA256 is much slower to execute. How much slower? Here is a table showing what happens when both MD5 and SHA256 are executed 1 million times each on the X64_7PRO platform against the same non-blank arguments of various lengths:

Argument Length	SHA256 Time	MD5 Time	SHA256 to MD5 Speed Ratio
2	7.53	0.25	1:30
4	7.13	0.29	1:25

Argument Length	SHA256 Time	MD5 Time	SHA256 to MD5 Speed Ratio
8	7.36	0.20	1:37
16	6.76	0.19	1:36
32	7.14	0.19	1:38
64	7.91	0.39	1:20

Since our goal here is improving performance, in our opinion using SHA256 instead of MD5 for the sake of preventing an infinitesimally small chance of collision at the expense of dramatically poorer performance makes no practicable sense.

CONCATENATION MAPPING

If our processing key is composite, such as the (*ID,Key*) above, its components need to be concatenated before they are passed as an argument to the hash function. The mapping of the composite key-value to the concatenated value must be strictly one-to-one; otherwise, the one-to-one mapping of the composite key-value to the hash signature will be compromised. There are two reasons why two *distinct* composite key-values can end up being mapped to the same concatenated value:

- The CATX buffer length is shorter than the actual concatenated value.
- The value of a partial key component being concatenated begins or ends with the same character as the concatenation delimiter (i.e. the value of the first CATX argument).

Let us look at these two possible issues and the methods of addressing them separately.

Insufficient CATX Buffer Length

If we do not size the length of the CATX expression (otherwise termed the length of the CATX buffer) beforehand and leave it to the finction's own devices, the buffer length is set to the default value of 200. If the total length of the concatenated value, including the delimiters, should exceed 200, its characters beyond 200 will be truncated. In this case, if two concatenated values are the same all the way up to the 200th position and differ only in the positions 201 and above, the truncation will chop the discriminating part of the value off. As a result, two distinct composite key-values will map to the same concatenated value. Therefore, in order to avoid that, we must make sure that the CATX buffer length is sufficiently longer than 200 to hold the concatenated value without truncating it.

If, on the other hand, we do size the CATX buffer ahead of time, we must ensure, once again, than its length is sufficient to hold the concatenated value without truncation. There are two ways to set the CATX buffer length.

First, we can create a separate appropriately sized character variable and assign the response of the CATX function to it, then pass the variable to the hash function. For example, if the composite key consists of *N* components *K*1-*K*n and it has been determined that their summary concatenated length (including the delimiters between them) will not exceed 256, then:

```
length _catx $ 256 HKey $ 16 ;
drop _catx ;
_catx = catx (':', K1, K2, ... , Kn) ;
HKey = MD5 (_catx) ;
```

Alternatively, with the aid of the $*w* format, it can be done without using *_catx* and/or prior sizing of *HKey*:

```
HKey = put (put (catx (':', K1, K2, ... , Kn), $256.), $16.) ;
```

A propos, this entire expression can be passed directly to the segmentation expression without assigning it to *HKey* first.

There may arise a well-understood temptation to avoid determining how much buffer length is actually needed and cover all the bases by setting the CATX buffer length to the maximum of $32767. However, it is not a good idea because the time needed to execute the CATX function (or any of its feline siblings) is proportional, not to the length of the actual concatenated value, but to the buffer length. For instance, if we execute two expressions:

```
c1 = put (catx (':', "A", "B"), $64.) ;
c2 = put (catx (':', "A", "B"), $32767.) ;
```

enough times to detect the difference, we will see that it will be at least 1:20. Thus, it pays to invest some time into figuring out the shortest buffer length long enough to prevent any concatenated values from being truncated. This is all the more true that in most cases, it can be determined programmatically by using the metadata related to the key components being concatenated.

Delimiters and Endpoints Conflation

The desire to eliminate concatenation collisions is the very reason a delimiter is used to separated the concatenated key components. Indeed, suppose that we have a 2-component composite key with two *different* composite key-values ("A1","B") and ("A","1B"). If we then concatenate the components of each without a delimiter, such as by using the expressions:

```
catS ("A1", "B") = "A1B"
catS ("A", "1B") = "A1B"
```

then both will result in the same concatenated value "A1B", thus producing a collision. However, if we use a delimiter, for example, a colon, then each will map to its unique concatenated value:

```
catX (':', "A1", "B") = "A1:B"
catX (':', "A", "1B") = "A:1B"
```

Great, is it not? Yes, it is - except in the case when the endpoints of the values being concatenated may contain the same character as the delimiter itself, for example:

```
catX (':', "A:", "B") = "A::B"
catX (':', "A", ":B") = "A::B"
```

So, using a delimiter in and by itself by no means guarantees one-to-one mapping. Of course, if you know that the keys in your data never contain some specific character, it can be used safely. For example, in our sample files *Trans* and *Extra* the colon character, ':' is not part of any key-value, which is why it could be used with impunity.

However, as it happens in the real world, virtually any character can be part of some key-value's endpoint. Does it mean that there is no way to avoid mapping collisions, no matter what delimiter is used?

Fortunately, there is a way around this seemingly crippling limitation. To see what it is, note that it is only the *endpoint characters* that cause the problem. Hence, if, for each concatenated component, we forcibly create new endpoints *different* from the delimiter, we can eliminate the issue altogether. Namely, before concatenating the components, we can always *surround* each of them with a character (or characters) *different from the delimiter*. For instance, let us surround each component in the example above with a pair of parentheses:

```
catX (':', '('||"A:"||')', '('|| "B"||')') = "(A:):(B)"
catX (':', '('||"A" ||')', '('||":B"||')') = "(A):(:B)"
```

This way, the concatenation mapping will be always one-to-one, as long as the delimiter and the surrounding characters are different, since it guarantees that *no newly created endpoint character is the same as the delimiter character*. Moreover, the surrounding characters do not even have to be different from one another like the parentheses above. In fact, the same character can be used on both sides as long as it is not the same as the delimiter. Programmatically, such surrounding can be done in a variety of ways, which we leave to the reader's creativity and imagination.

CONCLUSION

The subject of this paper did not originate from theorizing about "what happens if your input data are so big that ...". Rather, it is a result of a practical attempt on part of the authors to solve a real-world data aggregation problem for a client whose data were too large to yield to a frontal SQL attack and, to some degree, even to the power of the SAS hash object. The authors hope that the technique presented in the paper may one day save it for someone trying to get the job done when the data volume and the nature of the task, on one hand, and the available machine resources, on the other hand, seem to be at irreconcilable odds.

ACKNOWLEDGMENTS

The authors wish to thank Pete Lund for the invitation to share the content of this paper with the SAS Global Forum audience.

They would also like to thank our clients at a specific (un-named insurance company) for their diligent efforts to work with as we developed this approach. They know who they are and that is, frankly, all that matters.

.

CONTACT INFORMATION

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

Paul Dorfman
Proprietor, Dorfman Consulting
4437 Summer Walk Ct
Jacksonville, FL 32258
904-260-6509
sashole@gmail.com

Don Henderson
Henderson Consulting Services, LLC
3570 Olney-Laytonsville Road, #211
Olney, MD 20830
Phone: (301) 980-9027
Fax: (301) 576-3781
Email: Don.Henderson@hcsbi.com
Web: http://www.hcsbi.com
Blog: http://hcsbi.blogspot.com